Editors: Daniel Moldt and
Heiko Rölke

Proceedings of the International Workshop on

P etri
N ets and
S oftware
E ngineering

PNSE'14

University of Hamburg
Department of Informatics

These proceedings are published online by the editors as Volume 1160 at

CEUR Workshop Proceedings
ISSN 1613-0073
http://ceur-ws.org/Vol-1160

Preface

These are the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'14) in Tunis, Tunisia, June 23–24, 2014. It is a co-located event of *Petri Nets 2014*, the 35th international conference on Applications and Theory of Petri Nets and Concurrency and *ACSD 2014*, the 14th International Conference on Application of Concurrency to System Design. More information about the workshop can be found at:

`http://www.informatik.uni-hamburg.de/TGI/events/pnse14/`

For the successful realisation of complex systems of interacting and reactive software and hardware components the use of a precise language at different stages of the development process is of crucial importance. Petri nets are becoming increasingly popular in this area, as they provide a uniform language supporting the tasks of modelling, validation, and verification. Their popularity is due to the fact that Petri nets capture fundamental aspects of causality, concurrency and choice in a natural and mathematically precise way without compromising readability.

The use of Petri Nets (P/T-Nets, Coloured Petri Nets and extensions) in the formal process of software engineering, covering modelling, validation, and verification, will be presented as well as their application and tools supporting the disciplines mentioned above.

The program committee consists of:

Hanna Klaudel (Université d'Evry-Val d'Essonne, France)
Radek Kočí (Brno University of Technology, Czech Republic)
Lars Kristensen (Bergen University College, Norway)
Michael Köhler-Bußmeier (University of Applied Science Hamburg, Germany)
Niels Lohmann (University of Rostock, Germany)
Robert Lorenz (University of Augsburg, Germany)
Daniel Moldt (University of Hamburg, Germany) (Chair)
Berndt Müller (University of South Wales, United Kingdom)
Chun Ouyang (Queensland University of Technology, Australia)
Wojciech Penczek (ICS PAS and Siedlce UPH, Poland)
Laure Petrucci (University Paris 13, France)
Lucia Pomello (Università degli Studi di Milano-Bicocca, Italy)
Heiko Rölke (DIPF, Germany) (Chair)
Christophe Sibertin-Blanc (Université Toulouse 1, France)
Mark-Oliver Stehr (SRI International)
Harald Störrle (Technical University of Denmark, Denmark)
Eric Verbeek (Eindhoven University of Technology, Netherlands)
Jan Martijn van der Werf (Utrecht University, Netherlands)
Manuel Wimmer (Vienna University of Technology, Austria)
Karsten Wolf (University of Rostock, Germany)

There is one invited talk by Lars Kristensen from Bergen University College, Norway. We received more than 28 high-quality contributions. For each paper three to four reviews were made. The program committee has accepted five of them for full presentation. Furthermore the committee accepted 13 papers as short presentations and one short paper. Several more contributions were submitted and accepted as posters.

The international program committee was supported by the valued work of following sub reviewers: Sofiane Bendoukha, Maximilien Colange, Tadeusz Czachorski, Markus Huber, Yasir Khan, Görkem Kılınç, Luca Manzoni, Artur Niewiadomski, and Józef Winkowski. Their work is highly appreciated.

Furthermore, we would like to thank our colleagues in the local organization in Tunis, Tunisia, for their support.

Without the enormous efforts of authors, reviewers, PC members and the organizational team this workshop wouldn't provide such an interesting booklet.

Thank you,

Daniel Moldt and Heiko Rölke Hamburg, June 2014

Contents

PNSE'14 Proceedings

Contents

Part III PNSE'14: Short Presentations

Part I

PNSE'14: Invited Talk

An Approach for the Engineering of Protocol Software from Coloured Petri Net Models: A Case Study of the IETF WebSocket Protocol

Lars Michael Kristensen

Department of Computing, Bergen University College, Norway
Email: lmkr@hib.no

Invited Talk

The vast majority of software systems today can be characterised as concurrent and distributed systems as their operation inherently relies on protocols executed between independently scheduled software components. The engineering of correct protocols can be a challenging task due to their complex behaviour which may result in subtle errors if not carefully designed. Ensuring interoperability between independently made implementations is also challenging due to ambiguous protocol specifications. Model-based software engineering offers several attractive benefits for the implementation of protocols, including automated code generation for different platforms from design-level models. Furthermore, the use of formal modelling in combination with model checking provides techniques to support the development of reliable protocol implementations.

Coloured Petri Nets (CPNs) [3] is formal language combining Petri Nets with a programming language to obtain a modelling language that scales to large systems. In CPNs, Petri Nets provide the primitives for modelling concurrency and synchronisation while the Standard ML programming language provides the primitives for modelling data and data manipulation. CPNs have been successfully applied for the modelling and verification of many protocols, including Internet protocols such as the TCP, DCCP, and DYMO protocols [1, 4]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols considered and have resulted in improved protocol specifications. However, earlier work has not fully leveraged the investment in modelling by also taking the step to automated code generation as a way to obtain an implementation of the protocol under consideration.

In earlier work [5], we have proposed the PetriCode approach and a supporting software tool [7] has been developed for automatically generating protocol implementations based on CPN models. The basic idea of the approach is to enforce particular modelling patterns and annotate the CPN models with code generation *pragmatics*. The pragmatics are bound to code generation templates and used to direct a template-based model-to-text transformation that generates the protocol implementation. As part of earlier work, we have demonstrated the use of the PetriCode approach on small protocols. In addition, it has been shown that our approach supports code generation for multiple platforms, and that it leads to code that is readable and also compatible with other software [6].

In the present work we consider the application of our code generation approach as implemented in the PetriCode tool to obtain protocol software implementing the IETF WebSocket protocol [2] protocol for the Groovy language and platform. This demonstrates that our approach and tool scales to industrial-sized protocols. The WebSocket protocol is a relatively new protocol and makes it possible to upgrade an HTTP connection to an efficient message-based full-duplex connection. WebSocket has already become a popular protocol for several web-based applications such as games and media streaming services where bi-directional communication with low latency is needed.

The contributions of our work include showing how we have been able to model the WebSocket protocol following the PetriCode modelling conventions. Furthermore, we perform formal verification of the CPN model prior to code generation, and test the implementation for interoperability against the Autobahn WebSocket test-suite [8] resulting in 97% and 99% success rate for the client and server implementation, respectively. The tests show that the cause of test failures were mostly due to local and trivial errors in newly written code-generation templates, and not related to the overall logical operation of the protocol as specified by the CPN model. Finally, we demonstrate in this paper that the generated code is interoperable with other WebSocket implementations.

Acknowledgement. The results presented in this invited talk is based on joint work with Kent I.F. Simonsen, Bergen University College and the Technical University of Denmark, and Ekkart Kindler, the Technical University of Denmark.

References

1. J. Billington, G.E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, 2004.
2. I. Fette and A. Melnikov. The websocket protocol, 2011. `http://tools.ietf.org/html/rfc6455`.
3. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
4. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Transactions on Petri Nets and Other Models of Concurrency VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
5. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.
6. K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *To appear in Proc. of PNSE'14*, 2014.
7. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *SEFM 2013 Collocated Workshops: BEAT2, WS-FMDS, FM-RAIL-Bok, MoKMaSD, and OpenCert*, volume 8368 of *LNCS*, pages 151–166. Springer, 2014.
8. Tavendo GmbH. *Autobahn|Testsuite*. `http://autobahn.ws/testsuite/`.

Part II

PNSE'14: Long Presentations

Verification of Logs - Revealing Faulty Processes of a Medical Laboratory

Robin Bergenthum and Joachim Schick

Department of Software Engineering and Theory of Programming
FernUniversität in Hagen
{robin.bergenthum,joachim.schick}@fernuni-hagen.de
http://www.fernuni-hagen.de/sttp

Abstract. If there is a suspicion of Lyme disease, a blood sample of a patient is sent to a medical laboratory. The laboratory performs a number of different blood examinations testing for antibodies against the Lyme disease bacteria. The total number of different examinations depends on the intermediate results of the blood count. The costs of each examination is paid by the health insurance company of the patient. To control and restrict the number of performed examinations the health insurance companies provide a charges regulation document. If a health insurance company disagrees with the charges of a laboratory it is the job of the public prosecution service to validate the charges according to the regulation document.
In this paper we present a case study showing a systematic approach to reveal faulty processes of a medical laboratory. First, files produced by the information system of the respective laboratory are analysed and consolidated in a database. An excerpt from this database is translated into an event log describing a sequential language of events performed by the information system. With the help of the regulation document this language can be split in two sets - the set of valid and the set of faulty words. In a next step, we build a coloured Petri net model corresponding to the set of valid words in a sense that only the valid words are executable in the Petri net model. In a last step we translated the coloured Petri net into a PL/SQL-program. This program can automatically reveal all faulty processes stored in the database.

1 Introduction

A lot of information systems are used in the healthcare sector and each system produces some kind of log-data. This is particularly true in the domain of medical laboratories where all samples, materials and examination results have to be stored. This "good laboratory practice" is an important method of quality management and big medical laboratories own records about several millions of processed orders.

Every examination performed by a medical laboratory is paid by a health insurance company. The cost of each examination is rated by a fixed scale of charges given in a so-called charges regulation document. Of course, the correct

application of the regulations have to be proven to the health insurance companies. If a suspicion about irregular application of the regulations arises, it is the job of the public prosecution service to validate the billed charges according to the regulation document. Usually, the prosecution service orders a report investigating the issue from an expert-office.

In this case study we describe an approach using coloured Petri nets which is inspired by the methods of the area of process mining and process discovery to reveal faulty processes given in log-files of a medical laboratory. The files contain data recorded over a period of five years having 1500-2000 orders a day. Each order consisting of 20-30 events, examinations and results. Altogether, we face about 100 million lines of log that need to be analysed and verified. Each line describes an event or a sub-process of the medical laboratory. Each event refers to the occurrence of an action of the information systems and is annotated with a time stamp, order-id, variables etc. Typical actions of the system are *register order*, *register requirements*, *register examination results*, *validate results*, *make invoice*, *archive order* .... In addition to these basic actions, a medical laboratory is able to perform a huge number of different examinations. In this case study the prosecution service ordered a report revealing all faulty processes concerned with Lyme disease.

To reveal all faulty processes of a set of log-files we choose a four step approach. We call the first step *consolidation step*. The main goal is to develop a schema to integrate all the recorded files into a relational database. Using the same schema it is also easy to implement a view on top of the database tables. The view abstracts from redundant or superfluous information and reduces the data to events and results corresponding to processes considering Lyme disease. With the help of this view we are able to produce an event log, i.e. a sequence of events bearing only information about order-number, time-stamp and result.

The next step is called the *formalization step*. Each sequence of events corresponds to a sequence of actions. Each sequence of actions is called a word. The set of words is called the language of the event log. The main task in the formalization step is to split this language into two sublanguages, the set of valid and the set of faulty words. This has to be done manually with the help of the charges regulation document. Of course, this is a time consuming task, but we believe that it is very easy and hardly error-prone to classify single words. We could also try to directly build a model of regulations from the charges regulation document to classify the set of words automatically, but often the regulation document is given as plain text. Starting from such a description is error-prone and easily yield a model that does not fit the recorded event log regarding names of actions, values and level of abstraction. Remark, we only need to partition the set of words, we do not classify the complete event log. In the formalization step a set of valid sequences of actions is produced. We call this set the language of regulations.

The third step is called *integration step*. The language of regulations is integrated into a coloured Petri net model. Such an integration can be supported by synthesis or workflow mining algorithms. In our case study the language of

regulations is already highly compressed and settled, such that we construct a corresponding coloured Petri net model by hand using the editor CPN-Tools [1]. The constructed coloured Petri net model is a formalization of the charges regulation document using the language of the recorded files. Only valid process instances of the Lyme disease diagnostic processes are executable in this Petri net model. A big advantage of such a Petri net model is that it can be analysed, simulated and verified.

The fourth step is called *implementation step*. Coloured Petri nets are well readable and have an intuitive formal semantic. We will show how to translate such a coloured Petri net model into a PL/SQL-program. We translate transitions to functions, places to tables and arcs to delete or insert statements. With the help of such a PL/SQL-program all sequences of events can be replayed in the database. If the replay fails, the sequence corresponds to the occurrence of a faulty process of the medical laboratory.

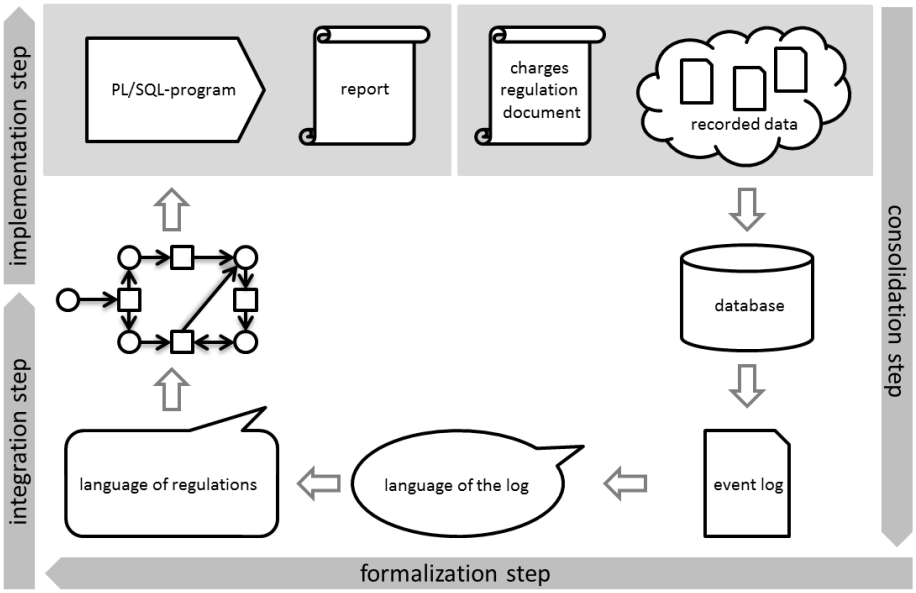


Fig. 1. Approach to reveal faulty processes

Figure 1 depicts an overview of the presented approach. A key feature is that it is built on a chain of formal models. The initial models, i.e. the schema, the view and the event log, consolidate the recorded data. Afterwards, the language of regulations, the coloured Petri net and the PL/SQL-program are build. The constructed models document of the whole inspection procedure, all results can easily be reconstructed, the produced models can be reused when inspecting other laboratories. Of course, stepping from one formal model to another highly supports the validity of the investigation report produced. Each step can be

supported by algorithms and tools. Some steps can even run fully automated using e.g. synthesis algorithms for the construction of the Petri net model or automated generation of the PL/SQL-program.

The chosen approach is inspired by techniques well known in the area of process mining where some recorded behaviour is merged into a formal model of the underlying process [2–4]. Remark, that it is of great importance to choose an appropriate process mining algorithm that does not introduce much additional behaviour to the model. There are language base discovery algorithms [5–7] or even synthesis algorithms [8–11] that meet this requirement. The approach is also inspired by work done in the field of business process modelling and requirements engineering were the starting point of the discovery phase is the construction of a formal and valid specification [12–16]. Nevertheless, there are two major points that are unusual to approaches known in both areas. We model the process of the underlying system by coloured Petri nets since they highly depend on the intermediate results of a chain of different blood examinations. In addition the formal language of the event logs needs to be filtered by hand according to the charges regulation document. This step can not be automated and is crucial for the quality of the report produced.

The paper is organized as follows: Section 2 provides formal definitions. Section 3 presents the approach and our case study. In Section 4, we sum up the results to prove the applicability of the developed approach.

2 Preliminaries

In this section we briefly recall the basic notions of languages, event logs and coloured Petri net.

An alphabet is a finite set A. The set of words over an alphabet A is denoted by A^*. The empty word is denoted by λ. A subset $L \subseteq A^*$ is called language over A.

Business processes describe the flow of work within an organisation [17]. Each process consist of a set of activities that needs to be performed. We denote T the set of all activities and call the execution of an activity an event. Events are labelled with the name of the corresponding activity. Furthermore, events can carry a time stamp showing the time of execution and values denoting results of the execution. We denote V the set of values. A set of events corresponding to the occurrence of a processes is called a case. Recording the behaviour of a system yields a set of interleaved cases we call an event log.

Definition 1. *Given a finite set of activities T, a finite set of values V and a finite set of cases C. An element $\sigma \in (T \times V \times C)^*$ is called an event log. Fix a case $c \in C$ we define the function $p_c : (T \times V \times C) \to (T \times V)$ by*

$$p_c(t, v, c') = \begin{cases} (t, v) & \text{,if } c = c' \\ \lambda & \text{,else.} \end{cases}$$

Given an event log $\sigma = e_1 \dots e_n \in (T \times V \times C)^*$ we define the language $L(\sigma)$ of σ by $L(\sigma) = \{p_c(e_1)...p_c(e_i) | i \leq n, c \in C\} \subseteq (T \times V)^*$.

The language of an event log is finite and prefix closed. It reflects the control flow between activities given by the events of the log. Each case adds a word to the set of words called language.

In this paper we use coloured Petri nets to model valid behaviour of a medical laboratory. The underlying Petri net models the control flow between actions while variables control the examination results. The following definition of coloured Petri nets was given in [1].

Definition 2. *A coloured Petri net is a tuple* $CPN = (P, T, F, \Sigma, V, D, G, E, I)$, *where:*

- P *is a finite set of places.*
- T *is a finite set of transitions, such that* $P \cap T = \emptyset$ *holds.*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs.*
- Σ *is a finite set of non-empty colour sets.*
- V *is a finite set of typed variables such that* $Type[v] \in \Sigma$ *for all variables* $v \in V$.
- $D : P \to \Sigma$ *assigns a colour set to each place.*
- $G : T \to EXP_V$ *assigns a guard to each transition* t *such that* $Type[G(t)] = Bool$.
- $E : F \to EXP_V$ *assigns an arc expression to each arc* f *such that* $Type[E(f)] = D(p)_{MS}$, *where* p *is the place connected to the arc* f.
- $I : P \to EXP_0$ *is an initialisation expression to each place* p *such that* $Type[I(p)] = D(p)_{MS}$.

In contrast to low-level Petri net a place of a coloured Petri net belongs to a given type called colour. According to this colour each place carries values called tokens. Arcs carry variables and if an arc is connected to a place, the tokens of the place can bind to variables of the arc. A binding b of a transition maps variables of related arcs into values of related places. A transition t is executable if there is a binding b such that the transition guard evaluates to $true$. When the transition occurs, as for low-level Petri net, it removes the specified tokens from the input places and produces tokens in the output places (see [1] for a formal definition).

The initialisation function I assigns tokens to places yielding an initial marking. Given a coloured Petri net CPN a sequence of sequential enabled transitions is called an occurrence sequence of CPN. In this paper we add the values of the respective bindings to each transition of an occurrence sequence. The language $L(CPN)$ of CPN is defined as the set of all occurrence sequences. Given an event log $log \in (T \times V \times C)^*$, log is executable in CPN if $L(log) \subseteq L(CPN)$ holds.

3 Verification of Logs

In this section we present an approach to validate a set of given recorded files with the help of a regulation document. In the following case study, on behalf of the public prosecution service, recorded data of an information system of a

medical laboratory has to be reviewed. During a period of five years 1800 files were produced and recorded. Each file contains about 1500 processed orders. The regulation document is given by a charges regulation document provided by health insurance companies. The goal is to identify faulty processes performed by the medical laboratory considering all processes corresponding to Lyme disease diagnostic. An overview of our approach is sketched in Figure 1 given in the introduction. The subsections of this section reflect the four steps of our approach.

3.1 Consolidation Step

In a first step the recorded files need to be consolidated and formalized. The aim of this step is to load the recorded files into a database to extract an event log from it afterwards. For the storage and processing of data, the commercial Oracle Database is used. This database system provides a procedural programming language named PL/SQL for the implementation of the stored procedures. To set up the database an entity-relationship diagram is produced. Of course, to produce this diagram first the recorded files need to be reviewed. Afterwards, we use the Oracle SQL Developer Data Modeler to construct the model.

An excerpt of a file recorded by the medical laboratory is depicted in Figure 2. All files of the laboratory's information system have a hierarchical structure with a flexible record length up to 1024 characters. Each file is a sequence of different types of blocks. Each block corresponds to a set of different actions of the system. The first line of each block is the header of the block and all following lines are indented.

The file depicted in Figure 2 starts with a block corresponding to the registration of a new order for a blood count. The header of this block reads as follows: The first number corresponds to the registration-id 727980834 generated for this new order. This id perfectly fits the need to identify cases in the given file. In our case study each registration-id corresponds to a case of the system. The next two numbers refer to the time the registration occurred, i.e. January 25th 2011, 11:49:54 in our example. The next two strings indicate that this action was manually triggered. The last number of the header encodes the name of the action occurred. In this particular information system the number 10 refers to the action *order blood count*. The inner lines of this first block carry the values of this registration action. Possible values are the name, birthday and address of the patient registered.

The next block corresponds to the scheduling of examinations. The header refers to the same case as the first registration block since both ids match. Remark, both recorded actions even occurred within the same second. The difference between both headers is only given by the number at the end of the line. In this block 20 refers to the action *schedule examination*. This block consists of two sub-blocks, both sub-blocks marked by the keyword BORR. BORR stands for Lyme disease and indicates that the scheduled examinations are part of Lyme disease diagnosis. Again, the inner lines carry values of the scheduling where BORG and BORM are abbreviations of two different blood examinations.

```
727980834       250111  114954  SF      erfass  10
        NAME    [redacted]
        GEBDAT  06.01.1979
        ...
727980834       250111  114954  SF      erfass  20
        BORR    BORG            0000
                RESTYPE W
                ST_BA[1]        X
                ST_BA[4]        X
                MVALBER LA
                UNTVERS 0000
        BORR    BORM            0000
                RESTYPE W
                ST_BA[1]        X
                ST_BA[4]        X
                MVALBER LA
                UNTVERS 0000
702673748       250111  115004  SF      erfass  10
        NAME    ...
        ...
702673748       250111  164235  MB      onlval  21
        JB      FT3
                WERT    2.8
                ...
702984083       250111  174847  MB      onlval  21
        TSH1    TSH1
                WERT    0.63
        ...
727980834       270111  123344  US      onlval  21
        BORR    BORG            0000
                WERT    < 10.0
                ST_RES  J
                ST_BA[1]        R
                ST_BA[4]        R
                ST_MVAL J
        BORR    BORM            0000
                WERT    < 18.0
                ST_RES  J
                ST_BA[1]        R
                ST_BA[4]        R
                ST_MVAL J
        ...
727980834       270111  195346  AB      abschl  13
        ...
727980834       280111  071406  HW      rechdr  11
        RNR     KV110128
        ROK     OK
```

Fig. 2. An excerpt of a recorded file of the medical laboratory.

In this example the block corresponds to the occurrence of two different actions. A BORG-examination and a BORM-examination is scheduled.

The sixth block shown in Figure 2 corresponds to the recording of results of the scheduled examinations. The number 21 refers to the action *receive result*. This block matches the schedule examination block besides two important differences. First, the keyword ONLVAL indicates that this event was automatically triggered by the information system when the results of examinations are received. Second, the inner lines of the block carry the results of these examinations. In this example the results of the BORG- and the BORM-examinations are received. The value of the BORG-examination is smaller than 10.00 and the value of the BORM-examination is smaller than 18.00. Both values show the absence of the corresponding antibodies, i.e. both examinations are negative and no further examinations need to be scheduled.

After knowing the structure of the files an entity-relationship diagram is built. With the help of this schema a PL/SQL-program is written to load all files into the Oracle Database. If all the data is stored, the next step is to extract a consolidated and formal event log from this database. The event log only contains events and values corresponding to processes that need to fulfil regulations given in the charges regulation document concerning Lyme disease diagnostic. We omit a detailed description of the produced entity-relationship diagram, but give a short impression in Figure 3.

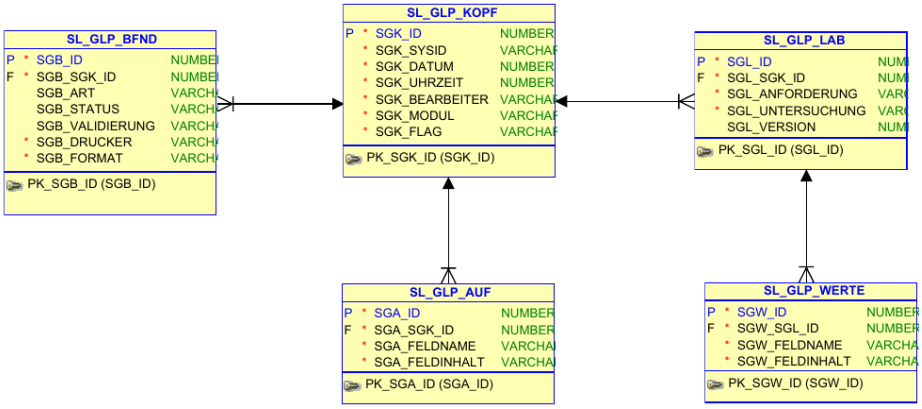


Fig. 3. Entity-relationship diagram of our Oracle Database.

Given the entity-relationship diagram it is easy to implement a view on top of the tables of the database to receive an appropriate event log. In our example, the excerpt depicted in Figure 2 only contains four blocks corresponding to Lyme disease diagnostic. In the first and in the third block two new orders arrived and both patients are registered. In the second block a BORG- and a BORM-examination for the first order is scheduled. The sixth block shows the results

of both examinations. We are able to discard all other blocks shown in Figure 2. If we apply the constructed view to this excerpt we get the event log shown in Table 1. This event log abstracts from additional events and values. It shows the six events corresponding to the four blocks concerned with Lyme disease of Figure 2.

id	action	value	stamp
727980834	10		25.01.11, 11:49:54
727980834	20 BORG		25.01.11, 11:49:54
727980834	20 BORM		25.01.11, 11:49:54
702673748	10		25.01.11, 11:50:04
727980834	21 BORG	< 10	27.01.11, 12:33:44
727980834	21 BORM	< 18	27.01.11, 12:33:44

Table 1. Event log of the file depicted in Figure 2.

With the help of the Oracle Database and the implemented view arbitrary extracts of the recorded files can be shown as event logs. These logs are the results of the consolidation step of our approach. In the next steps these logs are filtered with the help of the regulation document and integrated to an executable model.

3.2 Formalisation Step

In the second step of our approach first the event log is used to define the formal language of the recorded behaviour. Then, in a next step, this behaviour is filtered with the help of the charges regulation document yielding a language of valid words. The aim of this formalisation step is to bring together the recorded behaviour and the regulation document given as plain text. Remark, that it is much easier to only evaluate the recorded language with the help of the regulations and not to build an independent model of all regulations hoping it will fit the language of the recorded behaviour.

To deduct a formal language from the event log, first the actions of the system need to be identified. In our case study the list of significant actions reads as follows:

$$\begin{aligned} T = \{&10, 20BORG, 20BORM, 20BVLSEG, 20BP39G, 20BP83, 20BIV1, \\ &20BIV2, 20BIV3, 20BIV4, 20BOSPC, 20BVLSEM, 20BP39M, \\ &21BORG, 21BORM, 21BVLSEG, 21BP39G, 21BP83, 21BIV1, \\ &21BIV2, 21BIV3, 21BIV4, 21BOSPC, 21BVLSEM, 21BP39M\} \end{aligned}$$

The numbers 10, 20 and 21 indicate if a blood count for a patient is registered, an examination is scheduled or if a result is received. The attached letters are

the abbreviations of the corresponding examinations. There are 12 different tests corresponding to Lyme disease diagnostic leading to 25 different actions in total. Every action having a name starting with 21 carries a value of the type *boolean* (i.e. either the examination is negative or positive). At this point we are able to abstract from any other value given in the files such that any other action occurs without additional data. As stated above all events having the same registration-id belong to the same case. Events of the same case can be ordered by their time stamp. If we apply this knowledge to our event log we get a set of words. The following table shows three example words given by the event log of our case study:

$L(log) = \{$10 20BORG 20BORM (21BORG,false) (21BORM,false),

10 20BORG 20BORM 20BVLSEG 20BP39G 20BP83 20BIV1 20BIV2 20BIV3 20BIV4 20BOSPC 20BVLSEM 20BP39M (21BORG,true) (21BORM,true) (21BVLSEG,false) (21BP39G,false) (21BP83,false) (21BIV1,false) (21BIV2,false) (21BIV3,false) (21BIV4,false) (21BOSPC,false) (21BVLSEM,false) (21BP39M,false),

10 20BORG 20BORM 20BVLSEG 20BP39G 20BP83 20BIV1 20BIV2 20BIV3 20BIV4 20BOSPC 20BVLSEM 20BP39M (21BORG,false) (21BORM,false) (21BVLSEG,false) (21BP39G,false) (21BP83,false) (21BIV1,false) (21BIV2,false) (21BIV3,false) (21BIV4,false) (21BOSPC,false) (21BVLSEM,false) (21BP39M,false),

...

Table 2. The language of the event log.

The language depicted in Table 2 was automatically processed from the given event log. This language is a complete and formal description of the set of processes occurred in the information system of the medical laboratory. Any new sequence of actions and values given by the events of a case yields a new word in the language of the log. Of course, the language of the log is much smaller than the event log since cases corresponding to the same process are not distinguished.

Given the language of the log the next step is to distinguish valid and faulty words. This is a major task in the presented approach which can not be automated. The charges regulation document is given as text. It is absolutely necessary to understand the given regulations and apply them to the set of words. The main advantage of the presented approach is that the set of words is given in a very compact and formal style. There is no room for interpretations or ambiguities. The rules of the charges regulation document do not need to be modelled explicitly, they just need to be applied to the given language. As stated in [13, 16] a single word is much easier to understand than a whole system. The evaluation of single words can be performed by experts on the regulation document. There is no need that these experts know how to model a system or even can read the files or know how the information system works.

In our example given in Table 2 the first two words are valid. The third word is faulty since the set of examinations {BVLSEG, BP39G, BP83, BIV1, BIV2, BIV3, BIV4, BOSPC, BVLSEM, BP39M } may only be preformed if one of the BORG- and BORM-examination is positive. According to the regulation document the blood count needs to be performed in two steps. First, the BORG- and BORM-examination results need to be evaluated, if one of these is positive, a more detailed set of examinations should be performed.

The result of the formalization step is the set of valid words. This set can be seen as the relevant part of the language of the charges regulation document given in the language of the information system. If this language is found, the most challenging task of the investigation process has been completed. In the next steps this set is integrated into an executable model.

3.3 Integration Step

The third step of our approach is called integration step. The aim is to build an executable model having the language of the charges regulation document. As suggested in [18] it would be possible to skip this integration step and just filter the event log with the help of the set of valid words given by the language constructed in the former step, but there are mainly two important reasons to build an integrated model first. A model provides a more compact representation of the set of words such that the model can more easily be simulated and analysed. For this purpose there exist a lot of well known Petri net algorithms in the literature. Second, an executable model can easily be translated into executable code in the last step of our approach.

The problem of integrating a set of words into a Petri net is a well known problem. There exists a lot of work tackling the problem in the area of process mining [2, 19, 20, 7] and in the area of language based synthesis [8, 21, 11, 9, 6]. Algorithms from both areas can by applied to support the integration step. In the presented case study we built the corresponding model by hand. The constructed language of the charges regulation document was already compressed in such a way that there was no need for automated integration. At first, a transition is constructed for every action of the given language. According to the ordering of actions given in the language places are added to this set of transitions such that only words of the language are executable in the resulting net. In a second step the values carried by actions yield colourseds added to the constructed Petri net. Variables are added to arcs connected with the respective transitions corresponding to actions carrying a value. The coloured Petri net is adjusted in such a way that each pair of an action and value given in the language corresponds to a transition and a binding. In a last step, like it is common for coloured Petri nets, it is possible to merge some transitions. Similar parts of the Petri net are folded yielding additional coloured tokens representing each part. For modelling we use CPN-Tools [22, 23]. CPN-Tools is developed at the AIS group of the Technische Universiteit Eindhoven and supports all editing and simulation features for coloured Petri net.

In our case study our initial low-level Petri net contains 25 transitions corresponding to the 25 actions of our process identified in the formalization step. The control flow is rather simple and we just add the corresponding places. First, a blood count have to be registered, then an arbitrary number of the 12 examinations concerning Lyme disease can be scheduled. The execution of these 12 examinations must follow the simple rule, that first the BORG- and BORM-examination need to be performed before the other examinations occur. Remark, the control flow of the initial low-level net is independent form the values given in the language. Rules and regulation concerning values are added in the next step. All actions that corresponds to an examination result carry a value. For this reason we introduced a *boolean* colourset called RES and allow each such transition to be executed while binding to $true$ or $false$. At this point we are able to require that a BORG- or BORM-examination must be positive before any other examination can be executed. In a last step we folded transitions if possible. The resulting net is depicted in Figure 4.

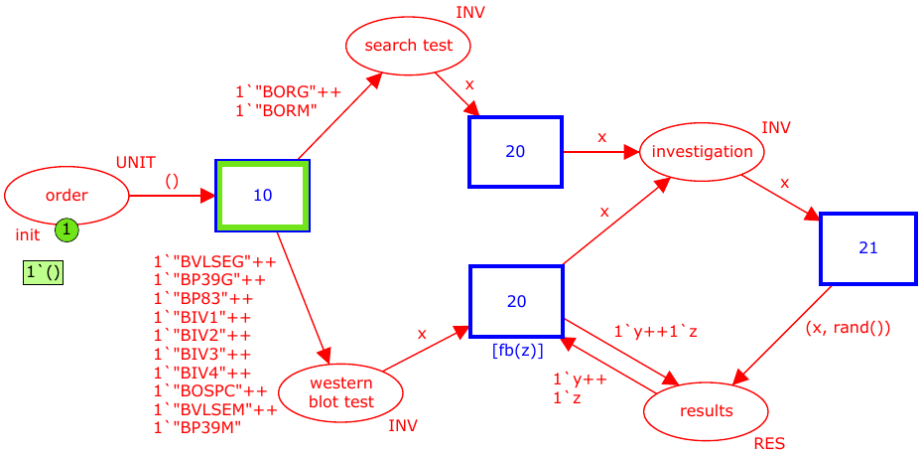


Fig. 4. Coloured Petri net representing the charges regulation document.

In Figure 4 the transition named 10 is enabled in the initial marking. If transition 10 fires, a BORG- and a BORM-token is produced in the place *search test* and tokens corresponding to all other examinations are produced in the place *western blot test*. In such a marking only the upper transition 20 is enabled. If transition 20 fires a BORG- or a BORM- examination is scheduled. As soon as an examination is scheduled transition 21 is enabled. If transition 21 fires, it consumes a token from the place *investigation* and moves this token to the place *results*. While the token is moved a random boolean value is attached. The lower transition named 20 is enable if the western blot tests are scheduled and if there are at least two tokens in the places *results*. The arc inscription $1'y + +1'z$ denotes a pair of tokens. One token is assigned to the variable y

and another token is assigned to the variable z. The guard [fb(z)] ensures that the token called z carries the value *true*. It follows that in the model shown in Figure 4 the western blot tests can only be preformed if the results of the BORG- and BORM-examination are present and at least one of these examinations was evaluated with *true*.

The model shown in Figure 4 is only able to reproduce one single run of the information system. In some sense it is a model of valid words, not a model of the running information system. Our goal is to replay each case of the event log in this model, there is no need to construct a model which is able to handle multiple cases at once.

Besides the possibility to validate the produced model by simulation, CPN-Tools provides some model checking algorithms (see [1] for details). Table 3 depicts a small part of the CPN-Tools state space report of the model shown in Figure 4.

Liveness Properties ————————
Dead Transition Instances: None
Live Transition Instances: None
Fairness Properties ————————
No infinite occurrence sequences.

Table 3. CPN-Tools state space report of the model shown in Figure 4.

The integration step yields a sound and integrated model of the valid language produced during the formalization step. Of course, if analysing this model uncovers faults or additional requirements, the language produced in the formalization step needs to be adopted according to the change made in the model. If model and language match and describe the valid behaviour of the underlying charges regulation document, in the last step of our approach, the model is translated into executable PL/SQL-code.

3.4 Implementation Step

The fourth and last step of our approach is called the implementation step. Although, the coloured Petri net model is executable we translate the produced Petri net into PL/SQL-code. PL/SQL is a proprietary programming language which is integrated in the Oracle Database. Since it can execute SQL statements directly it is more suitable than Java or C++ in our approach. The aim is to get an executable program directly running next to the recorded data. With the help of this program faulty processes preformed by the medical laboratory can automatically be revealed.

During the case study the coloured Petri net model depicted in Figure 4 is transformed into PL/SQL mainly using the following ideas:

(i) Each place of the coloured Petri net yields a temporary table in the database. The tables are able to store records representing tokens and their values.

(ii) Each transition of the coloured Petri net yields a parametrized function in the database. A function returns *true* only if the corresponding transition is executable. To check if a transition is executable arcs of the Petri net are translated into SQL-statements. Roughly speaking, these statements check if there exist appropriate values in the tables corresponding to places in the preset of the transition.
(iii) Each arc of the coloured Petri net yields an SQL-statement in the database. Arcs leading from a place to a transition correspond to DELETE-statements consuming tokens from tables. Arcs leading from a transition to a place correspond to INSERT-statements producing tokens in tables.
(iv) Each guard or function of the coloured Petri net yields a function in the database. The SML-functions given in the coloured Petri net can easily be translated.

CPN	PL/SQL-program
COLOUR	a list of attributes
PLACE	a table having a COLOUR
TOKEN	a record in PLACE
PT-Arc	DELETE from PLACE return TOKEN
TP-Arc	INSERT into PLACE values TOKEN
EXPRESSION	a WHERE expression
TRANSITION	a function using ARCS
GUARD	a sub-function of TRANSITION

Table 4. Pattern of transformations form a CPN into a PL/SQL-program.

A table of transformation patterns is given in Table 4. With the help of these transformation rules it is even possible to implement a fully automated transformation procedure.

To actually verify the event log with the help of the PL/SQL-program the set of cases of the event log is replayed. The registration-ids of the set of faulty cases is stored in an additional table. With the help of this procedure faulty processes can be revealed. The set of faulty processes is the basis of the report produced for the prosecution service. The specific results produced in our case study are presented in the next section.

4 Results and Conclusion

In the context of the presented case study, a set of files of a medical laboratory has been verified. The files record all occurred actions of the information system of the laboratory over a period of five years. In that given period, 22432 orders of Lyme disease diagnostic have been performed by the laboratory. The PL/SQL-program produced by our approach calculates the following results:

recorded processes	valid	faulty	runtime
22432	3311	19121	11 minutes

Table 5. Results of the presented case study.

As shown in Table 5 only 15% of the recorded behaviour is valid according to the charges regulation document. It turned out, that the considered laboratory in almost every case performed the complete set of 12 examinations in a first step. The regulations require that the BORG- and BORM-examination precede all other examinations. Only if one of the two examinations is positive, the set of all examinations can be charged.

To get a more detailed view on the recorded data, in a second step, we adjusted our coloured Petri net model. We removed the transition guard requiring a positive result from one of the BORG- or BORM-examination, assuming a more sloppy interpretation of the regulation document. If we repeat the validation procedure we get that 50% of all recorded processes are valid concerning this more liberal model. In other words, even if we allow that all 12 examinations can be performed at once, 50% of all processes contain additional faults like unnecessary actions or manual changing of examination values.

In the paper we presented an approach together with a case study to verify logs revealing faulty processes of a medical laboratory. The produced PL/SQL-program can directly be applied to any medical laboratory using the same information system. The main advantage of the presented approach is that it is based on a chain of formal models. With the help of these models it is easy to keep track of the validity of the produced report. Most of the steps can be supported using algorithms or tools well known in the area of Petri nets. Furthermore, experts on the regulation document can support the formalisation step without any knowledge about modelling techniques. If a model is produced it also can be adopted and reused. This can help to generate different criteria regarding only parts of the regulation document. Of course, all calculated results can be reproduced at any time, if this is required by the public prosecution service.

From the experience we gained in the case study we feel that the approach forces us to tackle the given task in a very structured way. The approach provides good documented, traceable results. In the future, we will test the presented approach on a larger regulation document yielding a larger regulation model and try to automate each step of the approach further.

References

1. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
2. van der Aalst, W.M.P.: Process Mining - Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
3. van der Aalst, W.M.P., Adriansyah, A., van Dongen, B.F.: Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
4. Rozinat, A.: Process Mining: Conformance and Extension. PhD thesis, TU Eindhoven (2010)
5. van Dongen, B.F., van der Aalst, W.M.P.: Multi-phase Process Mining: Building Instance Graphs. In Atzeni, P., Chu, W.W., Lu, H., Zhou, S., Ling, T.W., eds.: ER. Volume 3288 of Lecture Notes in Computer Science., Springer (2004) 362–376
6. Bergenthum, R., Mauser, S.: Mining with User Interaction. In Desel, J., Yakovlev, A., eds.: Proceedings of the Workshop Applications of Region Theory, Petri Nets 2011. Volume 725 of CEUR Workshop Proceedings. (2011) 79–84
7. Bergenthum, R., Mauser, S.: Folding Partially Ordered Runs. In Desel, J., Yakovlev, A., eds.: Proceedings of the Workshop Applications of Region Theory, Petri Nets 2011. Volume 725 of CEUR Workshop Proceedings. (2011) 52–62
8. Badouel, E., Darondeau, P.: Theory of Regions. In Reisig, W., Rozenberg, G., eds.: Petri Nets. Volume 1491 of Lecture Notes in Computer Science., Springer (1996) 529–586
9. Bergenthum, R., Desel, J., Mauser, S., Lorenz, R.: Construction of Process Models from Example Runs. Petri Nets and Other Models of Concurrency **2** (2009) 243–259
10. Darondeau, P.: Synthesis and Control of Asynchronous and Distributed Systems. In Basten, T., Juhás, G., Shukla, S.K., eds.: ACSD, IEEE Computer Society (2007) 13–22
11. Bergenthum, R., Desel, J., Kölbl, C., Mauser, S.: Experimental Results on Process Mining Based on Regions of Languages. In: Proceedings of the Workshop CHINA, Petri Nets 2008, China (2008) 73–87
12. Glinz, M.: Improving the Quality of Requirements with Scenarios. In: Second World Congress on Software Quality, Yokohama (2000) 55–60
13. Desel, J.: From Human Knowledge to Process Models. In Kaschek, R., Kop, C., Steinberger, C., Fliedl, G., eds.: UNISCON. Volume 5 of Lecture Notes in Business Information Processing., Springer (2008) 84–95
14. Weske, M.: Business Process Management - Concepts, Languages, Architectures, 2nd Edition. Springer (2012)
15. Mayr, H.C., Kop, C., Esberger, D.: Business Process Modeling and Requirements Modeling. In: ICDS, IEEE Computer Society (2007) 8
16. Mauser, S., Bergenthum, R., Desel, J., Klett, A.: An Approach to Business Process Modeling Emphasizing the Early Design Phases. In: Proceedings of the Workshop Algorithmen und Werkzeuge für Petrinetze. Volume 501 of CEUR Workshop Proceedings. (2009) 41–56
17. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes - A Petri Net-Oriented Approach. Cooperative Information Systems series. MIT Press (2011)
18. Harel, D.: Come, Let's Play - Scenario-based Programming using LSCs and the play-engine. Springer (2003)

19. van der Werf, J.M.E.M., van Dongen, B.F., Hurkens, C.A.J., Serebrenik, A.: Process Discovery using Integer Linear Programming. Fundam. Inform. **94**(3-4) (2009) 387–412
20. IEEE Task Force on Process Mining: Process Mining Manifest. In Daniel, F., Barkaoui, K., Dustdar, S., eds.: Business Process Management Workshop. Volume 99 of Lecture Notes in Business Information., Springer (2012) 169–194
21. Bergenthum, R., Desel, J., Lorenz, R., Mauser, S.: Process Mining Based on Regions of Languages. In Alonso, G., Dadam, P., Rosemann, M., eds.: BPM. Volume 4714 of Lecture Notes in Computer Science., Springer (2007) 375–383
22. Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K.: CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In van der Aalst, W.M.P., Best, E., eds.: ICATPN. Volume 2679 of Lecture Notes in Computer Science., Springer (2003) 450–462
23. Westergaard, M.: CPN Tools 4: Multi-formalism and Extensibility. In Colom, J.M., Desel, J., eds.: Petri Nets. Volume 7927 of Lecture Notes in Computer Science., Springer (2013) 400–409

On-The-Fly Model Checking of Timed Properties on Time Petri Nets

Kais Klai

LIPN, CNRS UMR 7030
Université Paris 13, Sorbonne Paris Cité
99 avenue Jean-Baptiste Clément
F-93430 Villetaneuse, France
kais.klai@lipn.univ-paris13.fr

Abstract. This paper deals with model checking of timed systems modeled by Time Petri nets (TPN). We propose a new finite graph, called Timed Aggregate Graph (TAG), abstracting the behavior of bounded TPNs with strong time semantics. The main feature of this abstract representation compared to existing approaches is the encoding of the time information. This is done in a pure way within each node of the TAG allowing to compute the minimum and maximum elapsed time in every path of the graph. The TAG preserves runs and reachable states of the corresponding TPN and allows for on-the-fly verification of reachability properties. We illustrate in this paper how the TAG can be used to check some usual timed reachability properties and we supply an algorithm for extracting an explicit timed trace (involving the elapsed time before each fired transition) from an abstract run of the TAG. The TAG-based approach is implemented and compared to two well known TPNs analysis approaches.

1 Introduction

Time Petri nets are one of the most used formal models for the specification and the verification of systems involving explicit timing constraints, such as communication protocols, circuits, or real-time systems. The main extensions of Petri nets with time are *time Petri nets* [18] and *timed Petri nets* [22]. In the former, a transition can fire within a time interval whereas, in the latter, time durations can be assigned to the transitions; tokens are meant to spend that time as reserved in the input places of the corresponding transitions. Several variants of timed Petri nets exist: time is either associated with places (p-timed Petri nets), with transitions (t-timed Petri nets) or with arcs (a-timed Petri nets) [23]. The same holds for time Petri nets [7]. In [21], the authors prove that p-timed Petri nets and t-timed Petri nets have the same expressive power and are less expressive than time Petri nets. Several semantics have been proposed for each variant of these models. Here we focus on t-time Petri nets, which we simply call TPNs. There are two ways of letting the time elapse in a TPN [21]. The first way, known as the *Strong Time Semantics* (STS), is defined in such a

manner that time elapsing cannot disable a transition. Hence, when the upper bound of a firing interval is reached, the transition must be fired. The other semantics, called *Weak Time Semantics* (WTS), does not make any restriction on the elapsing of time.

For real-time systems, dense time model (where time is considered in the domain $\mathbb{R}_{\geq 0}$) is the unique possible option, raising the problem of handling an infinite number of states. In fact, the set of reachable states of the TPN is generally infinite due to the infinite number of time successors a given state could have. Two main approaches are used to treat this state space: region graphs [1] and the state class approach [3]. The other methods [2,24,4,10,5,17,6,11] are either refinements, improvements or derived from these basic approaches. The objective of these representations is to yield a state-space partition that groups concrete states into sets of states presenting similar behavior with respect to the properties to be verified. These sets of states must cover the entire state space and must be finite in order to ensure the termination of the verification process.

In this work, we propose a new finite graph, called Timed Aggregate Graph (TAG), abstracting the behavior of bounded TPNs with strong time semantics. A preliminary version of this work has been published in [13,14], where a coarser abstraction of TPNs' state graph is proposed. The key idea behind the approach presented in this paper is the fact that the time information associated with each node is related to the current path leading to this node. In particular, given a node of the TAG, for each couple of enabled transitions $\langle t, t' \rangle$, the value of the earliest and latest firing times of t (reps. t') the last time, in the current path, it "met" t' (resp. t) is stored in the node. This information , represented by a matrix, allows us (1) to maintain the relative differences between the firing times of enabled transitions (diagonal constraints), (2) to determine the fireable transitions at each node, and (3) to compute dynamically the earliest and the latest firing time of each enabled transition for each node of the TAG. This new version of the TAG allows to preserve the timed traces of the underlying TPN while the abstraction proposed in [13,14] is an upper approximation of the set of traces of the underlying TPN. Moreover, one can compute the minimum and maximum elapsed time through every path of the graph which permits on-the-fly verification of timed reachability properties (e.g., is some state reachable between d and D time units).

This paper is organized as follows: In Section 2, some preliminaries about TPNs and the corresponding semantics are recalled. In Section 3, we define the Timed Aggregate Graph (TAG) associated with a TPN and we discuss the main preservation results of the TAG-based approach. In Section 4, we show how the verification of some usual reachability properties can be accomplished on-the-fly by exploring the TAG. Section 5 relates our work to existing approaches. In Section 6, we discuss the experimental results obtained with our implementation compared to two well-known tools, namely Romeo [9] and TINA [5]. Finally, a conclusion and some perspectives are given in Section 7.

2 Preliminaries and Basic Notations

A TPN is a P/T Petri net [20] where a time interval $[t_{\min}; t_{\max}]$ is associated with each transition t.

Definition 1. *A* TPN *is a tuple* $\mathcal{N} = \langle P, T, Pre, Post, I\rangle$ *where:*

- $\langle P, T, Pre, Post\rangle$ *is a* P/T Petri net
- $I : T \longrightarrow \mathbb{N} \times (\mathbb{N} \cup \{+\infty\})$ *is the* time interval function *such that:* $I(t) = (t_{\min}, t_{\max})$*, with* $t_{\min} \leq t_{\max}$*, where* $t_{\min}$ *(resp.* $t_{\max}$*) is the earliest (resp. latest) firing time of transition* t*.*

A *marking* of a TPN is a function $m : P \longrightarrow \mathbb{N}$ where $m(p)$, for a place p, denotes the number of tokens in p. A *marked TPN* is a pair $\mathcal{N} = \langle \mathcal{N}_1, m_0\rangle$ where $\mathcal{N}_1$ is a TPN and m_0 is a corresponding *initial marking*. A transition t is enabled by a marking m iff $m \geq Pre(t)$ and $Enable(m) = \{t \in T : m \geq Pre(t)\}$ denotes the set of enabled transitions in m. If a transition t_i is enabled by a marking m, then $\uparrow(m, t_i)$ denotes the set of newly enabled transitions [2]. Formally, $\uparrow(m, t_i) = \{t \in T \mid t \in Enable(m - Pre(t_i) + Post(t_i)) \wedge (t \notin Enable(m - Pre(t_i)) \vee (t = t_i))\}$. If a transition t is in $\uparrow(m, t_i)$, we say that t is newly enabled by the successor of m by firing t_i. Dually, $\downarrow(m, t_i) = Enable(m - Pre(t_i) + Post(t_i)) \setminus \uparrow(m, t_i)$ is the set of oldly enabled transitions. The possibly infinite set of reachable markings of $\mathcal{N}$ is denoted $Reach(\mathcal{N})$. If the set $Reach(\mathcal{N})$ is finite we say that $\mathcal{N}$ is bounded.

The semantics of TPNs can be given in terms of Timed Transition Systems (TTS) [15] which are usual transition systems with two types of labels: discrete labels for events (transitions) and positive real labels for time elapsing (delay). States (configurations) of the TTS are pairs $s = (m, V)$ where m is a marking and $V : T \longrightarrow \mathbb{R}_{\geq 0} \cup \{\bot\}$ a time valuation. In the following, $s.m$ and $s.V$ denote the marking and the time valuation respectively of a state s. If a transition t is enabled in m then $V(t)$ is the elapsed time since t became enabled, otherwise $V(t) = \bot$. Given a state $s = (m, V)$ and a transition t, t is said to be fireable in s iff $t \in Enable(m) \wedge V(t) \neq \bot \wedge t_{min} \leq V(t) \leq t_{\max}$.

Definition 2 (Semantics of a TPN). *Let* $\mathcal{N} = \langle P, T, Pre, Post, I, m_0\rangle$ *be a marked TPN. The semantics of* $\mathcal{N}$ *is a TTS* $\mathcal{S}_{\mathcal{N}} = \langle Q, s_0, \rightarrow\rangle$ *where:*

1. Q *is a (possibly infinite) set of states*
2. $s_0 = (m_0, V_0)$ *is the initial state such that:*

$$\forall t \in T,\ V_0(t) = \begin{cases} 0 & \text{if } t \in Enable(m_0) \\ \bot & \text{otherwise} \end{cases}$$

3. $\rightarrow \subseteq Q \times (T \cup \mathbb{R}_{\geq 0}) \times Q$ *is the discrete and continuous transition relations:*
   (a) *the discrete transition relation:*
   $\forall t \in T :\ (m, V) \xrightarrow{t} (m', V')$ *iff:*

$$\begin{cases} t \in Enable(m) \wedge m' = m - Pre(t) + Post(t) \\ t_{min} \leq V(t) \leq t_{max} \\ \forall t' \in T : V'(t') = \begin{cases} 0 & \text{if } t' \in \uparrow(m,t) \\ V(t') & \text{if } t' \in \downarrow(m,t) \\ \bot & \text{otherwise} \end{cases} \end{cases}$$

(b) the continuous transition relation: $\forall d \in \mathbb{R}_{\geq 0}$, $(m,V) \xrightarrow{d} (m',V')$ *iff:*

$$\begin{cases} \forall t \in Enable(m),\ V(t) + d \leq t_{max} \\ m' = m \\ \forall t \in T : \\ V'(t) = \begin{cases} V(t) + d & \text{if } t \in Enable(m); \\ V(t) & \text{otherwise.} \end{cases} \end{cases}$$

The above definition requires some comments. First, a state change occurs either by the firing of transitions or by time elapsing: The firing of a transition may change the current marking while the time elapsing may make some new transitions fireable. Second, the delay transitions respect the STS semantics: an enabled transition must fire within its firing interval unless it is disabled by the firing an other transition.

Given a TPN $\mathcal{N}$ and the corresponding TTS $\mathcal{S}_{\mathcal{N}}$, a path $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$, where $\alpha_i \in (T \cup \mathbb{R}_{\geq 0})$, is a run of $\mathcal{S}_{\mathcal{N}}$ iff $(s_i, \alpha_i, s_{i+1}) \in \rightarrow$ for each $i = 0, 1, \ldots$. The length of a run π can be infinite and is denoted by $\mid \pi \mid$. The possibly infinite set of runs of $\mathcal{S}_{\mathcal{N}}$ is denoted $[\mathcal{S}_{\mathcal{N}}]$. Without loss of generality, we assume that for each non empty run $\pi = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots$ of a STS corresponding to a TPN, there do not exist two successive labels α_i and α_{i+1} belonging both to $\mathbb{R}_{\geq 0}$. Then, π can be written, involving the reachable markings of $\mathcal{N}$, as $\pi = m_0 \xrightarrow{(d_1,t_1)} m_1 \xrightarrow{(d_2,t_2)} \ldots$ where d_i is the time elapsed at marking m_{i-1} before firing t_i. In order to associate a run π of $\mathcal{S}_{\mathcal{N}}$ with a run of $\mathcal{N}$, denoted $\mathcal{P}(\pi)$, we define the following projection function, where $.$ denotes the concatenation operator between paths and π^i, for $i = 0, 1 \ldots$, denotes the suffix of π starting at state s_i.

$$\mathcal{P}(\pi) = \begin{cases} s_0.m & \text{if } \mid \pi \mid = 0 \\ s_0.m \xrightarrow{(0,\alpha_1)} . \mathcal{P}(\pi^1) & \text{if } \alpha_1 \in T \\ s_0.m \xrightarrow{(\alpha_1,\alpha_2)} . \mathcal{P}(\pi^2) & \text{if } \alpha_1 \in \mathbb{R}_{\geq 0} \wedge \mid \pi \mid \geq 2 \\ s_0.m \xrightarrow{\alpha_1} . \mathcal{P}(\pi^1) & \text{if } \alpha_1 \in \mathbb{R}_{\geq 0} \wedge \mid \pi \mid = 1 \end{cases}$$

3 Abstraction of a TPN State Space

3.1 Timed Aggregate Graph

In this subsection, we propose to abstract the reachability state space of a TPN using a new graph called Timed Aggregate Graph (TAG) where nodes are called

aggregates and are grouping sets of states of a TTS. The key idea behind TAGs is the way the time information is encoded inside aggregates. In addition to the marking characterizing an aggregate, the time information is composed of two parts:

- The first part of the time information characterizing an aggregate is a dynamically updated interval, namely (α_t, β_t), associated with each enabled transition t. This interval gives the earliest and the latest firing times of any enabled transition starting from the corresponding aggregate. Either the corresponding transition is fireable at the current aggregate and the system must remain within the aggregate at least α_t time units and at most β_t time units (as long as the other enabled transitions remain fireable) before firing t, or t is not possible from the current aggregate (e.g. because of some diagonal constraint), and the system must move to an other aggregate by firing other transitions until t becomes fireable. In the latter case, the system must consume at least α_t, and can consume at most β_t to make t fireable in the future.
- The second part of the time information characterizing an aggregate is a matrix, namely $Meet$, allowing to dynamically maintain the relative differences between the firing times of enabled transitions (diagonal constraints). Given two enabled transitions t_1 and t_2, $Meet(t_1, t_2)$ is an interval representing the earliest and the latest firing times of t_1 the last time both t_1 and t_2 were enabled (through the paths leading to the aggregate).

Before we formally define the TAG and illustrate how the attributes of an aggregate are computed dynamically, let us first formally define aggregates.

Definition 3 (Timed Aggregate). *Let $\mathcal{N} = \langle P, T, Pre, Post, I\rangle$ be a* TPN. *A* timed aggregate *associated with $\mathcal{N}$ is a tuple $a = (m, E, Meet)$, where:*

- *m is a marking*
- *$E = \{\langle t, \alpha_t, \beta_t\rangle \mid t \in Enable(m),\ \alpha_t \in \mathbb{N} \wedge \beta_t \in \mathbb{N} \cup \{+\infty\}\}$ is a set of enabled transitions each asssociated with two time values.*
- *Meet is a matrix s.t. $\forall t, t' \in Enable(m)$, $Meet(t, t') = \langle \alpha, \beta\rangle$ where α (resp. β) represents the earliest (resp. latest) firing time of t the last time t and t' are both enabled before reaching the aggregate a.*

As for the states of a TTS, the attributes of an aggregate a are denoted by $a.m$, $a.E$ and $a.Meet$. Moreover, $a.Meet(t, t').\alpha$ (resp. $a.Meet(t, t').\beta$) is denoted by $a.\alpha_t^{m(t,t')}$ (resp. $a.\beta_t^{m(t,t')}$), or simply $\alpha_t^{m(t,t')}$ (resp. $\beta_t^{m(t,t')}$) when the corresponding aggregate is clear from the context. We use also $\alpha^{m(t,t')}$ (resp. $\beta^{m(t,t')}$) to denote $\alpha_t^{m(t,t')}$ (resp. $\beta_t^{m(t,t')}$) when the involved transition t is clear from the context.

The E attribute of an aggregate a allows to compute the minimum and the maximum time the system can elapse when its current state is within a. The following predicates (δ and Δ) compute these information for a given aggregate.

Definition 4 (Minimum and maximum stay times). *Let $a = \langle m, E\rangle$ be an aggregate, the minimum and maximum time the system can stay at a are denoted by $\delta(a)$ and $\Delta(a)$ respectively, and are defined by the two following predicates:*

- $\delta(a) = \min_{\langle t,\alpha_t,\beta_t\rangle \in E}(\alpha_t)$
- $\Delta(a) = \max_{\langle t,\alpha_t,\beta_t\rangle \in E}(\beta_t)$

The minimum (resp. maximum) stay time $\delta(a)$ (res. $\Delta(a)$) of an aggregate a allows to encapsulate the continuous transition relation within a.

Given an aggregate $a = \langle m, E\rangle$ and an enabled transition t (i.e., $\langle t, \alpha_t, \beta_t\rangle \in E$), two primordial issues must be achieved to define the semantics of the TAG: (1) is t fireable from a?, and (2) if it is the case, how do we obtain the successor aggregate by firing t from a. In the following, we answer these issues.

Definition 5. *Let $a = \langle m, E, Meet\rangle$ be an aggregate and let $\langle t, \alpha_t, \beta_t\rangle \in E$. Then, t is fireable at a, denoted by $a \overset{t}{\rightsquigarrow}$, iff $\forall \langle t', \alpha_{t'}, \beta_{t'}\rangle \in E,\ \alpha_t^{m(t,t')} \leq \beta_{t'}^{m(t',t)}$*

A transition t is fireable at an aggregate a iff there is no transition t', that is enabled by a, whose latest firing time was strictly smaller than the earliest firing time of t the last time both transitions were enabled.

Now that the firability condition is formally defined, the following definition computes the successor aggregate obtained by the firing of a given transition. In this definition, the notion of newly (and oldly) enabled transitions is extended to aggregates as follows: $\uparrow(a,t) = \uparrow(a.m,t)$ and $\downarrow(a,t) = \downarrow(a.m,t)$ for each transition t enabled by $a.m$

Definition 6. *Let $a = \langle m, E, Meet\rangle$ be an aggregate and let $\langle t, \alpha_t, \beta_t\rangle \in E$.*

Assume that t is fireable at a (following Definition 5). The aggregate $a' = \langle m', E', Meet'\rangle$ obtained by firing t from a, denoted by $a \overset{t}{\rightsquigarrow} a'$, is obtained as follows:

1. $m' = m - Pre(t) + Post(t)$
2. $E' = E'_1 \cup E'_2$, *where:*
   - $E'_1 = \bigcup_{t' \in \uparrow(a,t)} \{\langle t', t'_{\min}, t'_{\max}\rangle\}$
   - $E'_2 = \bigcup_{t' \in \downarrow(a,t)} \{\langle t', \alpha'_{t'}, \beta'_{t'}\rangle\}$ *where:*
     - $\alpha'_{t'} = \alpha_{t'} - SCR(a,t')$, *where* $SCR(a,t') =$ $Max(0, (Min_{t'' \in Enable(a)}(Min(\beta^{m(t',t'')}, \beta^{m(t'',t')}) - (\alpha_{t'}^{m(t',t'')} - \alpha_{t'}))$
     - $\beta'_{t'} = \beta_{t'} - Max(0, (\alpha_t^{m(t,t')} - (\beta_{t'}^{m(t',t)} - \beta_{t'}))$
   - $\forall(\langle t_1, \alpha_1, \beta_1\rangle,\ \langle t_2, \alpha_2, \beta_2\rangle) \in E' \times E'$

$$Meet'(t_1,t_2) = \begin{cases} [t_{1_{\min}}, t_{1_{\max}}] & \text{if } t_1 \in \uparrow(a,t) \\ [\alpha_1, \beta_1] & \text{if } t_1 \in \downarrow(a,t) \wedge t_2 \in \uparrow(a,t) \\ Meet(t_1,t_2) & \text{if } t_1 \in \downarrow(a,t) \wedge t_2 \in \downarrow(a,t). \end{cases}$$

The computation of a successor a' of an aggregate a by the firing of a transition t is guided by the following intuition: If $\downarrow(a,t) \neq \emptyset$, then *the more the system can remain at a, the less it can remain at a' and vice versa*. Otherwise, the time elapsed within a' is independent from the time elapsed within a. Thus, given a

transition t' enabled by a', two cases are considered: if t' is newly enabled, then its earliest and latest firing times are statically obtained by $t'_{\min}$ and $t'_{\max}$ respectively. Otherwise, the more one can remain at a, the less will be the necessary wait time at a' before firing t'. The function SCR (Still Can Remain) allows to compute the maximum remaining time at a under the hypothesis that, since t' became enabled, it remains the maximum time at each encountered aggregate before reaching a (note that this is different from $\Delta(a)$). Thus $SCR(a, t')$ is obtained by the following reasoning: given a transition t'' that is enabled by a, it is clear that since the last time t' and t'' became both enabled, the maximum elapsed time can not be greater than $Min(\beta^{m(t',t'')}, \beta^{m(t'',t')})$ (because of the STS semantics which is used in this paper). The maximum time the system can remain at a is then obtained by subtracting from this quantity the time that is already spent during the path leading to a (i.e., $(\alpha_{t'}^{m(t',t'')} - \alpha_{t'})$). By analyzing all the transitions enabled by a the function SCR takes the minimum values in order to not violate the STS semantics rule. Similarly, the latest firing time of t' corresponds to the situation where, between the last time t and t' were both enabled and the current aggregate a, each fired transition is fired as soon as possible. Each time a transition is fired, its earliest firing time is subtracted from the latest firing time of the old transitions. However, if the quantity of time that must be subtracted from the latest firing time of t' has already been subtracted in between, then the latest firing time of t' at a' is the same latest firing time of t' at the aggregate a.

Concerning the $Meet$ attribute, given two transitions t_1 and t_2 that are enabled at a', the value of $Meet(t_1, t_2)$ is simply obtained by considering the membership of these transitions to $\uparrow(a, t)$ and to $\downarrow(a, t)$. Finally, by considering that $\infty - \infty = 0$, the previous definition allows to handle transitions having an unbounded latest firing time.

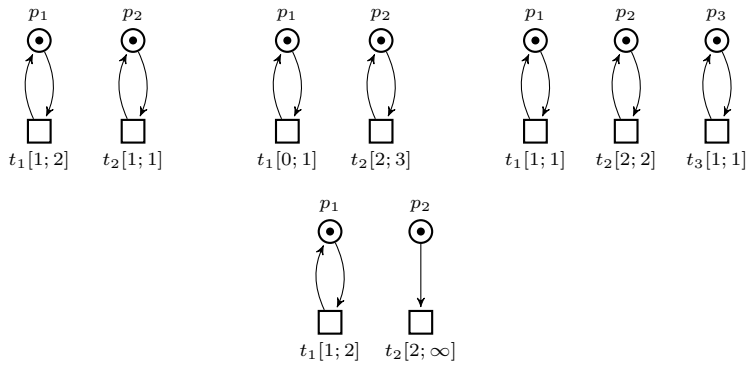


Fig. 1. Four TPN Examples

Now, we are ready to formally define the TAG associated with a marked TPN $\mathcal{N}$. It is a labeled transition system where nodes are timed aggregates. It has an initial aggregate, a set of actions (the set of transitions of $\mathcal{N}$) and a transition relation. The initial aggregate is easily computed by considering **static** information of the TPN while the transition relation is directly obtained by Definition 5 and Definition 6.

Definition 7 (Timed Aggregate Graph). *Let $\mathcal{N} = \langle P, T, Pre, Post, I, m_0\rangle$ be a TPN. The* TAG *associated with $\mathcal{N}$ is a tuple $G = \langle \mathcal{A}, T, a_0, \delta\rangle$ where:*

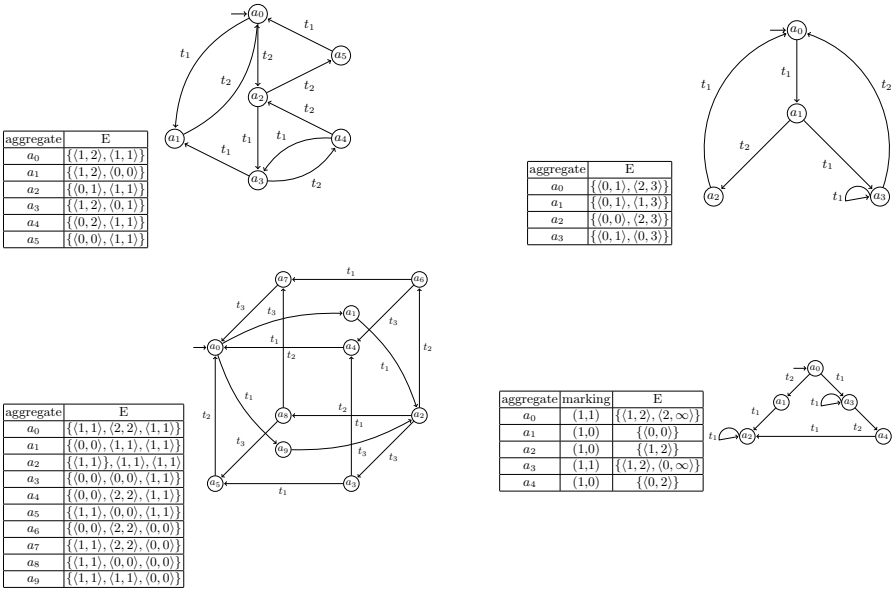
1. *$\mathcal{A}$ is a set of timed aggregates;*
2. *$a_0 = \langle m_0, h_0\rangle$ is the initial timed aggregate s.t.:*
   (a) m_0 is the initial marking of $\mathcal{N}$.
   (b) $E_0 = \{\langle t, t_{\min}, t_{\max}\rangle \mid t \in Enable(m_0)\}$
   (c) $\forall t, t \in Enable(a),\ Meet(t, t') = [t_{\min}, t_{\max}]$
3. *$\delta \subseteq \mathcal{A} \times T \times \mathcal{A}$ is the transition relation such that:*
   $\forall a \in A,\ \forall t \in T,\ (a, t, a') \in \delta$ iff $a \overset{t}{\rightsquigarrow} a'$

Since each transition having an unbounded static latest firing time will always maintain the same latest firing time at each aggregate where it is enabled, one can prove that the number of aggregates of a TAG is bounded when the corresponding TPN is bounded. Indeed, given a reachable marking m, the number of different aggregates having m as marking can be bounded by the number of possible values of its attributes. This number is finite because of the following facts: (1) if the number of the transitions that are enabled by m is e, there are $2^{|e|}$ possible subsets of old transitions; (2) for a given subset of old transitions o, the number of possible arrangements of the old transitions regarding the enabling time is at most equal to $\mid o \mid!$ (the 2^n elements corresponding to the orderings where two or more old transitions became enabled at the same time are not considered); (3) given an arrangement $t_1 \leq t_2 \leq \cdots \leq t_{|o|}$, the number of possible values of $\alpha_{t_1}^{m(t_1,t_2)}$ is at most equel to $\sum_{i=0}^{t_{1_{\min}}} (t_{1_{\max}} - i + 1)$. Similarly, the possible values of $\alpha_{t_2}^{m(t_2,t_3)}$ is equal to $\sum_{i=0}^{t_{2_{\min}}} (t_{2_{\max}} - i + 1)$, etc. Thus, the number of the possible different values of the matrix $Meet$, for this particular arrangement, is obtained by $\Pi_{j=2}^{|o|} \sum_{i=0}^{t_{j-1_{\min}}} (t_{j-1_{\max}} - i + 1)$; (4) for each enabled transition t (with $t_{max} \neq \infty$), there are at most $\sum_{i=0}^{t_{\min}} (t_{\max} - i + 1)$ different intervals that can represent the earliest and latest firing times associated with t in a given aggregate (i.e., α_t and β_t). When $t_{max} = \infty$, the number of possible time intervals associated with t is $t_{min} + 1$.

Figure 2 illustrates the TAGs corresponding to the TPNs of Figure 1. In the three first TAGs, the marking associated with each aggregate is omitted (it is the same as the initial one). The second column of the tables gives the dynamic earliest and latest firing times of the enabled transitions (i.e., t_1, t_2 and t_3 respectively). For sake of readability of the figures, the $Meet$ attribute is omitted.

Although the four models of Figure 1 are quite simple, they are representative enough to explain the TAG construction. Indeed, in the first one the transitions

intervals overlap, while the case of disjoint intervals is considered through the second and the third models. Finally, the fourth model illustrates the case of an unbounded latest firing time. More significant examples are considered in Section 6.



aggregate	E
a_0	$\{\langle 1,2\rangle, \langle 1,1\rangle\}$
a_1	$\{\langle 1,2\rangle, \langle 0,0\rangle\}$
a_2	$\{\langle 0,1\rangle, \langle 1,1\rangle\}$
a_3	$\{\langle 1,2\rangle, \langle 0,1\rangle\}$
a_4	$\{\langle 0,2\rangle, \langle 1,1\rangle\}$
a_5	$\{\langle 0,0\rangle, \langle 1,1\rangle\}$

aggregate	E
a_0	$\{\langle 0,1\rangle, \langle 2,3\rangle\}$
a_1	$\{\langle 0,1\rangle, \langle 1,3\rangle\}$
a_2	$\{\langle 0,0\rangle, \langle 2,3\rangle\}$
a_3	$\{\langle 0,1\rangle, \langle 0,3\rangle\}$

aggregate	E
a_0	$\{\langle 1,1\rangle, \langle 2,2\rangle, \langle 1,1\rangle\}$
a_1	$\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 1,1\rangle\}$
a_2	$\{\langle 1,1\rangle\}, \langle 1,1\rangle, \langle 1,1\rangle$
a_3	$\{\langle 0,0\rangle, \langle 0,0\rangle, \langle 1,1\rangle\}$
a_4	$\{\langle 0,0\rangle, \langle 2,2\rangle, \langle 1,1\rangle\}$
a_5	$\{\langle 1,1\rangle, \langle 0,0\rangle, \langle 1,1\rangle\}$
a_6	$\{\langle 0,0\rangle, \langle 2,2\rangle, \langle 0,0\rangle\}$
a_7	$\{\langle 1,1\rangle, \langle 2,2\rangle, \langle 0,0\rangle\}$
a_8	$\{\langle 1,1\rangle, \langle 0,0\rangle, \langle 0,0\rangle\}$
a_9	$\{\langle 1,1\rangle, \langle 1,1\rangle, \langle 0,0\rangle\}$

aggregate	marking	E
a_0	(1,1)	$\{\langle 1,2\rangle, \langle 2,\infty\rangle\}$
a_1	(1,0)	$\{\langle 0,0\rangle\}$
a_2	(1,0)	$\{\langle 1,2\rangle\}$
a_3	(1,1)	$\{\langle 1,2\rangle, \langle 0,\infty\rangle\}$
a_4	(1,0)	$\{\langle 0,2\rangle\}$

Fig. 2. TAGs associated with TPNs of Figure 1

3.2 Preservation Results

In this section, we establish the main result of our approach: The TAG is an exact representation of the reachability state space of a TPN. In fact, for each path in the TPN (resp. in the corresponding TAG) it is possible to find a path in the TAG (resp. TPN) involving the same sequence of transitions and where the time elapsed within a given state is between the minimum and the maximum stay time of the corresponding aggregate.

Theorem 1. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta\rangle$ be the TAG associated with $\mathcal{N}$. Then $\forall \overline{\pi} = m_0 \overset{(d_1,t_1)}{\longrightarrow} m_1 \overset{(d_2,t_2)}{\longrightarrow} \dots \overset{(d_n,t_n)}{\longrightarrow} m_n \overset{d_{n+1}}{\longrightarrow}$, with $d_i \in \mathbb{R}_{\geq 0}$, for $i = 1 \dots n+1$, $\exists \pi = a_0 \overset{t_1}{\longrightarrow} a_1 \longrightarrow \dots \overset{t_n}{\longrightarrow} a_n$ s.t. $\forall i = 0 \dots n$, $d_{i+1} \leq \Delta(a_i)$, $m_i = a_i.m$ and $\forall i = 1 \dots n$, $d_i \geq \alpha_{i-1_{t_i}}$.*

Proof. Let $\overline{\pi} = m_0 \xrightarrow{(d_1,t_1)} m_1 \xrightarrow{(d_2,t_2)} \dots \xrightarrow{(d_n,t_n)} m_n \xrightarrow{d_{n+1}}$ be a path of $\mathcal{N}$, with $d_i \in \mathbb{R}_{\geq 0}$, for $i = 1 \dots n+1$. Given a path $a_0 \longrightarrow a_1 \dots$, we denote by α_{i_t} (res. β_{i_t}), for $i = 0 \dots$, the dynamic earliest firing time (resp. latest firing time) of a transition t enabled by an aggregate a_i.

Let us prove by induction on the length of $\overline{\pi}$ the existence of a path π in the TAG satisfying the conditions of Theorem 1.

- $\mid \overline{\pi} \mid = 0$: Obvious since $m_0 = a_0.m$ (by construction) and since d_1 is less or equal to $\min_{t \in Enable(m_0)} t_{max}$ which is exactly the value of $\Delta(a_0)$.
- $\mid \overline{\pi} \mid = 1$ i.e., $\overline{\pi} = m_0 \xrightarrow{(d_1,t_1)} m_1 \xrightarrow{d_2}$. It is clear that $\alpha_{0_{t_1}} \leq d_1 \leq \Delta(a_0)$. The fact that t_1 is fireable at m_0 implies that it is at a_0 ($\forall t \in Enable(m_0),\ t_{1_{min}} \leq t_{max}$) and its firing leads to the aggregate a_1 satisfying $a_1.m = m_1$. Let us assume that $d_2 > \Delta(a_1)$ and let t_m be the transition that is enabled at a_1 and which has the smallest latest firing time i.e., $\beta_{1_{t_m}} = \Delta(a_1)$. If t_m is newly enabled at a_1 then d_2 should clearly be greater or equal to $\Delta(a_1)$. If $t_m \in\downarrow (a_0, t_1)$ then $\beta_{1_{t_m}} = t_{m_{max}} - t_{1_{min}}$. Since $d_1 \geq t_{1_{min}}$, then $t_{m_{max}} - t_{1_{min}} \geq t_{m_{max}} - d_1$. The fact that $d_2 > \beta_{1_{t_m}}$ would imply that $d_1 + d_2 > t_{m_{max}}$ which is contradictory with the STS semantics. Thus $d_2 \leq \Delta(a_1)$.
- Assume that for any path $\overline{\pi}$ s.t. $\mid \overline{\pi} \mid \leq n$, there exists a path in the TAG with the same trace and satisfying the above conditions. Let $\overline{\pi} = m_0 \xrightarrow{(d_1,t_1)} m_1 \xrightarrow{(d_2,t_2)} \dots \xrightarrow{(d_n,t_n)} m_n \xrightarrow{(d_{n+1},t_{n+1})} m_{n+1} \xrightarrow{d_{n+2}}$ be a path of length $n+1$. Let $\pi = a_0 \xrightarrow{t_1} a_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} a_n$ be the path in the TAG associated with the n-length prefix of $\overline{\pi}$ (by the induction hypothesis). Then $d_{n+1} \leq \Delta(a_n)$. Let us demonstrate that $d_{n+1} \geq \alpha_{n_{n+1}}$: It is clear that this is the case when $t_{n+1} \in\uparrow (a_n, t_{n+1}$. If $t_{n+1} \in\downarrow (a_n, t_{n+1}$, let $LastNew_i(t)$ be the function that returns the greatest integer, smaller than (or equal to) i, such that $t \in\uparrow (a_{l-1}, t_l)$. If such a value does not exist, then t became enabled, for the last time, at the initial aggregate a_0 and the function returns 0. Let $k = LastNew_i(t_{n+1})$, then $\alpha_{n_{n+1}} = t_{n+1_{min}} - \sum_{i=k}^{n-1} SCR(a_i, t_{n+1})$. The STS semantics implies that $\sum_{i=k}^{n-1} SCR(a_i, t_{n+1}) \geq \sum_{i=k}^{n-1} d_{i+1}$ Thus $t_{n+1_{min}} - \sum_{i=k}^{n-1} SCR(a_i, t_{n+1}) \geq t_{n+1_{min}} - \sum_{i=k}^{n-1} d_{i+1}$, and $d_{n+1} > \alpha_{n_{t_{n+1}}}$ would means that $\sum_{i=k}^{n} d_{i+1} < t_{n+1_{min}}$ which would prevent the firing of t_{n+1} at m_n. Thus, $d_{n+1} \leq \alpha_{n_{t_{n+1}}}$. Let us show now that t_{n+1} is fireable at a_n. Assume the opposite, this would imply that there exists a transition t enabled by a_n such that $\alpha^{m(t_{n+1},t)} > \beta^{m(t,t_{n+1})}$. Let $LastNew_n(t_{n+1}) = l$, $LastNew_n(t) = k$, and let us consider the three following cases:
  1. $l = k$, then $\beta^{m(t,t_{n+1})} = t_{max}$ and $\alpha^{m(t_{n+1},t)} = t_{n_{min}}$ and the fact that $t_{n_{min}} > t_{max}$ would prevent t_{n+1} from being fireable at m_n which is not the case. Thus, t_{n+1} is fireable at a_n as well.
  2. $l < k$. In this case, $\alpha^{m(t_{n+1},t)} = t_{n+1_{min}} - \sum_{j=l}^{k-1} SCR(a_j, t_{n+1})$ and $\beta^{m(t,t_{n+1})} = t_{max}$. Again, the STS semantics implies that $\sum_{i=l}^{k-1} SCR(a_i, t_{n+1}) \geq \sum_{i=l}^{k-1} d_{i+1}$. Thus, $t_{n+1_{min}} - \sum_{i=l}^{k-1} SCR(a_i, t_{n+1}) \leq t_{n+1_{min}} - \sum_{i=l}^{k-1} d_{i+1}$, and $\alpha^{m(t_{n+1},t)} > t_{max}$ would means that $\sum_{i=l}^{k} d_{i+1} < t_{n+1_{min}}$

which would prevent the firing of t_{n+1} at m_n. Thus, $\alpha^{m(t_{n+1},t)} \leq \beta^{m(t,t_{n+1})}$ and t_{n+1} is necessarily fireable at a_n.

3. $l > k$. In this case, $\alpha^{m(t_{n+1},t)} = t_{n+1_{min}}$ and $\beta^{m(t,t_{n+1})} = t_{max} - \sum_{i=k}^{l-1} Max(0, (\alpha^{m(t_{i+1},t)} - (\beta^{m(t,t_{i+1})} - \beta_{i_t}))$. Knowing that $\sum_{i=k}^{l-1} Max(0, (\alpha^{m(t_{i+1},t)} - (\beta^{m(t,t_{i+1})} - \beta_{i_t})) \leq \sum_{i=k}^{l-1} d_{i+1}$ (otherwise, the time spent between k and some $i \leq l$ is smaller than $\alpha^{m(t_i,t)}$, which is contradictory with the recurrence hypothesis), $t_{max} - \sum_{i=k}^{l-1} Max(0, (\alpha^{m(t_{i+1},t)} - (\beta^{m(t,t_{i+1})} - \beta_{i_t})) \geq t_{max} - \sum_{i=k}^{l-1} d_{i+1}$. Thus, if $t_{n+1_{min}} > \beta^{m(t,t_{n+1})}$ then $t_{n+1_{min}} > t_{max} - \sum_{i=k}^{l-1} d_{i+1}$ which prevent the firing of t_{n+1} at m_n (before firing t) which is not true. Thus, t_{n+1} is fireable at a_n.
Let us now demonstrate that $d_{n+2} \leq \Delta(a_{n+1})$. Assume the opposite, and let t_m be the transition enabled by a_{n+1} which has the smallest latest firing time i.e. $\beta_{n+1_{t_m}} = \Delta(a_{n+1})$. It is clear that if $t_m \in\uparrow (a_n, t_{n+1})$ then $d_{n+2} \leq \Delta(a_{n+1})$. Otherwise, if $t_m \in\downarrow (a_n, t_{n+1})$ and $k = LastNew_{n+1}(t_m)$ then $\beta_{n+1_{t_m}} = t_{m_{max}} - \sum_{i=k}^{n} Max(0, (\alpha^{m(t_{i+1},t_m)}$ Again, since $\sum_{i=k}^{n} d_{i+1} \geq \sum_{i=k}^{n} Max(0, (\alpha^{m(t_{i+1},t_m)}$, then $\beta_{n+1_{t_m}} \leq t_{m_{max}} - \sum_{i=k}^{n} d_{i+1}$, and the fact that $d_{n+1} > \beta_{n+1_{t_m}}$ would imply that $d_{n+2} + \sum_{i=k}^{n} d_{i+1} > t_{m_{max}}$ which is not allowed by the STS semantics. Thus, $d_{n+2} \leq \Delta(a_{n+1})$.

Theorem 2. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta \rangle$ be the TAG associated with $\mathcal{N}$. Then, for any path $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$ in the TAG, there exists a run $\overline{\pi} = m_0 \overset{(d_1,t_1)}{\longrightarrow} m_1 \longrightarrow \ldots \overset{(d_n,t_n)}{\longrightarrow} m_n$ in $\mathcal{N}$, s.t. $\forall i = 0 \ldots n$, $m_i = a_i.m$, $\forall i = 1 \ldots n$, $\alpha_{i-1_{t_i}}) \leq d_i \leq \Delta(a_{i-1})$, and $\forall d \in \mathbb{R}_{\geq 0}$, $m_n \xrightarrow{d} \Leftrightarrow d \leq \Delta(a_n)$*

Proof. Let $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$. We denote by α_{i_t} (res. β_{i_t}) the dynamic earliest firing time (resp. the dynamic latest firing time) of a transition t at aggregate a_i, for $i \in \{0, \ldots, n-1\}$. Let us demonstrate that the path $\overline{\pi} = m_0 \overset{(d_1,t_1)}{\longrightarrow} m_1 \longrightarrow \ldots \overset{(d_n,t_n)}{\longrightarrow} m_n$ obtained by the following algorithm satisfies the requirement. The function $LastNew_i(t)$ returns the greatest integer l, smaller than i, such that $t \in\uparrow (a_{l-1}, t_l)$. If such a value does not exist, then t became enabled, for the last time, at the initial aggregate a_0 and the function returns 0. We propose to proceed by construction and built a path $\overline{\pi}$ satisfying the Theorem 2. We use the following algorithm to compute a set of delays d_i, for $i = 1 \ldots n$ and prove that the $a_0.m \overset{(d_1,t_1)}{\longrightarrow} a_1.m \longrightarrow \ldots \overset{(d_n,t_n)}{\longrightarrow} a_n.m$ is a run of the TPN associated with the TAG.

Input: an abstract path $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$
Output: a concrete path $\overline{\pi} = m_0 \overset{(d_1,t_1)}{\longrightarrow} m_1 \longrightarrow \ldots \overset{(d_n,t_n)}{\longrightarrow} m_n$
begin
1 $\forall i = 1 \ldots n$
2 $d_i \leftarrow \alpha_{i-1_{t_i}}$
3 $\forall i = n-1 \ldots 1$
4 $k = LastNew_i(t_{i+1})$
5 **If**$(\sum_{j=k}^{i-1} d_{j+1} < t_{i+1_{min}})$

6 $\quad \forall k \leq j < i$
7 $\quad\quad d_{j+1} = Max(d_{j+1}, \alpha_{j_{t_{i+1}}} - \alpha_{j+1_{t_{i+1}}})$
8 **If**$(\sum_{j=k}^{i-1} d_{j+1} > t_{i+1_{\max}})$
9 $\quad \forall k \leq j < i$
10 $\quad\quad d_{j+1} = Min(d_{j+1}, \alpha_{j_{t_{i+1}}} - \alpha_{j+1_{t_{i+1}}})$
11 $\forall t \in Enable(a_i.m)$
12 $l = LastNew_i(t)$
13 **If**$((k > l) \wedge (t_{\max} - \sum_{j=l}^{k-1} d_{j+1} < t_{i+1_{\min}}))$
14 $\quad \forall l \leq j < k$
15 $\quad\quad\quad d_{j+1} = Min(d_{j+1}, \alpha_{j_t} - \alpha_{j+1_t})$
end

The intuition of the above algorithm is to build a concrete path $\overline{\pi}$ guided by the abstract path π. $\overline{\pi}$ is built by traversing π by backtracking. Initially (lines $1-2$), the stay time at each marking is set to the minimum i.e., as soon as the desired transition is fireable. Then, starting from the last aggregate, each time a transition t_{i+1} is fired from an aggregate a_i (for $i = 1 \ldots n$), the firability conditions are ensured by (possibly) changing the time that is elapsed before reaching a_i. Roughly speaking, two conditions must be satisfied in order to make the transition t_{i+1} fireable from a_i (i.e., $a_i.m$): (1) The first condition is that the elapsed time, since t_{i+1} became enabled for the last time, belongs to the interval $[t_{i+1_{min}}, t_{i+1_{max}}]$ while the second condition (2), is that there is no transition t enabled by a_i that prevents the firing of t_{i+1}. The only way for the last condition to be satisfied is that t has been enabled (for the lat time) before t_{i+1} and the elapsed time between the moment t became enabled and t_{i+1} became enabled is strictly greater than $t_{max} - t_{i+1_{min}}$. The first condition is treated at lines $5-10$: If the elapsed time since t_{i+1} has been enabled for the last time and the current state ($a_i.m$) is strictly smaller than $t_{i+1_{min}}$ (lines $5-7$) then it must be increased without exceeding $t_{i+1_{max}}$. This is ensured by the fact that, by construction of the TAG, $\sum_{j=k}^{i-1}(\alpha_{j_{t_{i+1}}} - \alpha_{j+1_{t_{i+1}}}) = T_{i+1_{min}}$ and $\sum_{j=k}^{i-1}(\alpha_{j_t} - \alpha_{j+1_t}) \leq t_{i+1_{max}}$ for any transition $t \in\downarrow (a_{i-1}, t_i)$. Now, the elapsed time since the last time t_{i+1} became enabled can exceed $t_{i+1_{max}}$. This can occur if, in order to ensure the firing of some transition t_j (for $j > i+1$) this time has been increased by the algorithm (lines $5-7$). Thus, one has to decrease this time while maintaining the firability of the transition t_j. This is ensured by lines $8-10$. The last condition that could prevent t_{i+1} from being fireable at a_i. is that condition (2) is violated: the time elapsed between the moment some transition t, enabled before, t_{i+1}, and the moment t_{i+1} became enabled is bigger than $t_{max} - t_{i+1_{min}}$. This can happen when the firing of some transition t_j, with $j > i+1$, involved the increase of this quantity of time. This case is treated at lines $11-15$, by fixing this problem while maintaining the future firability of t_j.

Thus, the algorithm ensures the construction of a run of the TPN associated with the TAG that has the same trace. It is clear that the values of d_i, for $i = 1 \ldots n$, respects the conditions of Theorem 2. Now, Theorem 1 ensures that if $m_n \xrightarrow{d}$, for some $d \in \mathbb{R}_{\geq 0}$, then $d \leq \Delta(a_n)$. Finally, given $d \in \mathbb{R}_{\geq 0}$ s.t.,

$d \leq \Delta(a_n)$, the algorithm used to build $\overline{\pi}$ implies that the involved markings are reached as soon as possible. By construction of the TAG, $\Delta(a_n)$ is the maximum time the system can stay at m_n.

Using the above results one can use the TAG associated with a TPN in order to analyse both event and state based properties. In particular, we can check whether a given marking (resp. transition) is reachable (resp. is fireable) before (or after) some time.

4 Checking Time Reachability Properties

Our ultimate goal is to be able, by browsing the TAG associated with a TPN, to check timed reachability properties. For instance, we might be interested in checking whether some state-based property φ is satisfied within a time interval $[d, D)$, with $d \in \mathbb{N}$ and $D \in (\mathbb{N} \cup \{\infty\})$, starting from the initial marking. The following usual reachability properties belong to this category.

1. $\exists\Diamond_{[d;D]}\varphi$: There exists a path starting from the initial state, consuming between d and D time units and leading to a state that satisfies φ.

2. $\forall\Box_{[d;D]}\varphi$: For all paths starting from the initial state, all the states, that are reached after d and before D time units, satisfy φ.

3. $\forall\Diamond_{[d;D]}\varphi$: For all paths starting from the initial state, there exists a state in the path, reached after d and before D time units that satisfies φ.

4. $\exists\Box_{[d;D]}\varphi$: There exists a path from the initial state where all the states, that are reached after d and before D time units, satisfy φ.

For the verification of time properties, an abstraction-based approach should allow the computation of the minimum and maximum elapsed time over any path. In the following, we establish that the TAG allows such a computation.

Definition 8. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta \rangle$ be the corresponding TAG. Let $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$ be a path in G. For each aggregate a_i (for $i = 0 \ldots n$), $MinAT_\pi(a_i)$ (resp. $MaxAT_\pi(a_i)$) denotes the minimum (resp. maximum) elapsed time between a_0 and a_i. In particular, $MinAT(a_0) := 0$ and $MaxAT(a_0) := \Delta(a_0)$.*

Proposition 1. *Let $\mathcal{N}$ be a TPN and let $G = \langle \mathcal{A}, T, a_0, \delta \rangle$ be the corresponding TAG. Let $\pi = a_0 \xrightarrow{t_1} a_1 \longrightarrow \ldots \xrightarrow{t_n} a_n$ be a path in G. We denote by α_{i_t} (resp. β_{i_t}) the dynamic earliest (resp. latest) firing time of a transition t at aggregate a_i, for $i = 1 \ldots n$. Then, $\forall i = 1 \ldots n$, the following holds:*

- $MinAT_\pi(a_i) = MinAT_\pi(a_{i-1}) + \alpha_{i-1_{t_i}}$
- $MaxAT_\pi(a_i) = MaxAT(a_{i-1}) + Min_{t \in Enable(a_i)} SCR(a_i, t)$

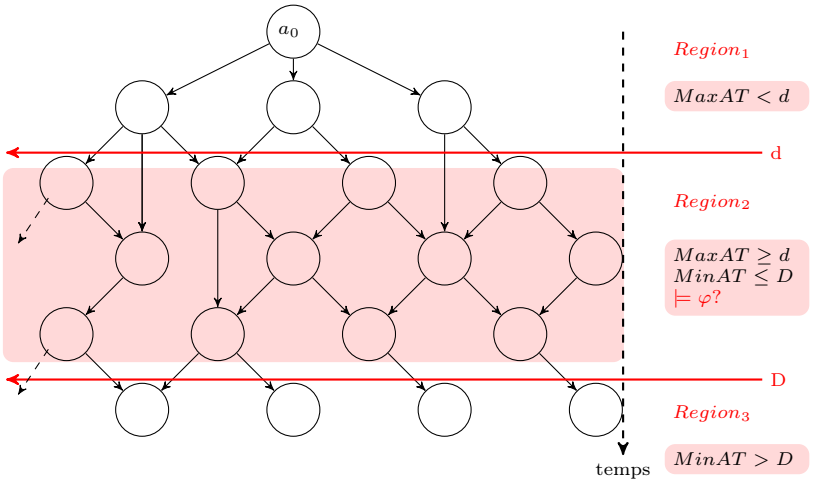


Fig. 3. Reachability analysis on the TAG

Using the previous proposition, one can browse the TAG graph and compute the minimum and maximum bounds of the elapsed time of the current path on-the-fly. If a path of the TAG is considered as a counterexample for some time reachability property, one can use the algorithm given in the proof of Theorem 2 in order to build a concrete counterexample. Here we do not give the detailed algorithms for checking reachability properties on-the-fly, but we give the main intuition. The TAG is represented as a tree which is partitioned into three regions (see. Figure 3). The first region ($Region_1$) contains the aggregates that are reachable strictly before d time units. The second region ($Region_2$) contains the aggregates that are reachable between d and D time units and the last region contains the aggregates that are reachable strictly after D time units. In case $D = \infty$ $Region_3$ is empty. By doing so, the verification algorithms behave as follows: only aggregates belonging to $Region_2$ are analyzed with respect to φ. $Region_1$ must be explored in order to compute the maximal and minimum access time of the traversed aggregates, but $Region_3$ is never explored. In fact, as soon as an aggregate is proved to belong to $Region_3$ the exploration of the current path is stopped. Furthermore, one has to check for a particular kind of Zeno behavior: if a cycle involving only aggregates whose minimal and maximal access times are equal, then the exploration of the current branch is stopped.

For instance checking the formula number 1 is reduced to the search of an aggregate a in $Region_2$ that satisfies φ. As soon as such an aggregate is reached the checking algorithm stops the exploration and returns *true*. When, all the aggregates of $Region_2$ are explored (none satisfies φ) the checking algorithm returns *false*. Dually, the formula number 2 is proved to be unsatisfied as soon as an aggregate in $Region_2$ that do not satisfy φ is reached. When all the aggregates of $Region_2$ are explored (each satisfies φ) the checking algorithm returns *true*.

Checking formulae number 3 and 4 is slightly more complicated. In fact, checking formula number 3 is reduced to check if, along any path in $Region_2$, there exists at least one aggregate satisfying φ. As soon as a path in $Region_2$ is completely explored without encountering an aggregate satisfying φ, the exploration is stopped and the checking algorithm returns *false*. Otherwise, it returns *true*. Finally, checking formula 4 is reduced to check that there exists a path in $Region_2$ such that all the aggregates belonging to this path satisfy φ. This formula is proved to be true as soon as such a path is found. Otherwise, when all the paths of $Region_2$ are explored (none satisfies the desired property), the checking algorithm returns *false*.

5 Related works

This section reviews the most known techniques, proposed in the literature, that abstract and analyse the state space of real-time systems described by means of TPN. Abstraction techniques aim at constructing, by removing some irrelevant details, a contraction of the state space of the model, which preserves properties of interest. The existing abstraction approaches mainly differ in the states agglomeration criteria, the characterization of states and state classes (interval states or clock states), the kind of preserved properties.

The *States Class Graph* (SCG) [3] was the first method of state space representation adapted to TPNs. A class (m, D) is associated with a marking m and a time domain D represented by a set of inequalities over variables. The variables represented in the SCG are the firing time intervals of enabled transitions. The SCG allows for the verification of some TPN properties like reachability, boundness. However, it preserves the linear time properties only. To address this limitation, a refinement of the method was proposed in [24], in the form of a graph called *Atomic States Class Graph* (ASCG). The authors use a cutting of state class by adding linear constraints so that each state of an atomic class has a successor in all the following classes. With this improvement, they are able to verify CTL* properties on TPN, but with the limitation that the time intervals of transitions are bounded. A new approach for the construction of atomic classes was proposed in [4] and allows the verification of CTL* without restriction on time intervals. The state class approach is implemented in a software tool called TINA [5].

The *Zones Based Graph* (ZBG) [10] is an other approach allowing to abstract the TPN state space. This approach is inspired by the *Region Graph* (RG) [1] technique, initially introduced for *timed automata*. In practice, the number of regions is too large for an effective analysis, thus, the regions are grouped into a set of *zones*. A zone is a convex union of regions and can be represented by a DBM (Difference Bound Matrix) [8]. In [10], the clocks of transitions are directly encoded within the zones. This allows to verify temporal and quantitative properties but not CTL* properties. As for timed automata, a disadvantage of the method is the necessary recourse to approximation methods (k-approximation or kx-approximation) in the case where the infinity is used in the bounds of time

intervals. Lime and Roux also used TPNs to model system behavior [17]. They used the state class approach to build a *timed automaton* that preserves the behavior of the TPN using as less clock variables as possible. The resulting model is then verified using the UPPAAL tool [16]. However, even though UPPAAL can answer about quantitative temporal properties, it can only verify a subset of TCTL. Adding a new transition to measure time elapse was proposed in [6] to perform TCTL model-checking in TPNs. Using this transition, TCTL formulae are translated into CTL formulae. Then a ZBG for TPN is refined leading to a graph called *Atomic Zone Based Graph* (AZBG) that preserves CTL properties.

Unlike the TAG, in all existing approaches, the time information does not appear explicitly in nodes which leads to additional and costly calculations such as: the manipulation of DBM to encode the zones (for zones based approaches) and the classes (for state-class based approaches), the approximations to counter the problem of unbounded transitions, conversion of graphs to timed automata (using UPPAAL) to model check properties (etc). In our work the time information is encoded within the aggregates allowing to check time properties just by browsing the graph, which has a significant impact on the construction complexity. The encoding of the timing information in the aggregates is such that the minimum and maximum elapsed time in every path of the TAG can be computed.

6 Experimental results

The efficiency of the verification of timed reachability properties is closely linked with the size of the explored structure to achieve this verification. Thus, it was important to first check that the TAG is a suitable/reduced abstraction before performing verification on it. Our approach for building TAG-TPN was implemented in a prototype tool (written in C++), and used for experiments in order to validate the size of the graphs generated by the approach (note that the prototype was not optimized for time efficiency yet, therefore no timing figures are given in this section). All results reported in this section have been obtained on a Mac-os with 2 gigahertz Intel with 8 gigabytes of RAM. The implemented prototype allowed us to have first comparison with existing approaches with respect to the size of obtained graphs. This section is dedicated to report, compare and discuss the experimental results obtained with three approaches: SCG, ZBG and TAG-TPN. Notice that we used the ROMEO tool to build both SCGs and ZBGs. The built versions preserve Linear-time Temporal Logic (LTL) properties. We tested our approach on several TPN models and we report here the obtained results for three well known examples of parametric TPN models.

The considered models are: (1) a TPN representing a composition of producer/consumer models by fusion of a single buffer (of size 5) [12], (2) the second example (adapted from [19]) is the Fischer's protocol for mutual exclusion, and (3) the last is the train crossing example [4].

Table1 reports the results obtained with the SCG, the ZBG and the TAG-TPN approaches, in terms of graph size number of nodes/number of edges).

Parameters	SCG (with Tina) (nodes / arcs)	ZBG (with Romeo) (nodes / arcs)	TAG-TPN (nodes / arcs)
Nb. prod/cons	TPN model of producer/consumer		
1	34 / 56	34 / 56	34 / 56
2	748 / 2460	593 / 1 922	740 / 2438
3	4604 / 21891	3240 / 15200	4553 / 21443
4	14086 / 83375	9504 / 56038	13878 / 80646
5	31657 / 217423	20877 / 145037	30990 / 207024
6	61162 / 471254	39306 / 311304	60425 / 449523
7	107236 / 907 708	67224 / 594795	106101 / 856050
Nb. processes	Fischer protocol		
1	4 / 4	4 / 4	4 / 4
2	18 / 29	19 / 32	20 / 32
3	65 / 146	66 / 153	74 / 165
4	220 / 623	221 / 652	248 / 712
5	727 / 2536	728 / 2 615	802 / 2825
6	2378 / 9154	2379 / 10098	2564 / 10728
7	7737 / 24744	7738 / 37961	8178 / 39697
8	25080 / 102242	25081 / 139768	26096 / 144304
Nb. processes	Train crossing		
1	11 / 1 4	11 / 1 4	11 / 14
2	123 / 218	114 / 200	123 / 218
3	3101 / 7754	2817 / 6944	2879 / 7280
4	134501 / 436896	122290 / 391244	105360 / 354270

Table 1. Experimentation results

The obtained preliminary results show that the size of the TAG is comparable to the size of the graphs obtained with the ZBG and the SCG approaches. The TAG achieves better performances than both SCG and ZBG for the train crossing example, while it is slightly worse for the Fischer's protocol and performs similarly to SCG but worse than ZBG for the producer/consumer example.

This is an encouraging result because of the following reasons: The TAG allows for checking *timed* properties while the SCG approach do not. Also, it can be used for the verification of event-based timed properties while the ZBG approach do not. An other difference consists in the fact that the verification of timed properties can be achieved directly on the TAG, without any synchronisation with an additional automaton (representing the formula to be checked), nor any prior step of translation to timed automata. Moreover, using the algorithm given in the proof of Theorem 2, and in the prospect of using the TAG in order to check timed properties, one can exhibit (e.g., from a counterexample abstract path in the TAG) an explicit run involving the time spent at each reached marking. Finally, we claim that the TAG is a suitable abstraction for further reductions, especially the partial order reduction which is based on the exploitation of the independency between the TPN transitions. The third exam-

ple of Figure 2 is a typical illustration of the gain one could have by applying such a reduction.

7 Conclusion

We proposed a new symbolic graph for the abstraction of the TPN state space. The proposed graph, called TAG, produces a finite representation of the bounded TPN behavior and allows for analyzing of timed reachability properties. Unlike, the existing approaches, our abstraction can be directly useful to check *timed* logic properties. We think that our approach is more understandable than the SCG and the ZBG approaches (the two main approaches for TPNs analysis since three decades) and easily implementable. Another feature of our approach is that each path of the TAG can be matched with a concrete path of the TPN model where the elapsed time at each encountered state is exhibited.

Our ultimate goal is to use the TAG traversal algorithm for the verification of timed reachability properties expressed in the *TCTL* logic. Several issues have to be explored in the future: We first have to improve our implementation so that time consumption criterion can be taken into account in our comparison to existing tools. We should also, carry out additional experimentations (using more significant use cases) to better understand the limits of our approach and to better compare the TAG technique to the existing approaches. Second, we believe that partial order reduction techniques can be used to reduce the size of the TAG while preserving time properties but without necessarily preserving all the paths of the underlying TPN. Finally, two challenging perspectives can be considered in the future: (1) the design and the implementation of model checking algorithms for verification of *TCTL* formulae, and (2), the extension of our approach to timed automata.

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. B. Berthomieu and M. Diaz. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Trans. Software Eng.*, 17(3):259–273, 1991.
3. B. Berthomieu and M. Menasche. An Enumerative Approach for Analyzing Time Petri Nets. In *IFIP Congress*, pages 41–46, 1983.
4. B. Berthomieu and F. Vernadat. State Class Constructions for Branching Analysis of Time Petri Nets. In *TACAS 2003*, volume 2619 of *LNCS*, pages 442–457. Springer, 2003.
5. B. Berthomieu and F. Vernadat. Time Petri Nets Analysis with TINA. In *QEST*, pages 123–124, 2006.
6. H. Boucheneb, G. Gardey, and O. H. Roux. TCTL Model Checking of Time Petri Nets. *J. Log. Comput.*, 19(6):1509–1540, 2009.
7. M. Boyer and O. H. Roux. Comparison of the Expressiveness of Arc, Place and Transition Time Petri Nets. In *ICATPN 2007*, volume 4546 of *LNCS*, pages 63–82. Springer, 2007.

8. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer-Verlag, 1990.
9. G. Gardey, D. Lime, M. Magnin, and O. (h. Roux. Roméo: A Tool for Analyzing time Petri nets. In *In Proc. CAVÕ05, vol. 3576 of LNCS*, pages 418–423. Springer, 2005.
10. G. Gardey, O. H. Roux, and O. F. Roux. Using Zone Graph Method for Computing the State Space of a Time Petri Net. In *FORMATS 2003*, volume 2791 of *LNCS*, pages 246–259. Springer, 2003.
11. R. Hadjidj and H. Boucheneb. Improving state class constructions for CTL* model checking of time Petri nets. *STTT*, 10(2):167–184, 2008.
12. R. Hadjidj and H. Boucheneb. On-the-fly TCTL model checking for time Petri nets. *Theor. Comput. Sci.*, 410(42):4241–4261, 2009.
13. K. Klai, N. Aber, and L. Petrucci. To appear in a new approach to abstract reachability state space of time petri nets. In *To appear in 20th International Symposium on Temporal Representation and Reasoning, TIME 2013*, 2013.
14. K. Klai, N. Aber, and L. Petrucci. Verification of reachability properties for time petri nets. In *Reachability Problems - 7th International Workshop, RP 2013*, volume 8169 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2013.
15. K. G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *FCT '95*, volume 965 of *LNCS*, pages 62–88. Springer, 1995.
16. K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL: Status and Developments. In *CAV*, pages 456–459, 1997.
17. D. Lime and O. H. Roux. Model Checking of Time Petri Nets Using the State Class Timed Automaton. *Discrete Event Dynamic Systems*, 16(2):179–205, 2006.
18. P. M. Merlin and D. J. Farber. Recoverability of modular systems. *Operating Systems Review*, 9(3):51–56, 1975.
19. W. Penczek, A. Pólrola, and A. Zbrzezny. SAT-Based (Parametric) Reachability for a Class of Distributed Time Petri Nets. *T. Petri Nets and Other Models of Concurrency*, 4:72–97, 2010.
20. C. A. Petri. Concepts of net theory. In *MFCS'73*. Mathematical Institute of the Slovak Academy of Sciences, 1973.
21. M. Pezzè and M. Young. Time Petri Nets: A Primer Introduction. In *Tutorial at the Multi-Workshop on Formal Methods in Performance Evaluation and Applications*, 1999.
22. C. Ramchandani. Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Cambridge, MA, USA, 1974.
23. J. Sifakis. Use of Petri nets for performance evaluation. *Acta Cybern.*, 4:185–202, 1980.
24. T. Yoneda and H. Ryuba. CTL model checking of time Petri nets using geometric regions. 1998.

SMT-based Abstract Temporal Planning*

Artur Niewiadomski[1] and Wojciech Penczek[1,2]

[1] ICS, Siedlce University, 3-Maja 54, 08-110 Siedlce, Poland
artur.niewiadomski@uph.edu.pl
[2] ICS, Polish Academy of Sciences, Jana Kazimierza 5, 01-248 Warsaw, Poland
penczek@ipipan.waw.pl

Abstract. An abstract planning is the first phase of the web service composition in the PlanICS framework. A user query specifies the initial and the expected state of a plan in request. The paper extends PlanICS with a module for temporal planning, by extending the user query with an LTL^k_{-X} formula specifying temporal aspects of world transformations in a plan. Our solution comes together with an example, an implementation, and experimental results.

Keywords: Web Service Composition, SMT, Abstract Planning, Temporal Planning, LTL

1 Introduction

Web service composition within Service-Oriented Architecture (SOA) [2] is still attracting a lot of interest, being a subject of many theoretical and practical approaches. The main idea consists in dealing with independent (software) components available via well-defined interfaces. As typically a simple web service does not satisfy the user objective, a composition is investigated in order to make the user fully satisfied. An automatic composition of Web services aims at relieving the user of a manual preparation of detailed execution plans, matching services to each other, and choosing optimal providers for all the components. The problem of finding such a satisfactory composition is NP-hard and well known in the literature as the Web Service Composition Problem (WSCP) [2, 1, 21]. There are many various approaches to solve WSCP [14, 16], some of them we discuss in the next section.

In this paper, we follow the approach of the system PlanICS [8, 9], which has been inspired by [1]. The main assumption is that all the web services in the domain of interest as well as the objects which are processed by the services, can be strictly classified in a hierarchy of *classes*, organised in an *ontology*. Another key idea consists in dividing planning into several stages. The first phase, called the *abstract planning*, deals with *classes of services*, where each class represents a set of real-world services. This phase has been implemented in PlanICS using

* This work has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.

two approaches: one based on a translation to SMT-solvers [17] and another one exploiting genetic algorithms [22]. The second phase, called *concrete planning*, deals with *concrete services*. Thus, while the first phase produces an *abstract plan*, it becomes a *concrete plan* in the second phase. Such an approach enables to reduce dramatically the number of concrete services to be considered as they are already eliminated in the abstract planning phase. This paper focuses on the abstract planning problem, but extends it to so called temporal planning. This extension together with the experimental results is the main contribution of the paper. The main idea behind this approach consists in providing the user with a possibility to specify not only the first and the expected state of a plan in request, but also to specify temporal aspects of state transformations in a plan. To this aim we introduce two general types of atomic properties for writing a temporal formula, namely *propositions* and *level constraints*. The propositions are used to describe (intermediate) states of a plan in terms of existence (or non-existence) of objects and abstract values of object attributes. The level constraints, built over a special set of objects, are used for influencing a service ordering within solutions. However, in order to express such restrictions the user has to rely on some knowledge about the planning domain. In order to get this knowledge, the planner can be first run without temporal constraints and then these restrictions can be added after a non-temporal planning results have been obtained.

We propose a novel approach based on applying SMT-solvers. Contrary to a number of other approaches, we focus not only on searching for a single plan, but we attempt to find all *significantly different* plans. We start with defining the abstract planning problem (APP, for short). Then, we present our original approach to APP based on a compact representation of abstract plans by multisets of service types. We introduce the language of LTL^{k}_{-X} for specifying the temporal aspects of the user query. This approach is combined with a reduction to a task for an SMT-solver. The encoding of blocking formulas allows for pruning the search space with many sequences which use the same multiset of service types in some plan already generated. Moreover, we give details of our algorithms and their implementations that are followed by experimental results. To the best of our knowledge, the above approach is novel, and as our experiments show it is also very promising.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Section 3 deals with the abstract planning problem. In Section 4 the temporal planning is presented. An example of an abstract temporal planning is shown in Section 5. Section 6 discusses the implementation and the experimental results of our planning system. The last section summarizes this paper and discusses a further work.

2 Related Work

A classification matrix aimed at the influence on the effort of Web service composition is presented in [14]. According to [14], situation calculus [6], Petri nets [11], theorem proving [20], and model checking [23] among others belongs to AI plan-

ning. A composition method closest to ours based on SMT is presented in [16], where the authors reduce WSCP to a reachability problem of a state-transition system. The problem is encoded by a propositional formula and tested for satisfiability using a SAT-solver. This approach makes use of an ontology describing a hierarchy of types and deals with an inheritance relation. However, we consider also the states of the objects, while [16] deals with their types only. Moreover, among other differences, we use a multiset-based SMT encoding instead of SAT.

Most of the applications of SMT in the domain of WSC is related to the automatic verification and testing. For example, a message race detection problem is investigated in [10], the paper [4] takes advantage of symbolic testing and execution techniques in order to check behavioural conformance of WS-BPEL specifications, while [15] exploits SMT to verification of WS-BPEL specifications against business rules.

Recently, there have also appeared papers dealing with temporal logics in the context of WSC. Bersani et al. in [5] present a formal verification technique for an extension of LTL that allows the users to include constraints on integer variables in formulas. This technique is applied to the substitutability problem for conversational services. The paper [13] deals with the problem of automatic service discovery and composition. The authors characterize the behaviour of a service in terms of a finite state machine, specify the user's requirement by an LTL formula, and provide a translation of the problem defined to SAT. However, the paper does not specify precisely experimental results and such important details as, e.g., the number of services under consideration. An efficient application of the authors method is reported for plans of length up to 10 only. The authors of [3] address the issue of verifying whether a composite Web services design meets some desirable properties in terms of deadlock freedom, safety, and reachability. The authors report on automatic translation procedures from the automata-based design models to the input language of the NuSMV verification tool. The properties to be verified can be expressed as LTL or CTL formulae.

Searching for plans meeting temporal restrictions is also a topic of interest of a broad planning community. The PDDL language [12] has been also extended with LTL-like modal operators, but for planning automata-based methods are used instead of SMT-based symbolic ones.

3 Abstract Planning

This section introduces APP as the first stage of WSCP in the PlanICS framework. First, the PlanICS ontology is presented. Next, we provide some basic definitions and explain the main goals of APP.

3.1 PlanICS Ontology

The OWL language [19] is used as the PlanICS ontology format. The concepts are organized in an inheritance tree of *classes*, all derived from the base class - *Thing*. There are 3 children of *Thing*: *Artifact*, *Stamp*, and *Service* (Fig. 1).

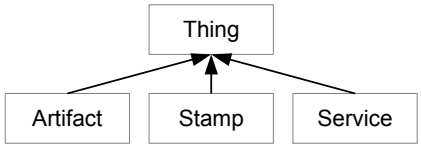


Fig. 1. The base classes in PlaniCS ontology

The branch of classes rooted at *Artifact* is composed of the types of the objects, which the services operate on. Each object consists of a number of attributes, whereas an attribute consists of a name and a type. Note that the types of the attributes are irrelevant in the abstract planning phase as they are not used by the planner. The values of the attributes of an object determine its state, but in the abstract planning it is enough to know only whether an attribute does have some value (i.e., is set), or it does not (i.e., it is null). The *Stamp* class and its descendants define special-purpose objects, often useful in constructing a user query, and in the planning process. A stamp is a specific type aimed at a confirmation of the service execution. The specialized descendants of the *Service* class can produce the *stamp* being an instance of any subtype of *Stamp* and describing additional execution features. Note that each service produces exactly one confirmation object. The classes derived from *Artifact* and *Stamp* are called the *object types*.

Each class derived from *Service*, called a *service type*, stands for a description of a set of real-world services. It contains a formalized information about their activities. A service type affects a set of objects and transforms them into a new set of objects. The detailed information about this transformation is contained in the attributes of a service type: the sets *in*, *inout*, and *out*, and the Boolean formulas *pre* and *post* (*pre* and *post*, for short). These sets enumerate the objects, which are processed by the service. The objects of the *in* set are read-only, i.e., they are passed unchanged to the next world. Each object of *inout* can be modified - the service can change some values of its attributes. The objects of *out* are produced by the service.

3.2 Basic definitions

Let $\mathbb{I}$ denote the set of all *identifiers* used as the type names, the objects, and the attributes. In APP we deal with abstract values only, the types of the attributes are irrelevant, and we identify the attributes with their names. Moreover, we denote the set of all attributes by A, where $\mathrm{A} \subset \mathbb{I}$. An *object type* is a pair $(t, Attr)$, where $t \in \mathbb{I}$, and $Attr \subseteq \mathrm{A}$. That is, an object type consists of the type name and a set of the attributes. By $\mathbb{P}$ we mean a set of all object types.

Example 1. Consider the following exemplary ontology containing in addition to *Thing* also the class *Artifact* and *Stamp*. The class *Artifact* corresponds to the object type $(Artifact, \{id\})$ (the only attribute is an identifier) while the class

Stamp corresponds to the object type $(Stamp, \{serviceClass, serviceId, level\})$, introducing the attributes describing the service generating the stamp, and the position of this service in an execution sequence we consider.

We define also a transitive, irreflexive, and antisymmetric *inheritance* relation $Ext \subseteq \mathbb{P} \times \mathbb{P}$, such that $\big((t_1, A_1), (t_2, A_2)\big) \in Ext$ iff $t_1 \neq t_2$ and $A_1 \subseteq A_2$. That is, a subtype contains all the attributes of a base type and optionally introduces more attributes. An *object* o is a pair $o = (id, type)$, where $id \in \mathbb{I}$ and $type \in \mathbb{P}$. By $type(o)$ we denote the type of o. The set of all objects is denoted by $\mathbb{O}$. Amongst all the objects we distinguish between the *artifacts* (the instances of the *Artifact* type) and the *stamps* (the instances of the *Stamp* type). The set of all the stamps is denoted by $\mathbb{ST}$, where $\mathbb{ST} \subseteq \mathbb{O}$. Moreover, we define the function $attr : \mathbb{O} \longmapsto 2^{\mathrm{A}}$ returning the set of all attributes for each object of $\mathbb{O}$.

Service types and user queries. The service types available for composition are defined in the ontology by *service type specifications*. The user goal is provided in a form of a *user query specification*, which is then extended by a temporal formula. Before APP, all the specifications are reduced to sets of objects and *abstract formulas* over them. An **abstract formula** over a set of objects O and their attributes is a DNF formula without negations, i.e., the disjunction of clauses, referred to as *abstract clauses*. Every abstract clause is the conjunction of literals, specifying abstract values of object attributes using the functions $isSet$ and $isNull$. In the abstract formulas used in APP, we assume that no abstract clause contains both $isSet(o.a)$ and $isNull(o.a)$, for the same $o \in O$ and $a \in attr(o)$. For example $(isSet(o.a) \wedge isSet(o.b)) \vee isNull(o.a)$ is a correct abstract formula. The syntax of the specifications of the user queries and of the service types is the same and it is defined below.

Definition 1. *A* **specification** *is a 5-tuple* $(\mathrm{in}, \mathrm{inout}, \mathrm{out}, \mathrm{pre}, \mathrm{post})$*, where* in*,* inout*,* out *are pairwise disjoint sets of objects, and* pre *is an abstract formula defined over objects from* $\mathrm{in} \cup \mathrm{inout}$*, while* post *is an abstract formula defined over objects from* $\mathrm{in} \cup \mathrm{inout} \cup \mathrm{out}$*.*

A user query specification q or a service type specification s is denoted by $spec_x = (in_x, inout_x, out_x, pre_x, post_x)$, where $x \in \{q, s\}$, resp. In order to formally define the *user queries* and the *service types*, which are interpretations of their specifications, we need to define the notions of *valuation functions* and *worlds*.

Definition 2. *Let* $\varphi = \bigvee_{i=1..n} \alpha_i$ *be an abstract formula. A* **valuation of the attributes** *over* α_i *is the partial function* $v_{\alpha_i} : \bigcup_{o \in \mathbb{O}} \{o\} \times attr(o) \longmapsto \{true, false\}$*, where:*

- $v_{\alpha_i}(o, a) = true$ *if* $isSet(o.a)$ *is a literal of* α_i*, and*
- $v_{\alpha_i}(o, a) = false$ *if* $isNull(o.a)$ *is a literal of* α_i*, and*
- $v_{\alpha_i}(o, a)$ *is undefined, otherwise.*

We define the restriction of a valuation function v_{α_i} to a set of objects $O \subset \mathbb{O}$ as $v_{\alpha_i}(O) = v_{\alpha_i}\big|_{\bigcup_{o \in O}\{o\}\times attr(o)}$. The undefined values appear when the interpreted abstract formula does not specify abstract values of some attributes, which is a typical case in the WSC domain. The undefined values are used also for representing families of total valuation functions. Next, for a partial valuation function f, by $total(f)$ we denote the family of the total valuation functions on the same domain, which are consistent with f, i.e., agree on the values defined of f. Moreover, we define a *family of the valuation functions* $\mathcal{V}_\varphi$ over the abstract formula φ as the union of the sets of the consistent valuation functions over every abstract clause α_i, i.e., $\mathcal{V}_\varphi = \bigcup_{i=1}^{n} total(v_{\alpha_i})$. The restriction of the family of functions $\mathcal{V}_\varphi$ to a set of objects O and their attributes is defined as $\mathcal{V}_\varphi(O) = \bigcup_{i=1}^{n} total(v_{\alpha_i}(O))$.

Definition 3. *A* **world** w *is a pair* (O_w, v_w)*, where* $O_w \subseteq \mathbb{O}$ *and* $v_w = v(O_w)$ *is a total valuation function equal to some valuation function* v *restricted to* O_w*. The* size *of* w*, denoted by* $|w|$ *is equal to* $|O_w|$*.*

That is, a world represents a state of a set of objects, where each attribute is either set or null. By a *sub-world* of w we mean a world built from a subset of O_w and v_w restricted to the objects from the chosen subset. Moreover, a pair consisting of a set of objects and a family of total valuation functions defines a *set of worlds*. That is, if $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a family of total valuation functions and $O \subseteq \mathbb{O}$ is a set of objects, then $\big(O, \mathcal{V}(O)\big)$ means the set $\{\big(O, v_i(O)\big) \mid 1 \leq i \leq n\}$, for $n \in \mathbb{N}$. Finally, the set of all worlds is denoted by $\mathbb{W}$.

Now, we are in a position to define a *service type* and a (basic) *user query* as an interpretation of its specification. In the next section the user query is extended to a temporal version.

Definition 4. *Let* $spec_x = (\mathrm{in}_x, \mathrm{inout}_x, \mathrm{out}_x, \mathrm{pre}_x, \mathrm{post}_x)$ *be a user query or a service type specification, where* $x \in \{q, s\}$*, resp. An interpretation of* $spec_x$ *is a pair of world sets* $x = (W^x_{pre}, W^x_{post})$*, where:*

- $W^x_{pre} = \big(\mathrm{in}_x \cup \mathrm{inout}_x, \mathcal{V}^x_{pre}\big)$*, where* $\mathcal{V}^x_{pre}$ *is the family of the valuation functions over* pre_x*,*
- $W^x_{post} = \big(\mathrm{in}_x \cup \mathrm{inout}_x \cup \mathrm{out}_x, \mathcal{V}^x_{post}\big)$*, where* $\mathcal{V}^x_{post}$ *is the family of the valuation functions over* post_x*.*

An interpretation of a user query (service type) specification is called simply a user query *(*service type*, resp.).*

For a service type (W^s_{pre}, W^s_{post}), W^s_{pre} is called the *input world set*, while W^s_{post} - the *output world set*. The set of all the service types defined in the ontology is denoted by $\mathbb{S}$. For a user query (W^q_{pre}, W^q_{post}), W^q_{pre} is called the *initial world set*, while W^q_{post} - the *expected world set*, and denoted by W^q_{init} and W^q_{exp}, respectively.

Abstract Planning Overview. The main goal of APP is to find a composition of service types satisfying a user query, which specifies some initial and some expected worlds as well as some temporal aspects of world transformations.

Intuitively, an initial world contains the objects owned by the user, whereas an expected world consists of the objects required to be the result of the service composition. To formalize it, we need several auxiliary concepts.

Let $o, o' \in \mathbb{O}$ and v and v' be valuation functions. We say that $v'(o')$ is compatible with $v(o)$, denoted by $v'(o') \succ^{obj} v(o)$, iff the types of both objects are the same, or the type of o' is a subtype of type of o, i.e., $type(o) = type(o')$ or $(type(o'), type(o)) \in Ext$, and for all attributes of o, we have that v' agrees with v, i.e., $\forall_{a \in attr(o)} v'(o', a) = v(o, a)$. Intuitively, an object of a richer type (o') is compatible with the one of the base type (o), provided that the valuations of all common attributes are equal.

Let $w = (O, v)$, $w' = (O', v')$ be worlds. We say that the world w' is compatible with the world w, denoted by $w' \succ^{wrl} w$, iff there exists a one-to-one mapping $map : O \longmapsto O'$ such that $\forall_{o \in O} v'(map(o)) \succ^{obj} v(o)$. Intuitively, w' is compatible with w if both of them contain the same number of objects and for each object from w there exists a compatible object in w'. The world w' is called *sub-compatible* with the world w, denoted by $w' \succ^{swrl} w$ iff there exists a sub-world of w' compatible with w.

World transformations. One of the fundamental concepts in our approach concerns a world transformation. A world w, called a *world before*, can be transformed by a service type s, having specification $spec_s$, if w is sub-compatible with some input world of s. The result of such a transformation is a world w', called a *world after*, in which the objects of out_s appear, and, as well as the objects of $inout_s$, they are in the states consistent with some output world of s. The other objects of w do not change their states. In a general case, there may exist a number of worlds possible to obtain after a transformation of a given world by a given service type, because more than one sub-world of w can be compatible with an input world of s. Therefore, we introduce a *context function*, which provides a strict mapping between objects from the worlds before and after, and the objects from the input and output worlds of a service type s.

Definition 5. *A* **context function** $ctx^s_O : in_s \cup inout_s \cup out_s \longmapsto O$ *is an injection, which for a given service type s and a set of objects O assigns an object from O to each object from in_s, $inout_s$, and out_s.*

Now, we can define a world transformation.

Definition 6. *Let $w, w' \in \mathbb{W}$ be worlds, called a* world before *and a* world after*, respectively, and $s = (W^s_{pre}, W^s_{post})$ be a service type. Assume that $w = (O, v)$, $w' = (O', v')$, where $O \subseteq O' \subseteq \mathbb{O}$, and v, v' are valuation functions. Let $ctx^s_{O'}$ be a context function, and the sets IN, IO, OU be the $ctx^s_{O'}$ images of the sets in_s, $inout_s$, and out_s, respect., i.e., $IN = ctx^s_{O'}(in_s)$, $IO = ctx^s_{O'}(inout_s)$, and $OU = ctx^s_{O'}(out_s)$. Moreover, let $IN, IO \subseteq (O \cap O')$ and $OU = (O' \setminus O)$.*

We say that a service type s transforms *the world w into w'* in the context *$ctx^s_{O'}$, denoted by $w \overset{s, ctx^s_{O'}}{\rightarrow} w'$, if for some $v^s_{pre} \in \mathcal{V}^s_{pre}$ and $v^s_{post} \in \mathcal{V}^s_{post}$, all the following conditions hold:*

1. $\big(IN, v(IN)\big) \succ^{wrl} \big(in_s, v^s_{pre}(in_s)\big)$,

2. $(IO, v(IO)) \succ^{wrl} \big(inout_s, v^s_{pre}(inout_s)\big)$,
3. $(IO, v'(IO)) \succ^{wrl} \big(inout_s, v^s_{post}(inout_s)\big)$,
4. $(OU, v'(OU)) \succ^{wrl} \big(out_s, v^s_{post}(out_s)\big)$,
5. $\forall_{o \in (O \setminus IO)}\ \forall_{a \in attr(o)}\ v(o,a) = v'(o,a)$.

Intuitively, (1) the *world before* contains a sub-world built over IN, which is compatible with a sub-world of some input world of the service type s, built over the objects from in_s. (2) The *world before* contains a sub-world built over IO, which is compatible with a sub-world of the input world of the service type s, built over the objects from $inout_s$. (3) After the transformation the state of objects from IO is consistent with $post_s$. (4) The objects produced during the transformation (OU) are in a state consistent with $post_s$. (5) The objects from IN and the objects not involved in the transformation do not change their states.

In the standard way we extend a world transformation to a sequence of world transformations *seq*. We say that a world w_0 *is transformed by the sequence seq* into a world w_n, denoted by $w_0 \overset{seq}{\rightsquigarrow} w_n$, iff there exists a sequence of worlds $\rho = (w_0, w_1, \ldots, w_n)$ such that $\forall_{1 \le i \le n}\ w_{i-1} \overset{s_i, ctx^{s_i}_{O_i}}{\rightarrow} w_i = (O_i, v_i)$ for some v_i. Then, the sequence $seq = (s_1, \ldots, s_n)$ is called a *transformation sequence* and ρ is called a *world sequence*.

Having the transformation sequences defined, we introduce the concept of *user query solutions* or simply *solutions*, in order to define a plan.

Definition 7. *Let* seq *be a transformation sequence,* $q = (W^q_{init}, W^q_{exp})$ *be a user query. We say that* seq *is a* solution *of* q*, if for* $w \in W^q_{init}$ *and some world* w' *such that* $w \overset{\text{seq}}{\rightsquigarrow} w'$*, we have* $w' \succ^{swrl} w^q_{exp}$*, for some* $w^q_{exp} \in W^q_{exp}$*. The world sequence corresponding to* seq *is called a* world solution*. The set of all the (world) solutions of the user query* q *is denoted by* $QS(q)$ *(*$WS(q)$*, resp.).*

Intuitively, by a solution of q we mean any transformation sequence transforming some initial world of q to a world sub-compatible to some expected world of q.

Plans. Basing on the definition of a solution to the user query q, we can now define the concept of an (abstract) plan, by which we mean a non-empty set of solutions of q. We define a plan as an equivalence class of the solutions, which do not differ in the service types used. The idea is that we do not want to distinguish between solutions composed of the same service types, which differ only in the ordering of their occurrences or in their contexts. So we group them into the same class. There are clearly two motivations behind that. Firstly, the user is typically not interested in obtaining many very similar solutions. Secondly, from the efficiency point of view, the number of equivalence classes can be exponentially smaller than the number of the solutions. Thus, two user query solutions are equivalent if they consist of the same number of the same service types, regardless of the contexts.

Definition 8. *Let* $\text{seq} \in QS(q)$ *be a solution of some user query* q*. An abstract plan is a set of all the solutions equivalent to seq, denoted by* $[\text{seq}]_\sim$.

It is important to notice that all the solutions within an abstract plan are built over the same *multiset* of service types, so a plan is denoted using a multiset notation, e.g., the plan $[2S + 4T + 3R]$ consists of 2 services S, 4 services T, and 3 services R.

In order to give the user a possibility to specify not only the initial and expected states of a solution, we extend the user query with an LTL^k_{-X} formula φ specifying temporal aspects of world transformations in a solution. Then, the temporal solutions are these solutions for which the world sequences satisfy φ. This is formally introduced in the next section.

4 Temporal Abstract Planning

In this section we extend the user query by an LTL^k_{-X} temporal formula and a solution to a temporal solution by requiring the temporal formula to be satisfied. The choice of linear time temporal logic is quite natural since our user query solutions are defined as sequences of worlds. The reason for disallowing a direct use of the operator X (by removing it from the syntax) is twofold. Firstly, we still aim at not distinguishing sequences which differ only in the ordering of independent service types. Secondly, if the user wants to introduce the order on two consequtive service types he can use formulas involving level constraints. On the other hand our language and the temporal planning method can be easily extended with the operator X.

We start with defining the set of propositional variables, the level constraints, and then the syntax and the semantics of LTL^k_{-X}.

4.1 Propositional variables

Let $o \in \mathbb{O}$ be an object, $a \in attr(o)$. The set of propositional variables $PV = \{\mathbf{pEx}(o), \mathbf{pSet}(o.a), \mathbf{pNull}(o.a) \mid o \in \mathbb{O}, a \in attr(o)\}$. Intuitively, $\mathbf{pEx}(o)$ holds in each world, where the object o exists, $\mathbf{pSet}(o.a)$ holds in each world, where the object o exists and the attribute a is set, and $\mathbf{pNull}(o.a)$ holds in each world, where the object o exists and the attribute a is null.

In addition to PV we use also the set of *level constraints* $\mathbf{LC}$ over the stamps $\mathbb{ST}$, defined by the following grammar:

$$\begin{aligned} \mathbf{lc} &::= \mathbf{lexp} \sim \mathbf{lexp} \\ \mathbf{lexp} &::= c \mid s.level \mid \mathbf{lexp} \oplus \mathbf{lexp} \end{aligned} \tag{1}$$

where $s \in \mathbb{ST}$, $c \in \mathbb{Z}$, $\oplus \in \{+, -, \cdot, /, \%\}$, $\sim \in \{\leq, <, =, >, \geq\}$, and $/, \%$ stand for integer division and modulus, respectively.

Intuitively, $s.level < c$ holds in each world, where the stamp s exists and the value of its level is smaller than c.

4.2 Syntax of LTL^k_{-X}

The LTL^k_{-X} formulae are defined by the following grammar:

$$\varphi ::= p \mid \neg p \mid \mathbf{lc} \mid \neg \mathbf{lc} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \mathrm{U}_{<k} \varphi \mid \varphi \mathrm{R}_{<k} \varphi.$$

where $p \in PV$, $\mathbf{lc} \in \mathbf{LC}$, and $k \in \mathbb{N}$.

Observe that we assume that the LTL^k_{-X} formulae are given in the *negation normal form* (NNF), in which the negation can be only applied to the propositional variables and the level constraints. The temporal modalities $\mathrm{U}_{<k}$ and $\mathrm{R}_{<k}$ are named as usual k-restricted *until* and *release*, respectively. Intuitively, $\varphi \mathrm{U}_{<k} \psi$ means that eventually, but in less than k steps, ψ holds and always earlier φ holds. The formula $\varphi \mathrm{R}_{<k} \psi$ expresses that either for the next $k-1$ states ψ holds or in less than k steps, φ holds and always earlier ψ holds.
The derived basic temporal modalities are defined as follows: $\mathrm{F}_{<k} \varphi \stackrel{def}{=} true \mathrm{U}_{<k} \varphi$ and $\mathrm{G}_{<k} \varphi \stackrel{def}{=} false \mathrm{R}_{<k} \varphi$.

4.3 Semantics of LTL^k_{-X}

We start with defining models over the world solutions, which are finite sequences of worlds.

Definition 9. *A* model *is a pair $M = (\rho, V_\rho)$, where $\rho = (w_0, w_1, \ldots, w_n)$ is a world solution with $w_i = (O_i, v_i)$ for $0 \leq i \leq n$, and $V_\rho : \bigcup_{i=0}^{n} \{w_i\} \times \mathbb{ST} \longrightarrow \mathbb{N} \cup \{\infty\}$ is the function over the worlds of ρ valuating the expressions of the form stamp.level, defined as follows:*

- $V_\rho(w_i, s.level) = \infty$ *if* $s \notin O_i$,
- $V_\rho(w_i, s.level) = 0$ *if* $s \in O_0$,
- $V_\rho(w_i, s.level) = j$ *if* $s \in O_j$ *and* $s \notin O_{j-1}$, *for some* $1 \leq j \leq i$.

The intuition behind the definition of V_ρ is as follows. If a stamp s is not an element of a world w, then the value of $s.level$ in w does not exist, and this is denoted by ∞. If a stamp s is an element of the world w_0, then the value of $s.level$ is 0 in all the worlds. If w_j is the world, where s appears for the first time, then the value of $s.level$ is equal to j in w_j as well as in all further worlds.

Before defining the semantics of LTL^k_{-X} we extend the stamp.level valuation function V_ρ from $\mathbb{ST}$ to the level expressions as follows:

- $V_\rho(w_i, c) = c$,
- $V_\rho(w_i, \mathbf{lexp} \oplus \mathbf{lexp}') = V_\rho(w_i, \mathbf{lexp}) \oplus V_\rho(w_i, \mathbf{lexp}')$ if $V_\rho(w_i, \mathbf{lexp}) \neq \infty \neq V_\rho(w_i, \mathbf{lexp}')$,
- $V_\rho(w_i, \mathbf{lexp} \oplus \mathbf{lexp}') = \infty$ if $V_\rho(w_i, \mathbf{lexp}) = \infty$ or $V_\rho(w_i, \mathbf{lexp}') = \infty$,

We say that an LTL^k_{-X} formula φ is true in $M = (\rho, V_\rho)$ (in symbols $M \models \varphi$) iff $w_0 \models \varphi$, where for $m \leq n$ we have:

- $w_m \models \mathbf{pEx}(o)$ iff $o \in O_m$.

- $w_m \models \mathbf{pSet}(o.a)$ iff $o \in O_m$ and $v_m(o,a) = true$,
- $w_m \models \mathbf{pNull}(o.a)$ iff $o \in O_m$ and $v_m(o,a) = false$,
- $w_m \models \neg p$ iff $w_m \not\models p$, for $p \in PV$,
- $w_m \models (\mathbf{lexp} \sim \mathbf{lexp}')$ iff $V_\rho(w(m), \mathbf{lexp}) \sim V_\rho(w_m, \mathbf{lexp}')$ and $V_\rho(w_m, \mathbf{lexp}) \neq \infty \neq V_\rho(w_m, \mathbf{lexp}')$,
- $w_m \models \neg \mathbf{lc}$ iff $w_m \not\models \mathbf{lc}$, for $\mathbf{lc} \in \mathbf{LC}$,
- $w_m \models \varphi \wedge \psi$ iff $w_m \models \varphi$ and $w_m \models \psi$,
- $w_m \models \varphi \vee \psi$ iff $w_m \models \varphi$ or $w_m \models \psi$,
- $w_m \models \varphi \mathrm{U}_{<k} \psi$ iff $(\exists_{min(m+k,n)>l\geq m})(w_l \models \psi$ and $(\forall_{m\leq j<l}) w_j \models \varphi)$,
- $w_m \models \varphi \mathrm{R}_{<k} \psi$ iff $(\forall_{min(m+k,n)>l\geq m})\ w_l \models \psi)$ or $(\exists_{min(m+k,n)>l\geq m})(w_l \models \varphi$ and $(\forall_{m\leq j\leq l}) w_j \models \psi)$.

The semantics of the propositions follows their definitions, for the level constraints the semantics is based on the valuation function V_ρ, whereas for the temporal operators the semantics is quite standard. Note that we interpret our language over finite sequences as the solutions we are dealing with are finite.

Now, by a *temporal query* we mean a query (as defined in the former section) extended with an LTL^k_{-X} formula φ. The temporal solutions are these solutions for which the world sequences satisfy φ. A *temporal plan* is an equivalence class of the temporal solutions, defined over the same multiset of services.

5 Example of Temporal Abstract Planning

This section contains an example showing how the abstract temporal planning can be used in practice for a given ontology and user (temporal) queries.

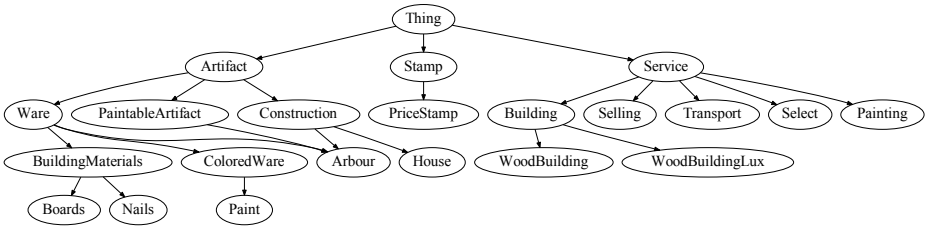


Fig. 2. Example ontology

Consider the ontology depicted in Fig. 2. In the $Artifact$ branch one can see several types of objects, like, e.g., *Arbour* (the main point of interest of this example), which is a subclass of $Ware$, $PaintableArtifact$, and $Construction$. At the left hand side the $Service$ branch and its subclasses are located. The service $Select$ (St) is able to search any $Ware$, $Selling$ (Sg) allows to purchase it, while $Transport$ (T) can be used to change its location. The $Painting$ (P) service is able to change colour of any $PaintableArtifact$, but it needs to use some $Paint$. The $Building$ (B) service can be used to obtain some $Construction$, but

it needs $BuildingMaterials$. Finally, two subclasses of $Building$ are specialized in production of wooden constructions using the supplied boards and nails. The services $WoodBuilding$ (Wb) and $WoodBuildingLux$ (Wbx) are similar, but the latter also paints the product to the chosen colour using their own paint, however for a higher price.

Assume the user wants to get a wooden arbour painted in blue. He formulates the query as follows: $in = inout = \emptyset$, $pre = true$, $out = \{Arbour\ a\}$, $post = (a.colour = blue\ \wedge\ a.owner = Me\ \wedge\ a.location = MyAddress)$. The $post$ formula is automatically translated to its abstract form, that is $(isSet(a.colour) \wedge isSet(a.owner) \wedge isSet(a.location))$. The shortest plans are $[St + Sg]$ and $[St + Sg + T]$. The former satisfies the user query only if the $Selling$ service is located in a close proximity of the user's address.

Assume that during the next planning steps (i.e., the offer collecting and the concrete planning) those plans turn out to have no realization acceptable by the user. Perhaps, there are no blue arbours in nearby shops or they are too expensive. Then, the alternative plan is to buy and transport an arbour in any colour, as well as some blue paint, and then use the $Painting$ service: $[2St+2Sg+2T+P]$, where one triple of services (St, Sg, T) provides the arbour, and the other a blue paint.

However, it could be the case that, e.g., the transport price of such a big object like an arbour exceeds the budget. If so, the possible solution is to buy boards, nails, and paint, transport them to the destination address, then to assembly the components with an appropriate building service, and paint, finally. This scenario is covered, for example, by the following plan: $[3St + 3Sg + 3T + Wb + P]$, where the triples of services (St, Sg, T) provide and transport boards, nails, and the paint.

Although, there are over eight hundred abstract plans of length from 2 to 11 satisfying the above user query, including these with multiple transportations of the arbour, or painting it several times. In order to restrict the plans to more specific ones, the user can refine the query demanding of specific types of services to be present in the plan using stamps. Thus, for example, by adding the following set of stamps to out: $\{Stamp\ t_1, Stamp\ t_2, Stamp\ t_3\}$ and extending $post$ by: $\bigwedge_{i=1..3}(t_i.serviceClass\ instanceOf\ Transport)$, the number of possible abstract plans (of length from 2 to 11) can be reduced below two hundred. Then, if instead of buying a final product the user wants to buy and transport the components, in order to build and paint the arbour, he can add two more stamps and conditions to the query. That is, by adding to out the set $\{Stamp\ b, Stamp\ p\}$, and by extending $post$ by the expression $(\wedge\ b.serviceClass\ instanceOf\ Building \wedge\ t_3.serviceClass\ instanceOf\ Painting)$, one can reduce the number of resulting plans to 2 only: $[3St+3Sg+3T+Wb+P]$ and $[3St+3Sg+3T+Wbx+P]$.

However, even 2 abstract plans only can be realized in a number of different ways, due to possible many transformation contexts, and the number of different partial orders represented by a single abstract plan. If the user wants to further reduce the number of possible plan realizations by interfering with an order of services, he should specify some temporal restrictions. For example, if the user

wants to ensure that all the transports are executed before the building starts, he can express it as a formula:

$$\varphi_1 = \mathrm{F}\big((b.level > t_1.level) \wedge (b.level > t_2.level) \wedge (b.level > t_3.level)\big)$$

Moreover, if the intention of the user is to proceed with some service directly after another one, for example, to start building just after the third transport, one can express such a constraint as:

$$\varphi_2 = \mathrm{F}(b.level = t_3.level + 1)$$

Moreover, using a temporal query the user can prevent some services from occurring in the plan. For example, using the following formula:

$$\varphi_3 = \neg\mathbf{pEx}(a)\ \mathrm{U}\ \mathbf{pNull}(a.colour),$$

which means that just after the arbour has been produced, its colour is not set, the user excludes the $WoodBuildingLux$ service (which builds and paints the arbour).

The other possibility of extending the user query by a temporal component includes using the k-restricted versions of modal operators. For example, consider the following formula:

$$\varphi_4 = \mathrm{F}_{<10}(\mathbf{pEx}(t_1) \wedge \mathbf{pEx}(t_2) \wedge \mathbf{pEx}(t_3)),$$

which states that three transportations should be executed in the first nine steps of the plan.

6 Implementation, Experiments, and Conclusions

In this section we sketch the implementation of the propositions and the level constraints, and then we evaluate the efficiency of our tool using several scalable benchmarks.

6.1 Implementation

The implementation of the propositions and the level constraints exploits our symbolic representation of world sequences. The objects and the worlds are represented by sets of *variables*, which are first allocated in the memory of an SMT-solver, and then used to build formulas mentioned in Section 4. The representation of an object is called a *symbolic object*. It consists of an integer variable representing the type of an object, called a *type variable*, and a number of Boolean variables to represent the object attributes, called the *attribute variables*. In order to represent all types and identifiers as numbers, we introduce a function $num : \mathrm{A} \cup \mathbb{P} \cup \mathbb{S} \cup \mathbb{O} \longmapsto \mathbb{N}$, which with every attribute, object type, service type, and object assigns a natural number.

A *symbolic world* consists of a number of symbolic objects. Each symbolic world is indexed by a natural number from 0 to n. Formally, the i-th symbolic object from the j-th symbolic world is a tuple: $\mathbf{o}_{i,j} = (\mathbf{t}_{i,j}, \mathbf{a}_{i,0,j}, ..., \mathbf{a}_{i,max_{at}-1,j})$, where $\mathbf{t}_{i,j}$ is the type variable, $\mathbf{a}_{i,x,j}$ is the attribute variable for $0 \leq x < max_{at}$, where max_{at} is the maximal number of the attribute variables needed to represent the object.

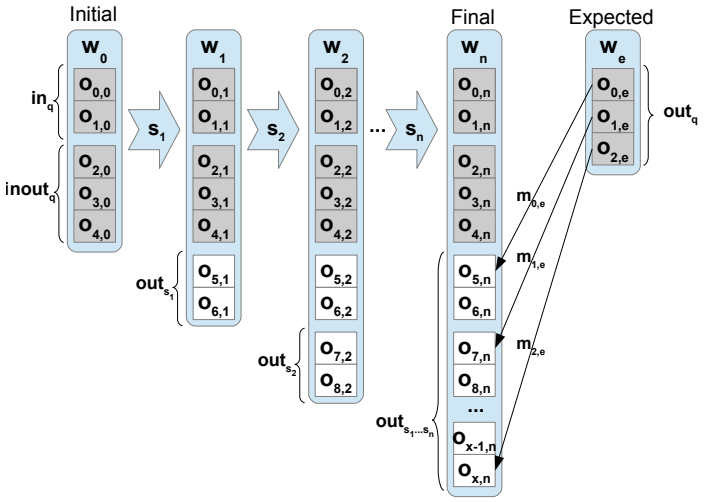


Fig. 3. Symbolic worlds of a transformation sequence

Note that actually a symbolic world represents *a set of worlds*, and only a *valuation* of its variables makes it a single world. The j-th symbolic world is denoted by $\mathbf{w}_j$, while the number of the symbolic objects in $\mathbf{w}_j$ - by $|\mathbf{w}_j|$. Note that the set of the initial worlds of the query q (W^q_{init}) is represented by a symbolic world $\mathbf{w}_0$. Fig. 3 shows subsequent symbolic worlds of a transformation sequence.

One of the important features of our encoding is that for a given index of a symbolic object i we are able to determine the step of a solution, in which the object was produced. This is done by the function $lev_q : \mathbb{N} \mapsto \mathbb{N}$, such that for a given query q:

$$lev_q(i) = \begin{cases} 0 \text{ for } i < |\mathbf{w}_0| \\ \lfloor \frac{(i-|\mathbf{w}_0|)}{max_{out}} \rfloor + 1 \text{ for } i \geq |\mathbf{w}_0| \end{cases} \quad (2)$$

where max_{out} is the maximal number of the objects produced by a single service.

Another important feature of our encoding is that the objects of out_q need to be identified among the objects of the symbolic world $\mathbf{w}_n$ (of indices greater than $|\mathbf{w}_0|$). To this aim, we allocate a new symbolic world $\mathbf{w}_e$ (with $e = n + 1$), containing all the objects from out_q. Note that the world $\mathbf{w}_e$ is not a part of a

world solution, but it provides a set of additional, helper variables. Finally, we need a mapping between the objects from a final world $\mathbf{w}_n$ produced during the subsequent transformations and the objects from $\mathbf{w}_e$. To this aim we allocate p additional *mapping variables* in the symbolic world $\mathbf{w}_e$, where $p = |out_q|$. These variables, denoted by $\mathbf{m}_{0,e}, \ldots, \mathbf{m}_{p-1,e}$, are intended to store the indices of the objects from a final world, which are compatible with the objects encoded over $\mathbf{w}_e$. Thus, we encode the state of the expected worlds of the query q (W^q_{exp}), imposed by $post_q$, using two sets of symbolic objects. The objects of $in_q \cup inout_q$ are encoded directly over the (final) symbolic world $\mathbf{w}_n$. The state of the objects from out_q are encoded over $\mathbf{w}_e$, and since their indices are not known, all possible mappings between objects from $\mathbf{w}_e$ and $\mathbf{w}_n$ are considered, by encoding a disjunction of equalities between objects from $\mathbf{w}_e$ and $\mathbf{w}_n$. See [18] for more details.

The translation of the propositions defined over the objects and their attributes of a user query q in a symbolic world $\mathbf{w}_m$ ($0 \leq m \leq n$) is as follows:

$$[\mathbf{pEx}(o)]^m = \begin{cases} true, \text{ for } o \in in_q \cup inout_q, \\ false, \text{ for } o \in out_q, m = 0, \\ lev_q(\mathbf{m}_{num(o),e}) \leq m, \text{for } o \in out_q, m > 0. \end{cases} \tag{3}$$

That is, the objects from the initial world exist in all the subsequent worlds, the objects from the *out* set do not exist in the world $\mathbf{w}_0$, and they appear in some subsequent world. Then, since the index of the object o is stored as the value of corresponding mapping variable $\mathbf{m}_{num(o),e}$, we can determine if it exists in the world $\mathbf{w}_m$ using the lev_q function.

The proposition $\mathbf{pSet}(o.a)$ is encoded over the symbolic world $\mathbf{w}_m$ as:

$$[\mathbf{pSet}(o.a)]^m = [\mathbf{pEx}(o)]^m \wedge \begin{cases} \mathbf{a}_{j,x,m}, \text{for } o \in in_q \cup out_q, \\ \bigvee_{i=|\mathbf{w}_0|}^{|\mathbf{w}_m|-1} (\mathbf{m}_{j,e} = i \wedge \mathbf{a}_{i,x,m}), \text{for } o \in out_q \end{cases} \tag{4}$$

where $j = num(o)$ and $x = num(a)$.

It follows from our symbolic representation that the indices of objects from an initial world are known, and we can get the value of the appropriate attribute variable directly. However, in the case of objects from out_q we have to consider all possible mappings between objects from $\mathbf{w}_e$ and $\mathbf{w}_m$. Note that the encoding of the proposition $\mathbf{pNull}(o.a)$ over the symbolic world $\mathbf{w}_m$ (i.e., $[\mathbf{pNull}(o.a)]^m$) is very similar. The only change is the negation of $\mathbf{a}_{i,x,m}$ in the above formula.

In order to encode the level constraints, we introduce a set of the special *level variables*. That is, for every stamp s used in some level constraint we introduce to the world $\mathbf{w}_e$ an additional integer variable $\mathbf{l}_{i,e}$, where $i = num(s)$, intended to store the level value of the stamp s. The level value is assigned to $\mathbf{l}_{i,e}$ using the following formula $[bind(i)] := (\mathbf{l}_{i,e} = lev_q(\mathbf{m}_{i,e}))$ for $i = num(s)$, where q is a user query. Then, for every stamp s used in a level constraint we add the corresponding $[bind(num(s))]$ formula as an SMT assertion. Thus, the encoding of the level constraints is as follows:

$$[\mathbf{lexp}] = \begin{cases} c \text{ for } \mathbf{lexp} = c \\ \mathbf{l}_{i,e}, \text{for } \mathbf{lexp} = s.level, i = num(s) \\ [\mathbf{lexp}'] \oplus [\mathbf{lexp}''] \text{ for } \mathbf{lexp} = \mathbf{lexp}' \oplus \mathbf{lexp}'' \end{cases} \quad (5)$$

The encoding of arithmetic operators is straightforward, since they are supported by theories built in SMT-solvers, like, e.g., Linear Integer Arithmetic or Bitvector theory. In what follows, $[\varphi]_n^m$ denotes the translation of the formula φ at the state w_m of the world sequence of length $n+1$.

Definition 10 (Translation of the LTL_{-X}^k formulae to SMT). *Let φ be an LTL_{-X}^k formula, $(\mathbf{w}_0, \ldots, \mathbf{w}_n)$ be a sequence of symbolic worlds, and $0 \leq m \leq n$.*

- $[p]_n^m := [p]^m$, *for* $p \in PV$,
- $[\neg p]_n^m := \neg[p]^m$, *for* $p \in PV$,
- $[\mathbf{lexp}' \sim \mathbf{lexp}'']_n^m := [\mathbf{lexp}'] \sim [\mathbf{lexp}''] \bigwedge_{s \in st(\mathbf{lexp}')} [\mathbf{pEx}(s)]^m \bigwedge_{s \in st(\mathbf{lexp}'')} [\mathbf{pEx}(s)]^m$,
- $[\neg \mathbf{lc}]_n^m := \neg[\mathbf{lc}]_n^m$, *for* $\mathbf{lc} \in \mathbf{LC}$,
- $[\varphi \wedge \psi]_n^m := [\varphi]_n^m \wedge [\psi]_n^m$,
- $[\varphi \vee \psi]_n^m := [\varphi]_n^m \vee [\psi]_n^m$,
- $[\varphi \mathrm{U}_{<k} \psi]_n^m := \bigvee_{i=m}^{min(m+k,n)} ([\psi]_n^i \wedge \bigwedge_{j=m}^{i-1} [\varphi]_n^j)$,
- $[\varphi \mathrm{R}_{<k} \psi]_n^m := \bigwedge_{i=m}^{min(m+k,n)} [\psi]_n^i \vee \bigvee_{i=m}^{min(m+k,n)} ([\varphi]_n^i \wedge \bigwedge_{j=m}^{i} [\psi]_n^j)$,

where $st(\mathbf{lexp})$ returns the set of the stamps over which the expression **lexp** *is built.*

Theorem 1. *The encoding of the temporal query is correct.*

Proof. This can be shown by induction on the length of a formula. Omitted here because of lack of space[3].

6.2 Experimental Results

In order to evaluate the efficiency of our approach we performed several experiments using standard PC with 2GHz CPU and 8GB RAM, and Z3 [7] version 4.3 as an SMT-solver. The results are summarized in Table 1. Using our Ontology Generator (OG) we generated 15 ontologies, each of them consisting of 150 object types and from 64 to 256 service types (the column named **n** of Tab. 1). For each ontology a query has been generated in such a way that it is satisfied by exactly 10 plans of length from 6 to 18 (the parameter **k**). The queries demand at least two objects to be produced, and impose restrictions on (abstract) values of some of their attributes. A generated query example is as follows: $in = \{Rjlbp\ rjlbp1\}$, $inout = \{Bozwd\ bozwd1\}$, $out = \{Opufo\ opufo1, Ehxjb\ ehxjb2\}$, $pre = isSet(bozwd1.avg) \wedge isSet(rjlbp1.ppw)$,

[3] The full version of this paper is available at http://artur.ii.uph.edu.pl/pnse14ltl.pdf.

$post = isSet(opufo1.epv) \wedge isNull(bozwd1.dyn) \wedge isSet(ehxjb2.zdv) \wedge isNull(ehxjb2.rxz) \wedge isSet(bozwd1.fsl)$.

First, we ran our planner for each ontology and each query instance without a temporal query (column ψ_1), in order to collect statistics concerning the time needed to find the first plan ($\mathbf{P}_1$), all 10 plans ($\mathbf{P}_{10}$), as well as the total time (column **T**) and the memory consumed (**M**) by the SMT-solver in order to find all the plans and checking that no more plans of length k exist. We imposed the time limit of 1000 seconds for the SMT-solver. Each time-out is reported in the table by *TO*. It is easy to observe that during these experiments as many as 9 instances ran out of time.

Table 1. Experimental results

		$\psi_1 = true$				ψ_3				ψ_4				ψ_5			
n	**k**	$\mathbf{P}_1$ [s]	$\mathbf{P}_{10}$ [s]	**T** [s]	**M** [MB]	$\mathbf{P}_1$ [s]	$\mathbf{P}_{10}$ [s]	**T** [s]	**M** [MB]	$\mathbf{P}_1$ [s]	$\mathbf{P}_{10}$ [s]	**T** [s]	**M** [MB]	$\mathbf{P}_1$ [s]	$\mathbf{P}_{10}$ [s]	**T** [s]	**M** [MB]
64	**6**	7.23	9.73	19.5	19.7	2.34	7.17	8.49	14.1	2.19	4.14	4.21	11.6	3.51	4.7	6.48	10.8
	9	23.5	53.1	177	173	20.1	34.5	47.0	38.3	15.9	18.8	19.2	22.3	14.8	40.1	44.7	36.0
	12	165	479	TO	-	101	216	354	117	60.5	85.2	87.6	59.3	95.1	122	127	68.1
	15	305	TO	TO	-	329	762	TO	-	119	216	241	105	195	345	351	129
	18	TO	TO	TO	-	TO	TO	TO	-	461	TO	TO	-	604	TO	TO	-
128	**6**	16.0	28.1	55.7	42.7	9.75	19.8	22.6	21.3	7.68	10.1	10.2	16.8	10.9	12.7	15.1	16.3
	9	53.1	94.9	270	250	55.5	98.2	104	44.7	21.8	34.0	34.2	31.0	38.1	54.7	62.6	44.9
	12	136	677	TO	-	134	428	474	96.3	76.8	99.9	102	61.9	93.1	114	117	47.9
	15	TO	TO	TO	-	456	780	TO	-	116	199	202	86	183	258	263	79.8
	18	TO	TO	TO	-	TO	TO	TO	-	381	556	573	143	383	708	714	130
256	**6**	16.1	30.6	41.5	35.4	20.3	25.7	30.0	26.1	11.1	13.9	14.1	21.7	14.9	18.3	21	21.6
	9	84.4	137	374	466	63.7	99.3	131	53.4	26.2	38.4	39.1	37.5	88.5	119	156	74.9
	12	267	TO	TO	-	242	584	692	112	114	181	183	80.6	250	315	321	73.4
	15	685	TO	TO	-	562	TO	TO	-	198	304	309	86.8	472	582	592	120
	18	TO	TO	TO	-	TO	TO	TO	-	574	919	937	137	942	TO	TO	-

The next experiments involve temporal queries using level constraints. To this aim we extended the *out* set of the generated queries by the appropriate stamp set. Moreover, the *post* formulas of the queries have been also extended with the expression: $\bigwedge_{i=1}^{\lfloor \frac{k}{2} \rfloor}(s_i.serviceClass\ instanceOf\ C_i)$, where $s_i \in \mathbb{ST}$, while $C_i \in \mathbb{S}$ are service types occurring in the solutions generated by OG.

Our second group of experiments involved the temporal formula ψ_2:

$$\psi_2 = \mathrm{F}(\bigwedge_{i=1}^{\lfloor \frac{k}{2} \rfloor - 1} s_i.level < s_{i+1}.level),$$

which expresses that about a half of the stamps being effects of the solution execution, should be produced in the given order. We do not present detailed

results, because they are in general comparable with the performance in the former experiments. Similarly, there are 9 time-outs, but the time and the memory consumption varies a bit - for some cases the results are slightly better, while for others are a little worse.

In the third group of the experiments we imposed stronger restrictions on the possible service orders of the solutions using the following formula:

$$\psi_3 = \mathrm{F}(\bigwedge_{i=1}^{\lfloor \frac{k}{2} \rfloor} s_i.level < i + 2).$$

This formula still leaves a certain degree of freedom in a service ordering, however its encoding as an SMT-instance is more compact, since the constant values are introduced in place of some level variables. Thus, probably, it is also easier to solve. The results are summarized in the column ψ_3 in Table 1. It is easy to observe that the time and the memory consumption is significantly lower. Moreover, the number of time-outs dropped to 6. Thus, this is an example showing an improvement in the planning efficiency using a temporal query.

Our next experiment involves the formula ψ_4 specifying the strict ordering of several services in the solution using stamp-based level constraints:

$$\psi_4 = \mathrm{F}(\bigwedge_{i=1}^{\lfloor \frac{k}{2} \rfloor} s_i.level = i).$$

The analysis of the results (given in the column ψ_4 of Table 1) indicates a dramatic improvement of our planner efficiency, in terms of time and memory consumption by the SMT-solver. Moreover, in this experiments group the planner has been able to terminate its computations in the given time limit for all but one instances.

Finally, we want to confront the planner behaviour with other kind of temporal formulae. Using the *Until* modality, we demand that one of the objects from out_q has to be produced no later than in the middle of the plan. Moreover, knowing the structure of the generated queries, we impose that after the object appears one of its attributes should already be set. This is expressed by the following formula:

$$\psi_5 = \neg\mathbf{pEx}(o)\ \mathrm{U}_{<\lceil \frac{k}{2} \rceil}\ \mathbf{pSet}(o.a)$$

where $o \in out_q$, $a \in attr(o)$, and the expression $isSet(o.a)$ is not contradictory with $post_q$. The results has been summarised in column ψ_5 of Table 1. It is easy to observe that also this time the plans have been found faster and using less memory than in the case when no temporal formula is involved.

7 Conclusions

In this paper we have applied the logic LTL^k_{-X} to specifying temporal queries for temporal planning within our tool PlanICS. This is a quite natural extension

of our web service composition system, in which the user gets an opportunity to specify more requirements on plans. These requirements are not only declarative any more. The overall conclusion is that the more restrictive temporal query, the more efficient planning, given the same ontology. Assuming that the more restrictive temporal queries, the longer formulas expressing them, the above conclusion shows a difference with model checking, where the complexity depends exponentially on the length of an LTL^k_{-X} formula.

Our temporal planner is the first step towards giving the user even more freedom by defining a so-called parametric approach. We aim at having a planner which in addition to the current capabilities, could also suggest what extensions to the ontology or services should be made in order to get better or unrealizable plans so far. This is going to be a subject of our next paper.

References

1. S. Ambroszkiewicz. Entish: A language for describing data processing in open distributed systems. *Fundam. Inform.*, 60(1-4):41–66, 2004.
2. M. Bell. *Introduction to Service-Oriented Modeling.* John Wiley & Sons, 2008.
3. J. Bentahar, H. Yahyaoui, M. Kova, and Z. Maamar. Symbolic model checking composite web services using operational and control behaviors. *Expert Systems with Applications*, 40(2):508 – 522, 2013.
4. L. Bentakouk, P. Poizat, and F. Zaidi. Checking the behavioral conformance of web services with symbolic testing and an SMT solver. In *Tests and Proofs*, volume 6706 of *LNCS*, pages 33–50. Springer, 2011.
5. M. M. Bersani, L. Cavallaro, A. Frigeri, M. Pradella, and M. Rossi. SMT-based verification of LTL specification with integer constraints and its application to runtime checking of service substitutability. In *SEFM*, pages 244–254, 2010.
6. V. Chifu, I. Salomie, and E. St. Chifu. Fluent calculus-based web service composition - from OWL-S to fluent calculus. In *Proc. of the 4th Int. Conf. on Intelligent Computer Communication and Processing*, pages 161 –168, 2008.
7. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer-Verlag, 2008.
8. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, and J. Skaruz. HarmonICS - a tool for composing medical services. In *ZEUS*, pages 25–33, 2012.
9. D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, and A. Zbrzezny. PlanICS - a web service compositon toolset. *Fundam. Inform.*, 112(1):47–71, 2011.
10. M. Elwakil, Z. Yang, L. Wang, and Q. Chen. Message race detection for web services by an SMT-based analysis. In *Proc. of the 7th Int. Conference on Autonomic and Trusted Computing*, ATC'10, pages 182–194. Springer, 2010.
11. V. Gehlot and K. Edupuganti. Use of colored Petri nets to model, analyze, and evaluate service composition and orchestration. In *System Sciences, 2009. HICSS '09.*, pages 1 –8, jan. 2009.
12. A. E. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5–6):619 – 668, 2009. Advances in Automated Plan Generation.

13. S. Hao and L. Zhang. Dynamic web services composition based on linear temporal logic. In *Information Science and Management Engineering (ISME), 2010 International Conference of*, volume 1, pages 362–365, Aug 2010.
14. Z. Li, L. O'Brien, J. Keung, and X. Xu. Effort-oriented classification matrix of web service composition. In *Proc. of the Fifth International Conference on Internet and Web Applications and Services*, pages 357–362, 2010.
15. G. Monakova, O. Kopp, F. Leymann, S. Moser, and K. Schäfers. Verifying business rules using an SMT solver for BPEL processes. In *BPSC*, pages 81–94, 2009.
16. W. Nam, H. Kil, and D. Lee. Type-aware web service composition using boolean satisfiability solver. In *Proc. of the CEC'08 and EEE'08*, pages 331–334, 2008.
17. A. Niewiadomski and W. Penczek. Towards SMT-based Abstract Planning in PlanICS Ontology. In *Proc. of KEOD 2013 – International Conference on Knowledge Engineering and Ontology Development*, pages 123–131, September 2013.
18. A. Niewiadomski, W. Penczek, and A. Półrola. Abstract Planning in PlanICS Ontology. An SMT-based Approach. Technical Report 1027, ICS PAS, 2012.
19. OWL 2 web ontology language document overview. http://www.w3.org/TR/owl2-overwiew/, 2009.
20. J. Rao, P. Küngas, and M. Matskin. Composition of semantic web services using linear logic theorem proving. *Inf. Syst.*, 31(4):340–360, June 2006.
21. J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of SWSWPC'04*, volume 3387 of *LNCS*, pages 43–54. Springer, 2004.
22. J. Skaruz, A. Niewiadomski, and W. Penczek. Automated abstract planning with use of genetic algorithms. In *GECCO (Companion)*, pages 129–130, 2013.
23. P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. In *The Semantic Web – ISWC 2004*, volume 3298 of *LNCS*, pages 380–394. 2004.

Kleene Theorems for Labelled Free Choice Nets

Ramchandra Phawade and Kamal Lodaya

The Institute of Mathematical Sciences, CIT Campus, Chennai *600113*, India

Abstract. In earlier work [LMP11], we showed that a graph-theoretic condition called "structural cyclicity" enables us to extract syntax from a conflict-equivalent product system of automata. In this paper we have a "pairing" property in our syntax which allows us to connect to a broader class of product systems, where the conflict-equivalence is not statically fixed. These systems have been related to labelled free choice nets.

1 Introduction

Petri nets are an excellent visual representation of concurrency. But like any graphical notation they are less amenable to syntax. For finite automata, Kleene's regular expressions provide us with a formalism where we can switch between the graphical and the textual. For 1-bounded Petri nets, equivalent syntax has been provided by Grabowski [Gra81], Garg and Ragunath [GR92] and other authors. Here we place restrictions on this syntax in an effort to match the 1-bounded labelled free choice nets, a very well-studied subclass [Hac72] with more efficient analysis and algorithms [DE95]. It has been claimed that free choice nets can be useful in business process modelling [SH96], but our motivation is more conceptual than dictated by business concerns.

As is usual when dealing with subclasses, this turns out to be challenging. We also follow the example of finite automata and work directly with labelled nets, not relying on a renaming operator in the syntax. As in our earlier paper [LMP11], we rely on an intermediate formalism, "direct" products of automata, which are known to be weaker than 1-bounded nets [Zie87,Muk11]. There we identified a subclass called FC-products, and a graph-theoretic property called "structural cyclicity", for which we presented an equivalent syntax which was restricted to being without nested Kleene star operators.

The improvement in this paper is that on the system side we have an enlarged subclass called **FC-matching products**. On the syntax side we drop the structural cyclicity condition and do not place any restriction on the Kleene stars, thus (unlike in our earlier paper) including all regular expressions. We do have global restrictions. A "pairing" condition identifies synchronizations which will take place at run-time. Assuming a communication alphabet $\{a, b, c\}$, the expression $(a + a + b)(a + c + c)$ the a's in the two groups of parentheses will be paired into different synchronizations. Correspondingly we have a "matching" condition in the product systems. The matching condition produces free choice nets (and the converse also holds). Our proofs go through a subclass where communications are labelled with the place from which they are issued.

2 Preliminaries

Let Σ be a finite alphabet and Σ^* be the set of all words over alphabet Σ, including the empty word ε. A language over an alphabet Σ is a subset $L \subseteq \Sigma^*$. The projection of a word $w \in \Sigma^*$ to a set $\Delta \subseteq \Sigma$, denoted as $w{\downarrow_\Delta}$, is defined by:

$$\varepsilon{\downarrow_\Delta} = \varepsilon \text{ and } (a\sigma){\downarrow_\Delta} = \begin{cases} a(\sigma{\downarrow_\Delta}) & \text{if } a \in \Delta, \\ \sigma{\downarrow_\Delta} & \text{if } a \notin \Delta. \end{cases}$$

Definition 1. *Let Loc denote the set $\{1, 2, \ldots, k\}$. A distribution of Σ over Loc is a tuple of nonempty sets $(\Sigma_1, \Sigma_2, \ldots, \Sigma_k)$ with $\Sigma = \bigcup_{1 \le i \le k} \Sigma_i$. For each action $a \in \Sigma$, its locations are the set $loc(a) = \{i \mid a \in \Sigma_i\}$. Actions $a \in \Sigma$ such that $|loc(a)| = 1$ are called local, otherwise they are called global.*

A regular expression over alphabet Σ_i defining a nonempty language is given by:

$$s ::= a \in \Sigma_i | s_1 \cdot s_2 | s_1 + s_2 | s_1^*$$

As a measure of the size of an expression we will use $wd(s)$ for its alphabetic width—the total number of occurrences of letters of Σ in s. We will use syntactic entities associated with regular expressions which are known since the time of Brzozowski [Brz64], Mirkin [Mir66] and Antimirov [Ant96].

For each regular expression s over Σ_i, its initial actions form the set $Init(s) = \{a \mid av \in Lang(s) \text{ and } v \in \Sigma_i^*\}$ which can be defined syntactically. Similarly, we can syntactically check whether the empty word $\varepsilon \in Lang(s)$. Next we syntactically define derivatives [Ant96].

Definition 2. *Given regular expression s and symbol a, the partial derivatives of s wrt a, written $Der_a(s)$ are defined as follows.*

$$\begin{aligned} Der_a(b) &= \emptyset \text{ if } a \neq b \\ Der_a(a) &= \{\varepsilon\} \\ Der_a(s_1 + s_2) &= Der_a(s_1) \cup Der_a(s_2) \\ Der_a(s_1^*) &= Der_a(s_1) \cdot s_1^* \\ Der_a(s_1 \cdot s_2) &= \begin{cases} Der_a(s_1) \cdot s_2 \cup Der_a(s_2) & \text{if } \varepsilon \in Lang(s_1) \\ Der_a(s_1) \cdot s_2 & \text{otherwise} \end{cases} \end{aligned}$$

Inductively $Der_{aw}(s) = Der_w(Der_a(s))$.

The set of all derivatives $Der(s) = \bigcup_{w \in \Sigma_i^*} Der_w(s)$.

We have the Antimirov derivatives $Der_a(ab + ac) = \{b, c\}$ and $Der_a(a(b + c)) = \{b + c\}$, whereas the Brzozowski a-derivative [Brz64] (which is used for constructing deterministic automata, but which we do not use in this paper) for both expressions would be $\{b + c\}$.

A derivative d of s with global $a \in Init(d)$ is called an **a-site** of s. An expression is said to have **equal choice** if for all a, its a-sites have the same set of initial actions. For a set D of derivatives, we collect all initial actions to form $Init(D)$. We syntactically partition the a-sites of s, each set of the partition containing those coming from a common source derivative, as follows.

Definition 3. *For partitions X_1, X_2 with blocks D_1, D_2 containing elements d_1, d_2 respectively, we use the notation $(X_1 \cup X_2)[d/d_1, d_2]$ for the modified partition $((X_1 \cup X_2) \setminus \{D_1, D_2\}) \cup \{(D_1 \cup D_2 \cup \{d\}) \setminus \{d_1, d_2\}\}$.*

$$Part_a(b) = \emptyset \text{ if } a \neq b$$
$$Part_a(a) = \{\{a\}\}$$
$$Part_a(s_1 + s_2) = \begin{cases} (Part_a(s_1) \cup Part_a(s_2))[s_1 + s_2/s_1, s_2] & \text{if } a \in Init(s_1 + s_2) \\ Part_a(s_1) \cup Part_a(s_2) & \text{otherwise} \end{cases}$$
$$Part_a(s_1^*) = \begin{cases} Part_a(s_1)[s_1^*/s_1] & \text{if } a \in Init(s_1) \\ Part_a(s_1) \cdot s_1^* & \text{otherwise} \end{cases}$$
$$Part_a(s_1 \cdot s_2) = \begin{cases} Part_a(s_1)[s_1 \cdot s_2/s_1] \cup Part_a(s_2) & \text{if } \varepsilon \in Lang(s_1) \\ Part_a(s_1) \cdot s_2 \cup Part_a(s_2) & \text{otherwise} \end{cases}$$

The next definition and the following proposition identify the key property of this partition of a-sites for this paper.

Definition 4. *Given a set of derivatives D and an action a, define the prefixes $Pref_a^D(L) = \{x \mid xay \in L, \exists d \in Der_x(L) \cap D, \varepsilon \in Der_{ay}(d)\}$, suffixes $Suf_a^D(L) = \{y \mid xay \in L, x \in Pref_a^D(L)\}$, and the relativized language $L^D = \{xay \mid xay \in L, \exists d \in Der_x(L) \cap D, \varepsilon \in Der_{ay}(d)\}$. We say that the derivatives in set D **a-bifurcate** L if $L^D \cap \Sigma^* a \Sigma^* = Pref_a^D(L)\ a\ Suf_a^D(L)$. If D is the set of all derivatives, we say L is **a-bifurcated**.*

Proposition 1. *Every block D of the partition $Part_a(s)$ a-bifurcates $Lang(s)$.*

Proof. By induction on the definition. □

Consider a regular expression s in the context of a distribution $(\Sigma_1, \ldots, \Sigma_k)$, so that some of the actions are global. The following properties of expressions will be important in this paper, where the derivatives are taken for regular expressions and also for the connected expressions defined in the next section.

Definition 5. *If for all global actions a occurring in s, the partition $Part_a(s)$ consists of a single block, then we say s has **unique sites**. It has **deterministic global actions** if for every global action a and every a-site $d \in Der(s)$, $|Der_a(d)| = 1$. It has **unique global actions** if it has both these properties.*

3 Connected Expressions over a Distribution

We have a simple syntax of connected expressions. The s_i can be any regular expressions (of any star-height), which is different from our earlier paper [LMP11].

$$e ::= 0 | fsync(s_1, s_2, \ldots, s_k), s_i \text{ over } \Sigma_i$$

When $e = fsync(s_1, s_2, \ldots, s_k)$ and $I \subseteq \Sigma$, let the projection $e{\downarrow}I = \Pi_{i \in I} s_i$.

For the connected expression 0, we have $Lang(0) = \emptyset$. For the connected expression $e = fsync(s_1, s_2, \ldots, s_k)$, its language is given by

$$Lang(e) = Lang(s_1) \| Lang(s_2) \| \ldots \| Lang(s_k),$$

where the synchronized shuffle $L = L_1 \| \ldots \| L_k$ is defined by

$$w \in L \text{ iff } \text{ for all } i \in \{1, \ldots, k\}, w{\downarrow_{\Sigma_i}} \in L_i.$$

The definitions of derivatives can be easily extended to connected expressions. 0 has no derivatives on any action. Given $e = fsync(s_1, s_2, \ldots, s_k)$, its derivatives are defined by induction using the derivatives of the s_i on action a:

$$Der_a(e) = \{fsync(r_1, r_2, \ldots, r_k) \mid \forall i \in loc(a),\ r_i \in Der_a(s_i);\ \text{otherwise}\, r_j = s_j\}.$$

We will use the word derivative for expressions such as $d = fsync(r_1, r_2, \ldots, r_k)$ above (essentially tuples of derivatives of regular expressions), and $d[i]$ for r_i. The number of derivatives can be exponential in k. Define $Init(d)$ to be those actions a such that $Der_a(d)$ is nonempty. If $a \in Init(d)$ we call d an **a-site**. The reachable derivatives are $Der(e) = \{d \mid d \in Der_x(e), x \in \Sigma^*\}$. For example, $fsync(ab, ba)$ has derivatives other than the expression itself, but none of them is reachable.

3.1 Properties of Connected Expressions

We now define some properties of connected expressions over a distribution. These will ultimately lead us to construct free choice nets. All but the last property are PTIME-checkable. The last property requires PSPACE since it runs over all reachable derivatives.

Definition 6. *Let $e = fsync(s_1, s_2, \ldots, s_k)$ be a connected expression over Σ. For a global action a, an **a-pairing** is a subset of tuples $\Pi_{i \in loc(a)} Part_a(s_i)$, the projections of these tuples covering the a-sites in s_i, such that if a block of $Part_a(s_j), j \in loc(a)$ appears in one tuple of the pairing, it does not appear in another tuple. (For convenience we also write $pairing(a)$ as a subset of $\Pi_{i \in loc(a)} Der(s_i)$ which respects the partition.) We call $pairing(a)$ equal choice if for every tuple in the pairing, the derivatives in the tuple have equal choice.*
*We extend the definition to connected expressions. A derivative $fsync(r_1, \ldots, r_k)$ is in pairing(a) if there is a tuple $D \in pairing(a)$ such that $r_i \in D[i]$ for all $i \in loc(a)$. For convenience we may write a derivative as an element of $pairing(a)$. Expression e is said to have **(equal choice) pairing of actions** if for all global actions a, there exists an (equal choice) $pairing(a)$. Expression e is said to be **consistent with a pairing of actions** if every reachable a-site $d \in Der(e)$ is in $pairing(a)$.*

Example 1. Let $(\Sigma_1 = \{a\}, \Sigma_2 = \{a\})$. Expression $fsync(aa, a)$ does not have a *pairing*. The two a's on the left are in different blocks of the partition and they have to pair with one block on the right, which is not allowed.

Example 2. Let $(\Sigma_1 = \{a\}, \Sigma_2 = \{a, b, c, d, f\})$. In expression $e = fsync(aa, bad + caf)$ we have two blocks on the left and two blocks on the right, so we can have a pairing. But e cannot be consistent with any pairing.

Example 3. Let $(\Sigma_1 = \{a, c\}, \Sigma_2 = \{b, c\}), \Sigma_3 = \{a, b, c\})$. Consider this expression $fsync((ac)^*, (bc)^*, (a(b + c))^*)$. Individual regular expressions are $r_1 = (ac)^*$, $r_2 = (bc)^*$ and $r_3 = (a(b + c))^*$. Now we have $r_1' = Der_a(r_1) = c(ac)^*$ and $Init(r_1') = \{c\}$. For r_3 we have, $r_3' = Der_a(r_3) = (b + c)(a(b + c))^*$ and $Init(r_3') = \{b, c\}$. r_1' and r_3' do not have equal choice.

Proposition 2. *For a connected expression e checking existence of a pairing of actions and checking whether it is equal choice can be done in polynomial time, checking consistency with a pairing of actions is in* PSPACE.

Proof. We have to visit each derivative of all the regular expressions to construct the a-partitions for every a. We can record their initial actions. Maximum number of Antimirov derivatives of any regular expression s is at most $wd(s) + 1$ [Ant96]. There are k regular expressions in e. If the number of blocks in two a-partitions is not the same, there cannot be an a-pairing, otherwise there always exists an a-pairing. For an equal choice pairing, we have to count blocks whose sets of initial actions are the same, this can be done in cubic time.

On the other hand, to check consistency with a pairing of actions, we have to visit each reachable derivative, this can be done in PSPACE. □

4 Product Systems over a Distribution

Fix a distribution $(\Sigma_1, \Sigma_2, \ldots, \Sigma_k)$ of Σ. We define product systems over this.

Definition 7. *A sequential system over a set of actions Σ_i is a tuple $A_i = \langle P_i, \rightarrow_i, G_i, p_i^0 \rangle$ where P_i are called **places**, $G_i \subseteq P_i$ are final places, $p_i^0 \in P_i$ is the initial place, and $\rightarrow_i \subseteq P_i \times \Sigma_i \times P_i$ is a set of **local moves**.*

Let $\rightarrow_a^i$ denote the set of all a-labelled moves in the sequential system A_i.

A run of the sequential system A_i on word w is a sequence $p_0 a_1 p_1 a_2, \ldots, a_n p_n$, from set $(P_i \times \Sigma_i)^* P_i$, such that $p_0 = p_i^0$ and for each $j \in \{1, \ldots, n\}$, $p_{j-1} \xrightarrow{a_j} p_j$. This run is said to be accepting if $p_n \in G_i$. The sequential system A_i accepts word w, if there is at least one accepting run of A_i on w. The language $L = Lang(A_i)$ of sequential system A_i is defined as $L = \{w \in {\Sigma_i}^* | w \text{ is accepted by } A_i\}$.

Given a place p of A_i, we also define relativized languages and we will extend this definition to product systems: $Pref_a^p(L) = \{x \mid xay \in L, p_0 \xrightarrow{x} p \xrightarrow{ay} G_i\}$, similarly $Suf_a^p(L)$, $L^p = \{xay \mid xay \in L, p_0 \xrightarrow{x} p \xrightarrow{ay} G_i\}$. Say the place p **a-bifurcates** L if $L^p = Pref_a^p(L)\ a\ Suf_a^p(L)$.

Definition 8. *Let $A_i = \langle P_i, \rightarrow_i, G_i, p_i^0 \rangle$ be a sequential system over alphabet Σ_i for $1 \leq i \leq k$. A **product system** A over the distribution $\Sigma = (\Sigma_1, \ldots, \Sigma_k)$ is a tuple $\langle A_1, \ldots, A_k \rangle$.*

Let $\Pi_{i \in Loc} P_i$ be the set of product states of A. We use $R[i]$ for the projection of a product state R in A_i, and $R{\downarrow}I$ for the projection to $I \subseteq Loc$. The relativizations L^R of a language $L \subseteq \Sigma_i^*$ consider projections to place $R[i]$ in A_i.

The initial product state of A is $R^0 = (p_1^0, \ldots, p_k^0)$, while $G = \Pi_{i \in Loc} G_i$ denotes the final states of A.

Let $\Rightarrow_a = \Pi_{i \in loc(a)} \rightarrow_a^i$. The set of global moves of A is $\Rightarrow = \bigcup_{a \in \Sigma} \Rightarrow_a$. Then for a global move

$$g = \langle \langle p_{l_1}, a, p'_{l_1} \rangle, \langle p_{l_2}, a, p'_{l_2} \rangle, \ldots \langle p_{l_m}, a, p'_{l_m} \rangle \rangle \in \Rightarrow_a,\ loc(a) = \{l_1, l_2, \ldots, l_m\},$$

we write $g[i]$ for $\langle p_i, a, p'_i \rangle$, the projection to A_i, $i \in loc(a)$ and $pre(a)$ for the product states where such a move is enabled.

Please note that the set of product states as well as the global moves are not explicitly provided when a product system is given as input to some algorithm.

4.1 Properties of Product Systems

The first property for a product system is modelled on the free choice property of nets. It can be checked in PTIME by counting local moves with the same label. We also define another stronger property.

Definition 9. *For global $a \in \Sigma$, an a-matching is a subset of tuples $\Pi_{i \in loc(a)} P_i$, such that if a place $p \in P_j, j \in loc(a)$ appears in one tuple, it does not appear in another tuple. We say a product state R is in an a-matching if its projection $R{\downarrow}loc(a)$ is in the matching.*

A product system is said to have ***matching of labels*** *if for all global $a \in \Sigma$, there is an a-matching such that for $i, j \in loc(a), \langle p, a, q \rangle \in \rightarrow_i$, the pre-place p is matched to a pre-place p' such that $\langle p', a, q' \rangle \in \rightarrow_j$ and such that all pre-places with a-transitions are covered by the tuples of the matching. A product system A is said to have* ***separation of labels*** *if for all $i \in Loc$, if $\langle p, a, p' \rangle, \langle q, a, q' \rangle \in \rightarrow_i$ then $p = q$.*

Proposition 3. *Let $A = \langle A_1, \ldots, A_k \rangle$ be a product system over distribution $\Sigma = (\Sigma_1, \ldots, \Sigma_k)$. If A has separation of labels, then for every i and every global action a, $L_i = Lang(A_i)$ is a-bifurcated. If A has matching of labels, then for every i and every global action a,*

$$L_i \cap \Sigma_i^* a \Sigma_i^* = \bigcup_{R{\downarrow}loc(a) \in matching(a)} Pref_a^{R[i]}(L_i)\ a\ Suf_a^{R[i]}(L_i).$$

Proof. Let A be a product system as above with separation of labels. Let $L(q)$ be the set of words accepted starting from any place q in A_i. If $Pref_a(L(q))$ is nonempty then $L(q)$ is a-bifurcated, because the words containing a have to pass through a unique place. When A has a matching of labels, since the places $R[i]$ appear in unique tuples, one can separately consider the places a-bifurcating $L(q)$ and the required property follows. □

The next property is necessary for product systems to represent free choice in equivalent nets. In our earlier paper [LMP11] we used the definition of an FC-product below. The definition of FC-matching product is a generalization since conflict-equivalence is not required for all a-moves uniformly but refined into smaller equivalence classes depending on the matching.

Definition 10. *In a product system, we say the local move* $\langle p, a, q_1\rangle \in \rightarrow_i$ *is* ***conflict-equivalent*** *to the local move* $\langle p', a, q_1'\rangle \in \rightarrow_j$*, if for every other local move* $\langle p, b, q_2\rangle \in \rightarrow_i$*, there is a local move* $\langle p', b, q_2'\rangle \in \rightarrow_j$ *and, conversely, for moves from* p' *there are moves from* p*. If the product system has a matching of labels and we require this whenever* p, p' *are related by the matching, we call the matching* ***conflict-equivalent****. A system having a conflict-equivalent matching is a weaker condition than the system being conflict-equivalent.*

We call $A = \langle A_1, \ldots, A_k\rangle$ *an* ***FC-product*** *if for every global action* $a \in \Sigma$*, every* a*-labelled move in* A_i *is conflict-equivalent to every* a*-labelled move in* A_j*. We call* A *an* ***FC-matching product*** *if it has a conflict-equivalent matching.*

Checking that a system is an FC-product or an FC-matching product is in PTIME because one makes a pass through all transitions with the same locations, computing for each pre-place which partition it falls into.

Proposition 4. *Let* A *be an FC-matching product system. For any* i*, if there exist local moves* $\langle p, a, p'\rangle, \langle p, b, p''\rangle$ *in* $\rightarrow_i$*, then* $loc(a) = loc(b)$*.*

Proof. Since p has an outgoing a-move, p belongs to some tuple of $matching(a)$. If $j \in loc(a)$, then in this tuple there exists a state $q \in P_j$, which has an outgoing a-move. Since A is an FC-matching product, $matching(a)$ is conflict-equivalent. And, as states p and q appear in a tuple of $matching(a)$, these states are conflict-equivalent. Therefore there exists a local move $(q, b, q') \in \rightarrow_j$. This implies that $j \in loc(b)$. □

4.2 Language of a Product System

Now we describe runs of A over some word w by associating product states with prefixes of w: the empty word is assigned initial product state R^0, and for every prefix va of w, if R is the product state reached after v and Q is reached after va where, for all $j \in loc(a), \langle R[j], a, Q[j]\rangle \in \rightarrow_j$ and for all $j \notin loc(a), R[j] = Q[j]$. Let $pre(a) = \{R \mid \exists Q, R \xrightarrow{a} Q\}$.

A run is said to be accepting if the product state reached after w is in G. We define the language $Lang(A)$ of product system A, as the words on which the product system has an accepting run.

We use the following characterization of direct product languages, which appears in [MR02,Muk11].

Proposition 5. $L = Lang(A)$ *is the language of product system* $A = \langle A_1, \ldots, A_k\rangle$ *over distribution* Σ *iff*

$$L = \{w \in \Sigma^* \mid \forall i \in \{1, \ldots, k\},\ \exists u_i \in L \text{ such that } w{\downarrow_{\Sigma_i}} = u_i{\downarrow_{\Sigma_i}}\}.$$

Further $L = Lang(A_1) \| \ldots \| Lang(A_k)$*.*

The next definition is semantic, new to this paper and not easy to check (in PSPACE). If a system has separation of labels, the property obviously holds.

Definition 11. *A run of* A *is said to be* ***consistent with a matching of labels*** *if for all global actions* a *and every prefix of the run* $R^0 \stackrel{v}{\Rightarrow} R \stackrel{a}{\Rightarrow} Q$*, the pre-places* $R{\downarrow}loc(a)$ *are in the matching.*

5 Connected Expressions and Product Systems

In this section we prove the main theorems of the paper. To place them in context of our earlier paper [LMP11], there we used a "structural cyclicity" condition which allowed a run to be split into finite parts from the initial product state to itself, since it was guaranteed to be repeated. The new idea in this paper is that runs are split up using matchings which correspond to synchronizations, what happens in between is not relevant for the connections across sequential systems. Hence extending our syntax to allow full regular expressions for the sequential systems does not affect the synchronization properties which are the main issue we are addressing. In Section 6 we outline the connections to labelled free choice nets which are detailed in another paper [PL14].

5.1 Synthesis of Systems from Expressions

We begin by constructing product automata for our syntactic entities. For regular expressions, this is well known. We follow the construction of Antimirov, which in polynomial time gives us a finite automaton of size $O(wd(s))$, using partial derivatives as states.

Now we come to connected expressions, for which we will construct a product of automata.

Lemma 1. *Let e be a connected expression with unique global action sites. Then there exists a product system A with separation of labels accepting $Lang(e)$ as its language. If e had equal choice, then A is conflict-equivalent.*

Proof. Let $e = fsync(s_1, s_2, \ldots, s_k)$. Then for each s_i, which is a regular expression, defined over some alphabet Σ_i, we produce a sequential system A_i over Σ_i, using Antimirov's derivatives, such that $Lang(s_i) = Lang(A_i)$, $\forall\ i \in \{1, \ldots, k\}$. Next we trim it—remove places not reachable from the initial place p_i^0 and places from where a final state is not reachable. Now, for each global action a, we quotient A_i by merging all derivatives d such that $a \in Init(d)$ into a single place.

Call the resulting automaton A_i'. Let p be the merged place in A_i' which is now the source of all a-transitions. Clearly $Lang(A_i) \subseteq Lang(A_i')$ since no paths are removed, we show next that the inclusion in the other direction also holds, using the unique global action sites condition.

Let a be a global action. Consider a word $w = x_1 a x_2 \ldots a x_n$ in $Lang(A_i')$, where the factors $x_1, x_2, \ldots, x_n$ do not contain the letter a. We wish to find derivatives $d_0, d_1, \ldots, d_n$ of A_i such that d_n is a final place and for every j there is a run $d_j \xrightarrow{a x_{j+1}} \ldots \xrightarrow{a x_n} d_n$ of A_i when $j > 0$, and $d_0 \xrightarrow{x_1} \xrightarrow{a x_2} \ldots \xrightarrow{a x_n} d_n$ when $j = 0$, which will show the desired inclusion.

We proceed from n downwards. For any place d_n in G there is a run from d_n on $\varepsilon \in Lang(d_n)$ in A_i. Inductively assume we have d_j such that there is a run $d_j \xrightarrow{a x_{j+1}} \ldots \xrightarrow{a x_n} d_n$ of A_i, so $x_{j+1} a x_{j+2} \ldots a x_n$ is in $Suf_a(Lang(s_i))$ since d_j is reachable from the initial place. Since there is a run $p \xrightarrow{a x_j} p$ in A_i' there are derivatives d_{j-1}, c_j of e, such that there is a run $d_{j-1} \xrightarrow{a x_j} c_j$ in A_i (when $j = 1$

we get $d_0 \xrightarrow{x_1} c_1$ by this argument). Since c_j quotients to p, it has an a-derivative c such that c is in $Der_{ax_ja}(d_{j-1})$ ($Der_{x_0a}(d_0)$ when $j = 1$). Because d_{j-1} is reachable from the initial place by some v and because some final state is reachable from c, $vx_j \in Pref_a(Lang(s_i))$ which is nonempty. By the unique global action sites condition and Proposition 1, since $x_{j+1} \dots ax_n$ is in $Suf_a(Lang(s_i))$, $vax_jax_{j+1} \dots ax_n$ is in $Lang(s_i)$ and so $x_jax_{j+1} \dots ax_n$ is in $Suf_a(Lang(s_i))$. This means that there is a run from some d_{j-1} on $ax_jax_{j+1} \dots ax_n$ ending in a final state d_n of A_i. So we have the induction hypothesis restored. If $j = 1$ we get d_0 which quotients to p_0 and has a run on w to d_n in G.

So we get a product system $A' = \langle A'_1, A'_2, \dots, A'_k \rangle$ defined over Σ. If the expression had equal choice, this system is conflict-equivalent. Because of the quotienting A' has separation of labels.

$$\begin{aligned} w \in Lang(e) &\text{ iff } \forall i, w{\downarrow_{\Sigma_i}} \in Lang(s_i), \text{ by definition} \\ &\text{iff } \forall i, w{\downarrow_{\Sigma_i}} \in Lang(A'_i) \\ &\text{iff } w \in Lang(A'), \text{ by Proposition 5.} \end{aligned}$$

Theorem 1. *Let $e = fsync(s_1, \dots, s_k)$ be a connected expression over a distribution Σ with a pairing of actions. Then there exists an FC-matching product system A over Σ, accepting $Lang(e)$. If the expression had deterministic sites, the constructed product will have deterministic global actions. If the pairing was equal choice, the matching is conflict-equivalent. If the expression is consistent with the pairing, all runs of A will be consistent with the matching.*

Proof. We first rewrite e to another expression e', construct an automaton A' for $Lang(e')$, and then change it to recover an automaton for $Lang(e)$.

Consider global action a and tuple of blocks $D = \Pi_{i \in loc(a)} D_i \subseteq pairing(a)$. By Proposition 1 D_i a-bifurcates $Lang(s_i)$. We rename for all i in $loc(a)$, the occurrences of a in s_i which correspond to an a in $Init(D_i)$, by the new letter a^D. This is done for all global actions to obtain from e a new expression $e' = fsync(s'_1, \dots, s'_k)$ over a distribution Σ', where every s'_i now has the unique sites property. For any word $w \in Lang(e)$, there is a well-defined word $w' \in Lang(e')$.

By Lemma 1 we obtain an FC-product A' with separation of labels for $Lang(e')$. Say $p(a^D)$ is the pre-place for action a^D in A'_i. We change all the $\langle p(a^D), a^D, q \rangle$ transitions to $\langle p(a^D), a, q \rangle$ in all the A'_i to obtain an FC-product A over the alphabet Σ. As $w' \in Lang(e') = Lang(A')$ is well-defined from w and, as the renaming of transition labels does not remove any paths, w is in $Lang(A)$. Conversely, for every run on w accepted by A, because of the separation of labels property, there is a well-defined run on w' with the label of a transition appropriately renamed depending on the source state, which is accepted by A', hence w' is in $Lang(e')$. So renaming w' to w gives a word in $Lang(e)$. This construction preserves determinism.

Now we refer to the pairing of actions in e. This defines for each global action a and tuple of blocks of a-sites D, a relation between pre-places of a^D-moves in different components in the product A'. By the separation of labels property of A', the tuples in the relation are disjoint, that is, the relation is functional. So

for pre-places of a-moves in the product A we have a matching. If the pairing was equal choice, the matching is conflict-equivalent.

If the expression e is consistent with the pairing, all reachable a-sites are in the pairing, so we can partition $Lang(e) \cap \Sigma^* a \Sigma^*$ using the partitions in $Part_a(e)$. Letting D range over blocks of connected expressions, each block D contributes a global action a^D in the renaming, so we get an expression e' such that for every global action a^D, we have the unique a-sites property. Applying Lemma 1, we have the product system A' with separation of labels. By Proposition 3, every $Lang(A'_i)$ is a^D-bifurcated, and using the characterization of Proposition 5, $Lang(A') \cap (\Sigma')^* a^D (\Sigma')^* = Pref_{a^D}(Lang(A')) a^D Suf_{a^D}(Lang(A'))$. Since several actions a^D are renamed to a and the corresponding tuples of pre-places are recorded in the matching, by Proposition 3 and Proposition 5:

$$\bigcup_{R \in matching(a)} Pref_a^R(Lang(A))\ a\ Suf_a^R(Lang(A)) \subseteq Lang(A) \cap \Sigma^* a \Sigma^*.$$

But this means that all runs of A are consistent with the matching. □

5.2 Analysis of Expressions from Systems

Lemma 2. *Let A be a FC-product system with separation of labels. Then we can compute a connected expression for the language of A, where every regular expression has unique sites. If the FC-product had deterministic global actions, then so do the regular expressions in the computed expression. If the FC-product was conflict-equivalent, the constructed expression has equal choice.*

Proof. Let $A = \langle A_1, \ldots, A_k \rangle$ be an FC-product with separation of labels, where A_i is a sequential system of A with places P, initial place p_0 and final places G. Kleene's theorem gives us an expression s_i for the language of A_i. We claim the required connected expression is $fsync(s_1, \ldots, s_k)$.

Consider global action a. By separation of labels there is a single state p in A_i enabling a. For simplicity let us assume there is only one global action a enabled at p. Let $Q = P \setminus \{p\}$. Let T be the set of transitions excluding the a-actions enabled at p. We wish to decompose the expression s_i that we started with into paths which go through p and paths which do not. Depending on whether we have a sequential transition $p \xrightarrow{a} p$, or transitions $p \xrightarrow{a} p_j,\ p_j \neq p$, or a combination of these two types, we obtain an expression with the same language as s_i:

$$e_p = \sum_{f \in G} e^T_{p_0,f} + e^T_{p_0,p} e^*_{p,p} e^Q_{p,f},$$

where the expression $e_{p,p}$ is given by one of the following refinements, for the three cases considered above respectively:

$$(a + e^T_{p,p}),\ \text{or}\ ((\sum_j a e^T_{p_j,p}) + e^T_{p,p}),\ \text{or}\ (a + (\sum_j a e^T_{p_j,p}) + e^T_{p,p}).$$

The superscripts T, Q indicates that these expressions are derived, as in the McNaughton-Yamada construction [MY60], for runs which only use the states Q or transitions T. Whichever be the case, we note that we have an expression with $D^a(e_p) = \{e^*_{p,p}e^Q_{p,f}\}$ as its singleton set of a-sites. If the system had deterministic global actions, the a-site would have only had one a-derivative. This idea can be easily extended to considering several global actions enabled at the same place, by considering a different refinement of s_i taking into account the combined possibilities. If the product system was conflict-equivalent, the a-sites are all equal choice.

But the expression s_i could have been obtained by considering the place p at an arbitrary point in the McNaughton-Yamada construction. Consider e_p as refining some intermediate expression s'_i for the place p. The expression e_p may make copies of parts of s'_i. This does not affect the deterministic global actions property. For $c \neq a$ the c-sites $D^c(e_p)$ are obtained as:

$$D^c(e_p) = \bigcup_{f \in G} D^c(e^T_{p_0,f}) \cup D^c(e^T_{p_0,p}) \cup D^c(e_{p,p}) \cdot e^*_{p,p} \cdot e^Q_{p,f} \cup D^c(e^Q_{p,f}).$$

That is, $Part_c(e_p)$ is preserved as a single block if it formed a single block in the earlier expressions. Thus the expression s_i has the unique sites property. □

Theorem 2. *Let A be a FC-matching product system. Then we can compute a connected expression for the language of A, where every regular expression has a pairing of actions. If the FC-product had deterministic global actions, then so do the regular expressions in the computed expression. If the matching was conflict-equivalent the pairing is equal choice. If all runs of A were consistent with the matching, the expression constructed will be consistent with the pairing.*

Proof. Let A be a product system with a conflict-equivalent matching. Enumerate the global actions $a, b, \ldots$. Say the a-matching has n tuples.

We construct a new product system A' where, for the places in the j'th tuple of the a-matching, we change the label of the outgoing a-transitions to a^j; similarly for the places in tuples of the b-matching; and so on. We now have a new product system where the letter a of the alphabet has been replaced by the set $\{a^1, \ldots, a^n\}$; the letter b has been replaced by another set; and so on, obtaining a new distribution Σ'. By definition of a matching, the various labels do not interfere with each other, so we have a matching with the new alphabet, conflict-equivalent if the previous one was. Runs which were consistent with the matching continue to be consistent with the new matching. Again by the definition of matching, the new system A' has separation of labels. Hence we can apply Lemma 2.

From the lemma we get a connected expression $e' = fsync(s_1, \ldots, s_k)$ for the language of A' over Σ' where every regular expression has unique global action sites. From the proof of the lemma we get for every sequential system A'_i in the product, for the global actions $a^1, \ldots, a^n$, tuples $D'(a^j) = \Pi_{i \in loc(a)} D'_i(a^j)$ which are sites for a^j in the expression s_i, for every j. Now substitute a for every letter $a^1, \ldots, a^n$ in the expression, each tuple D' is isomorphic to a tuple D of sites

for a in e and the sites are disjoint from one another. We let $pairing(a)$ be the partition formed by these tuples. Do the same for b obtaining $pairing(b)$. Repeat this process until all the global actions have been dealt with. The result is an expression e with pairing of actions. If the matching was conflict-equivalent, the pairing has equal choice.

The runs of A have to use product states in $pre(a)$ for global action a, define

$$L = Lang(A) \cap \Sigma^* a \Sigma^* = \bigcup_{R \in pre(a)} Pref_a^R(Lang(A))\ a\ Suf_a^R(Lang(A)).$$

The renaming of transitions depends on the source state, so L is isomorphic to

$$L' = Lang(A') \cap (\sum_j (\Sigma')^* a^j (\Sigma')^*) = \bigcup_{j=1,n} Pref_{a^j}(Lang(A')) a^j Suf_{a^j}(Lang(A')).$$

Keeping Proposition 5 in our hands, the lemma ensures that $Lang(A') = Lang(e')$ and the expression e' has unique a^j-sites forming a block $D'(j)$. Then L' can be written as $\bigcup_{j=1,n} Pref_{a^j}^{D'(j)}(Lang(e')) a^j Suf_{a^j}^{D'(j)}(Lang(e'))$. When we rename the a^j back to a we have a partition of $pairing(a)$ into sets D such that

$$L = \bigcup_{D \subseteq pairing(a)} Pref_a^D(Lang(e))\ a\ Suf_a^D(Lang(e)).$$

If all runs of A were consistent with the matching, the product states in $pre(a)$ would all be in the matching, and we obtain that the expression e is consistent with the pairing. □

6 Nets

Definition 12. *A labelled net N is a tuple (S, T, F, λ), where S is a set of places, T is a set of transitions labelled by the function $\lambda : T \to \Sigma$ and $F \subseteq (T \times S) \cup (S \times T)$ is the flow relation. It will be convenient to define $loc(t) = loc(\lambda(t))$.*

Elements of $S \cup T$ are called nodes of N. Given a node z of net N, set ${}^\bullet z = \{x \mid (x, z) \in F\}$ is called pre-set of z and $z^\bullet = \{x \mid (z, x) \in F\}$ is called post-set of z. Given a set Z of nodes of N, let ${}^\bullet Z = \bigcup_{z \in Z} {}^\bullet z$ and $Z^\bullet = \bigcup_{z \in Z} z^\bullet$. We only consider nets in which every transition has nonempty pre- and post-set.

Definition 13. *Let $N' = (S \cap X, T \cap X, F \cap (X \times X))$ be a subnet of net $N = (S, T, F)$, generated by a nonempty set X of nodes of N. N' is called a **component** of N if,*

- *For each place s of X, ${}^\bullet s, s^\bullet \subseteq X$ (the pre- and post-sets are taken in N),*
- *For all transitions $t \in T$, we have $|{}^\bullet t| = 1 = |t^\bullet|$ (N' is an S-net [DE95]),*
- *Under the flow relation, N' is connected.*

A set $\mathcal{C}$ of components of net N is called ***S-cover*** *for N, if every place of the net belongs to some component of $\mathcal{C}$. A net is* covered by components *if it has an S-cover.*

Note that our notion of component does not require strong connectedness and so it is different from notion of S-component in [DE95], and therefore our notion of S-cover also differs from theirs.

Fix a distribution $(\Sigma_1, \Sigma_2, \ldots, \Sigma_k)$ of Σ. The next definition appears in several places for unlabelled nets, starting with [Hac72].

Definition 14. *A labelled net $N = (S, T, F, \lambda)$ is called* ***S-decomposable*** *if, there exists an S-cover $\mathcal{C}$ for N, such that for each $T_i = \{\lambda^{-1}(a) \mid a \in \Sigma_i\}$, there exists S_i such that the induced component (S_i, T_i, F_i) is in $\mathcal{C}$.*

Now from S-decomposability we get an S-cover for net N, since there exist subsets $S_1, S_2, \ldots, S_k$ of places S, such that $S = S_1 \cup S_2 \cup \ldots S_k$ and ${}^\bullet S_i \cup S_i^\bullet = T_i$, such that the subnet (S_i, T_i, F_i) generated by S_i and T_i is an S-net, where F_i is the induced flow relation from S_i and T_i.

6.1 Properties of Nets

Definition 15 ([DE95]). *Let x be a node of a net N. The* cluster *of x, denoted by $[x]$, is the minimal set of nodes contaning x such that*

- *if a place $s \in [x]$ then $s^\bullet$ is included in $[x]$, and*
- *if a transition $t \in [x]$ then ${}^\bullet t$ is included in $[x]$.*

A cluster C is called free choice *(FC) if all transitions in C have the same pre-set. A net is called* free choice *if all its clusters are free choice.*

The next definitions will turn out to be the analogue to the separation of labels property of product systems. It is checkable in linear time.

Definition 16. *A labelled net $N = (S, T, F, \lambda)$ is said to have the* ***unique cluster property*** *(briefly,* ***ucp****) if $\forall a \in \Sigma$ having $|loc(a)| > 1$, there exists at most one cluster in which all transitions labelled a occur. It is* ***deterministic for synchronization*** *if for every a, every cluster contains at most one a-labelled transition.*

6.2 Net Systems and their Languages

For our results we are only interested in 1-bounded (or condition/event) nets, where a place is either marked or not marked. Hence we define a marking as a function from the states of a net to $\{0, 1\}$.

A transition t is enabled in a marking M if all places in its pre-set are marked by M. In such a case, t can be fired to yield the new marking $M' = (M \setminus {}^\bullet t) \cup t^\bullet$. We write this as $M[t\rangle M'$ or $M[\lambda(t)\rangle M'$.

A firing sequence (finite or infinite) $\lambda(t_1)\lambda(t_2)\ldots$ is defined by composition, from $M_0[t_1\rangle M_1[t_2\rangle\ldots$ For every $i \leq j$, we say that M_j is reachable from M_i. A net system (N, M_0) is **live** if, for every reachable marking M and every transition t, there exists a marking M' reachable from M which enables t.

Definition 17. *For a labelled net system $(N, M_0, \mathcal{G})$, its language is defined as $Lang(N, M_0, \mathcal{G}) = \{\lambda(\sigma) \in \Sigma^* \mid \sigma \in T^* \text{ and } M_0[\sigma\rangle M, \text{ for some } M \in \mathcal{G}\}$.*

If a net (S, T, F, λ) is 1-bounded and S-decomposable then a marking can be written as a k-tuple from its components $S_1 \times S_2 \times \ldots \times S_k$. It is known [Zie87,Muk11] that if we do not enforce the "direct product" condition below we get a larger subclass of languages.

Definition 18. *An **S-decomposable labelled net system** $(N, M_0, \mathcal{G})$ is an S-decomposable labelled net $N = (S, T, F, \lambda)$ along with an initial marking M_0 and a set of markings $\mathcal{G} \subseteq \wp(S)$, which is a **direct product**: if $\langle q_1, q_2, \ldots q_k\rangle \in \mathcal{G}$ and $\langle q'_1, q'_2, \ldots q'_k\rangle \in \mathcal{G}$ then $\{q_1, q'_1\} \times \{q_2, q'_2\} \times \ldots \times \{q_k, q'_k\} \subseteq \mathcal{G}$.*

6.3 Product Systems to Nets

Given a product system $A = \langle A_1, A_2, \ldots, A_k\rangle$ over distribution Σ, we can produce a net system $(N = (S, T, F, \lambda), M_0, \mathcal{G})$ as follows using a standard construction. When we construct nets from product systems with a conflict-equivalent matching of labels with respect to which all runs are consistent, we can refine the construction above to choose $T' \subseteq T$ and get a free choice net.

Theorem 3 ([PL14]). *Let $(N, M_0, \mathcal{G})$ be the net system constructed from product system A above. Then N is an S-decomposable net with $Lang(N, M_0, \mathcal{G}) = Lang(A)$. Further, if A has deterministic global actions and all runs of A are consistent with a conflict-equivalent matching of labels, we can choose $T' \subseteq T$ such that the subnet N' generated by T' is a free choice net with deterministic synchronization and $(N', M_0, \mathcal{G})$ accepts the same language.*

6.4 Nets to Product Systems

Even if a net is 1-bounded and S-decomposable each component need not have only one token in it, but when we say that a 1-bounded net is S-decomposable we assume that each component has one token. For live and 1-bounded free choice nets, such S-covers can be guaranteed [DE95]. Now we can prove:

Theorem 4 ([PL14]). *Let $(N, M_0, \mathcal{G})$ be a live, 1-bounded, S-decomposable labelled free choice net system with deterministic synchronization. Then one can construct a product system A with deterministic global actions, which has a conflict-equivalent matching of labels that all its runs are consistent with. Further $Lang(N, M_0, \mathcal{G}) = Lang(A)$.*

7 Conclusion

In earlier work [LMP11], we showed that a graph-theoretic condition called "structural cyclicity" enables us to extract syntax from a conflict-equivalent product system. In the present work we have generalized this condition so that we can deal with a larger class of product systems with a conflict-equivalent matching. In our paper [PL14] we show a connection between free choice nets with deterministic synchronization and product systems which have these properties along with deterministic global actions. Thus we obtain a Kleene characterization for the class of labelled free choice nets with deterministic synchronization.

Acknowledgements. We would like to thank the referees of the PNSE workshop for urging us to improve the presentation of the proofs of the main theorems. This led us to invent Definition 3 and correct the site properties in Definition 5.

References

[Ant96] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comp. Sci.*, 155(2):291–319, 1996.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[DE95] Jörg Desel and Javier Esparza. *Free choice Petri nets.* Cambridge University Press, New York, USA, 1995.

[GR92] Vijay K. Garg and M.T. Ragunath. Concurrent regular expressions and their relationship to Petri nets. *Theoret. Comp. Sci.*, 96(2):285–304, 1992.

[Gra81] Jan Grabowski. On partial languages. *Fund. Inform.*, IV(2):427–498, 1981.

[Hac72] Michel Henri Théodore Hack. Analysis of production schemata by Petri nets. Project Mac Report TR-94, MIT, 1972.

[LMP11] Kamal Lodaya, Madhavan Mukund, and Ramchandra Phawade. Kleene theorems for product systems. In Markus Holzer, Martin Kutrib, and Giovanni Pighizzini, editors, *Proc. 13th DCFS, Limburg*, volume 6808 of *LNCS*, pages 235–247, 2011.

[Mir66] Boris G. Mirkin. An algorithm for constructing a base in a language of regular expressions. *Engg. Cybern.*, 5:110–116, 1966.

[MR02] Swarup Mohalik and R. Ramanujam. Distributed automata in an assumption-commitment framework. *Sādhanā*, 27, part 2:209–250, April 2002.

[Muk11] Madhavan Mukund. Automata on distributed alphabets. In Deepak D'Souza and Priti Shankar, editors, *Modern applications of automata theory*, pages 257–288. World Scientific, 2011.

[MY60] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IEEE Trans. IRS*, EC-9:39–47, 1960.

[PL14] Ramchandra Phawade and Kamal Lodaya. Direct product automaton representation of labelled free choice nets. Submitted, 2014.

[SH96] Pablo A. Straub and L. Carlos Hurtado. Business process behaviour is (almost) free-choice. In *Proc. CESA, Lille*, pages 9–12. IEEE, 1996.

[Zie87] Wiesław Zielonka. Notes on finite asynchronous automata. *Inform. Theor. Appl.*, 21(2):99–135, 1987.

Using Symbolic Techniques and Algebraic Petri Nets to Model Check Security Protocols for Ad Hoc Networks

Mihai Lica Pura and Didier Buchs

Centre Universitaire d'Informatique
University of Geneva
Carouge, Switzerland

Abstract. Petri nets have proved their effectiveness in modeling and formal verification of a large number of applications: control systems, communication protocols, application workflows, hardware design, etc. In the present days, one important focus of computer science is on security and secure communications. The use of Petri nets for verifying security properties is not a mature field due to a lack of convenient modeling and verification capabilities. So far, in the Petri Net field there is only the CPN tool that is mature enough for modeling using the colored Petri nets formalism. Nevertheless verification cannot be performed on large systems as CPN tool verification is based on an exhaustive way of computing the semantics of a model. In this paper we present the use of AlPiNA, another candidate for this task. AlPiNA is a symbolic model checker that uses the formalism of algebraic Petri nets. We have used it successfully for modeling ad hoc networks and for verifying security protocols designed for this type of networks. As a case study and benchmark we have chosen the ARAN secure routing protocol. We managed to find all the attacks that were already reported for this protocol. To our knowledge this work is also the first successful attempt to use Petri nets for model checking the security properties of ad hoc networks protocols.

Keywords: model checking, ad hoc networks, algebraic Petri Nets.

1 Introduction

Place/Transition nets are a modeling language that proved its effectiveness in modeling a large variety of systems based on concurrent processes. Over the years, the initial Petri net formalism was enriched in order to simplify the specification of more and more complex systems. Two of the applications targeted were the model checking of security protocols and of the ad hoc network protocols (but not ad hoc network security protocols). To the best of our knowledge, model checking the security protocols specially designed for ad hoc networks has not been reported yet.

There is no need to argue for the importance of security in computer science, or for the need to prove the security properties of the protocols used in the

information systems. Ad hoc networks are a novel approach to assuring communications. The communications networks that are now in use are based on an infrastructure composed of devices like switches, hubs, gateways, routers, and so on. Ad hoc networks aim to assure communications without the use of any infrastructure. In such networks there are no other devices, except the ones that actually form it, and want to communicate. And they will also act as the infrastructure devices from a classical network, by routing the messages of all the other nodes. Such a behavior is assured by specially designed ad hoc routing protocols. These routing protocols and their possible attack schemes are more complex than the ones of the other kinds of networks, so for their specification a more powerful language is needed.

One of the enrichments of P/T nets dedicated specifically to data based functionality is High Level Petri Nets (HLPN). In HLPN the tokens have different types and these types are part of a many-sorted algebra ([1]). The possibility to use other types than the usual black tokens made it possible to use HLPN in modeling and verification of security protocols.

Colored Petri Nets (CPN) were the first concrete realization of HLPN that were used for model checking security properties, because they were the first one who was expressive enough for this ([2]). But besides CPN, there are other implementations of HLPN. The difference between the different implementations of HLPN stands in the way the many-sorted algebra is defined. In CPN the many-sorted algebra is defined using the CPN ML language, which was built upon the standard ML.

For modeling ad hoc networks we focus on the model checker AlPiNA ([3, 4]). AlPiNA implements HLPN by algebraic Petri nets (APN), in which the colored tokens are defined using algebraic abstract data types (AADT) ([1]). Like all the other model checkers, the focus of AlPiNA is to handle the state explosion problem in order to perform verification on real size system models. When using HLPN, the state space explosion has one more dimension (the data) than in the case of P/T nets. HLPN are more expressive and as a consequence, the state space of a HLPN model is in general much bigger. AlPiNA addresses this problem by using symbolic techniques based on several layers of Data Decision Diagrams, Set Decision Diagrams and Sigma Decision Diagrams [1]. In addition, some optimizations specific to the APN formalism (algebraic clustering, partial algebraic unfolding) [5] are supported. The tool can be downloaded from [5].

We have successfully used AlPiNA for modeling ad hoc networks and for model checking security protocols of ad hoc networks. From our studies, we have seen some advantages that this tool has over the other tools used for these purposes; in terms of modeling the protocol itself, as well as the possible attackers. In this paper we will present the modeling of ad hoc networks and the verification of ARAN (Authenticated Routing for Ad Hoc Networks [6]) security protocol with APNs, and the advantages of AlPiNA for performing these tasks.

The rest of the paper is organized as follows. The second section presents the use of Petri nets in literature for modeling ad hoc networks and verifying properties related to them. In the third section we describe the use of algebraic Petri

nets and AlPiNA for modeling ad hoc networks and the ARAN protocol. The fourth section contains the presentation of our results regarding verification of routing information correctness for ARAN. The last section contains conclusions and our future work directions.

2 The Use of Petri nets in modeling ad hoc networks

Petri nets already proved their effectiveness in modeling ad hoc networks. So far, researchers have used Fuzzy Petri nets, Stochastic Petri nets and Colored Petri nets to model ad hoc networks. The purpose of these models was to obtain qualitative or quantitative information about the behavior of applications and protocols in the context of ad hoc networks. As far as we know, algebraic Petri nets were never used so far to model ad hoc networks.

We will continue by presenting some of the latest published results concerning the use of Petri nets in ad hoc networks research.

2.1 Modeling for Quantitative evaluation

The research presented in [7] uses Fuzzy Petri Nets for modeling and analyzing the QoS dimension in order to evaluate how to manage congestion in wireless ad hoc networks. The networks itself, the nodes, the communication protocol are not actually modeled. In [8] Fuzzy Petri Nets are used to represent the multicast routing in an ad hoc network and to calculate multicast trees. The authors only model the topology of the network but not the actual routing protocol.

In [9] the authors present how to use Stochastic Petri Nets to model ad hoc networks. An ad hoc network is modeled by a single node, for which a proper amount of traffic is generated. By measuring how the node behaves under the given traffic, using suitable metrics, some conclusions can be obtained regarding a whole network with a given number of nodes like the modeled one. In [10] Stochastic Petri Nets are used to model mobility of ad hoc networks, but the actual ad hoc network is not modeled, neither the ad hoc routing, only an application level protocol that takes into account the fact that the nodes are moving between different geographic regions, and also the required performance indices. Thus the authors are able to obtain quantitative data about the specified performance indices.

The authors of [11] and [12] use Colored Petri Nets. They propose models for the nodes of the network, for the routing protocol AODV (Ad Hoc On-Demand Distance Vector Routing) [12] and DSR (Dynamic Source Routing) [11] and for the behavior of the ad hoc network. The purpose of the modeling was to conduct a comparison between the two ad hoc routing protocols mentioned above, from the point of view of their efficiency (number of generated overhead packets, data packet delivery delay). In [13] Colored Petri Nets are used to model and to compare another pair of routing protocols, AOMDV (Ad Hoc On-Demand Multipath Distance Vector Routing) and DSR. In [14], Colored Petri Nets are used to model and validate the specification of a multicast routing protocol for

ad hoc networks called DYMO (Dynamic MANET On-Demand). The properties that the authors specify and verify are all related to the correctness of the protocol: establishments of routes, and correct processing of the routing messages. By this work, the authors also found several ambiguities in the definition of the protocol, which were taken into consideration in two revisions.

2.2 Modeling for Qualitative evaluation

From the point of view of model checking security protocols, Colored Petri nets are the only type of Petri nets used for this purpose up to now. But as far as we know, no Petri nets were used to model check the security protocols of ad hoc networks. So our paper is the first presentation using algebraic Petri nets to model ad hoc networks and to do model checking of security properties for specific ad hoc network protocols.

For example, [2] and [15] present the work of using CPN to model check confidentiality and authentication for TMN authenticated key exchange protocol. In [16] CPN are used to verify the same security properties for Andrew secure RPC protocol. In all these papers, the use of CPN helps to find attacks over the considered protocols, and even some attacks that were previously unknown. So this indicates the high potential of using these techniques for model checking ad hoc network specific security protocols.

In the next sections, we will present the state of the art of modeling ad hoc networks with the help of Petri nets. Modeling an ad hoc network implies modeling the following elements: the nodes and the topology of the network.

2.3 Modeling the nodes

For modeling the nodes of an ad hoc network, a single approach was used by all the researchers. The nodes were modeled by their behavior in the considered protocol or application. The Petri net contains a single instance of a node's behavior. But this behavior is parameterized with the identity of a node. The identities of the nodes, which are part of the considered network, are placed inside a special place. When the state space is calculated, all these identities are considered as executing the modeled behavior ([11]).

2.4 Modeling the topology

When modeling the topology of the ad hoc networks, two aspects should be taken into consideration. The first one is how to model the actual topology of the network at a given time. The second aspect is how to model the mobility of the nodes which implies the modeling of the dynamicity of the topology. Both of these aspects influence the modeling of the way messages travel through the network. Based on the current topology, a message transmitted by a node should only be received by the other nodes which are in the coverage area of the transmitting node.

So far, researches have proposed three ways for modeling topology. We will briefly present them in the following paragraphs.

In [11], [12] and [13] the network topology was modeled by an approximation mechanism. Let us presume that the network has n nodes. When a node A sends a broadcast message, it actually sends n-1 copies of the message to a place that stores them in order to distribute them to the corresponding nodes. Based on a probability that represents how many nodes are in the coverage area of A, a certain number of these messages will be forwarded to other nodes, and the remaining messages will be dropped. In the case of unicast messages, they are sent only to the corresponding nodes. The authors of [12] call this model a topology approximation mechanism and prove through simulation that it can indeed mimic the mobility of a mobile ad hoc network (MANET).

In [14] the wireless mobile ad hoc network is modeled by two parts: a part that handles the transmission of the packets, and another part that handles the mobility of the nodes. The transmission of the packets is done based on the current topology of the network, which is explicitly represented in the following way: each node A has an adjacency list of nodes. Each node from this list is a node that is in the coverage area of A, and thus can receive packets from it. Based on the information from these lists, the transmission part of the model of the ad hoc network sends the packets to the appropriate nodes. The mobility part of the model is responsible with making modification to the topology. At the beginning of the validation, there is an initial topology and also the possible topology changes. Based on these changes, the mobility part modifies the topology as the validation continues.

The authors of [17] and [18] use reconfigurable algebraic higher-order net systems in order to model mobility for the ad hoc networks. The idea is to apply graph transformation (rewriting of the model) to algebraic nets. That is, the net gets reconfigured at run time in order to simulate the mobility of the nodes in an ad hoc network. The modeling is abstracted from the network layer, and the considered application is modeled in terms of work-flows.

3 Using Algebraic Petri Nets in Modeling Ad Hoc Networks

3.1 Algebraic Petri nets definition

An APN is a HLPN where algebraic abstract data types are used. The structure of the net is the structure of a Place/Transition net, but algebraic values are used as tokens. Also, the transitions can have guards that are pairs of algebraic terms that allow the firing of the respective transitions. In the following a sketch of the model components are given, more details can be found in [1].

An algebraic Petri net specification is a 5-tuple
$N - SPEC =< Spec, T, P, X, AX >$, where:

- $Spec =< \Sigma, X', E >$ is an algebraic specification extended in $< [\Sigma], X', E >$, where $[\Sigma]$ is a multiset over the signature $\Sigma =< S, F >$ ([19]) such that:

  - S is a finite set of sorts;
  - $F = (F_{w,s})_{w \in S^*, s \in S}$ is a $(S^* \times S)$ sorted set of function names;
- T is the set of transition names;
- P is the set of place names and there is a function $\tau : P \to S$ which associates a sort to each place;
- X is a S-sorted set of variables;
- AX is a set of axioms and it will be defined below.

Given an algebraic Petri net specification $N{-}SPEC =< Spec, T, P, X, AX >$, an axiom in AX is a 4-tuple $< t, Cond, In, Out >$ such that:

- $t \in T$ is the transition name for which the axiom is defined;
- $Cond \subseteq T_{\Sigma,X} \times T_{\Sigma,X}$ is a set of equalities attached to the transition name t for this axiom; $Cond$ is satisfied if and only if all the equalities from the set are satisfied;
- $In = (In_p)_{p \in P}$ is a P-sorted set of terms such that $\forall p \in P, In_p \in (T_{[\Sigma],X})_{[\tau(p)]}$ is the label of the arc from place p to transition t;
- $Out = (Out_p)_{p \in P}$ is a P-sorted set of terms such that $\forall p \in P, Out_p \in (T_{[\Sigma],X})_{[\tau(p)]}$ is the label of the arc from transition t to place p.

In AlPiNA, the input of a transition is a set that can only contain variables and closed terms [4]. However, this limitation has no effect over the complexity of the systems that can be modeled and verified. It is just simplifying the complexity of the computations.

In order to provide a semantics to a specification $N - SPEC$, we can define the set of reachable states $StN - SPEC(M)$ from a given marking M. In this paper we do not need the precise definition; please consult [1] for more details.

3.2 Case study: ARAN secure routing protocol

In order to present our methodology for modeling ad hoc networks, we have taken as case study the ARAN secure routing protocol. We have chosen it because it is simple, well known and it is the state of the art regarding secure routing in ad hoc networks. The purpose of ARAN is to provide a route path for any node in the network. It is an implicit routing protocol, which means that it will not respond with the whole path, but only with the identity of the next node in the path. ARAN uses digital signatures to assure authentication and integrity for the exchanged routing information.

ARAN uses two message types: route discovery and route response. Each message is signed by its source node. As it travels to its destination, the signed message is also cosigned by each intermediate node, after eliminating the signature of the previous intermediary, if it exists. Each node validates the received message by validating the signature(s) from the message. If the signature(s) are not valid, the message is discarded. Otherwise, the intermediary node broadcasts the message, if it is a route discovery message, or unicast it, if it is a route response message. When a route discovery message reaches destination, the node will respond with a route response message. When a route response

reaches destination, the node will modify its routing table accordingly. Also, each intermediary node that receives a routes response for a route discovery that he processed, will also update its routing table. Each route from the routing table has a given lifetime. When no traffic has occurred on an existing route for that route's lifetime, the route is deactivated. When data is received for an inactive route, the corresponding note will demand the source node of the data to make a new route request for the targeted destination node. So topology changes will determine route inactivation in some nodes' routing tables, which will further determine new route requests for the destination. For more information regarding the protocol, please consult [6].

The modeling of ARAN for the purpose of its verification implies the modeling of the following elements: the nodes, the ad hoc network, the adversary and the protocol operation. The general model for ARAN is given in Fig. 1. We will now continue with the presentation of all the parts of the model.

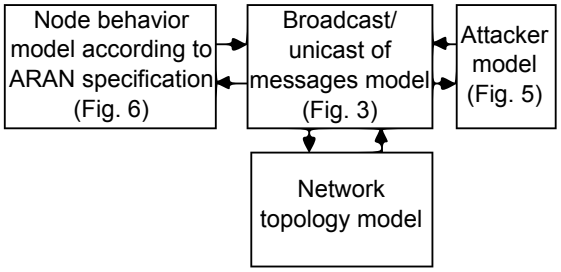


Fig. 1. ARAN general model

3.3 Modeling the nodes

A node of the ad hoc network is modeled as a AADT *Node*. Each node has an identity which is unique in the ad hoc network. Each node has also a routing table and some other structures needed for the operation of the considered ad hoc routing protocol. Because ARAN uses digital signatures, each node also has a pair of public/private keys and a digital certificate. In addition, each node knows the public key of the certification authority that issued his certificate.

Since all the nodes are identical, they all behave the same way. So in the actual Petri net, all the nodes are placed inside the same place called *Nodes* collecting identifiers of type *Node*. Here is the AADT *Node* in the case of ARAN:

```
Adt node
Sorts node;
Generators
node: Identity, RouteDiscoveryRequests, RouteDiscoveryRequests,
          RoutingTable, Nonce, Certificate, PrivateKey,
```

```
          PublicKey -> Node;
Operations
get_identity: Node -> Identity;
...
Axioms
get_identity(node($i, $rdr, $rp, $rt, $n, $c, $priv, $pub))=$i;
Variables
i : Identity;
...
```

All the elements used by the generator for the AADT *Node*, are other AADTs that define (in this order): the identity of the node, a list with the route discovery requests that were already broadcasted, a list with the route discovery responses that were already forwarded, a lists with the routes, the current value for the nonce used in the messages, the certificate of the node, the private key of the node, and the public key of the certification authority that issues certificates for the nodes.

3.4 Modeling the topology

An ad hoc network can be defined as a graph. We have assumed the connections are bidirectional, so the graph is an undirected one. The nodes of the graph are the nodes of the ad hoc network, and the arcs represent the fact that two nodes can communicate directly through their wireless devices. So the topology of an ad hoc network can be represented as a graph. We modeled it as the AADT *Topology*, which is in fact a list of pairs of node identities, and represents the arc list that defines the graph.

The actual topology is a variable of the type *Topology*. Its value can be given in two different ways. Depending on the type of properties that will be verified, the first or the second approach will be preferred. The first way is to give the value explicitly. In this case, the model will represent the exact ad hoc network that has that topology. For example, the topology of the ad hoc network given in Fig. 2, will be defined by the next term:

```
cons(pairIdentityIdentity(i(i0), i^2(i0)),
cons(pairIdentityIdentity(i^2(i0), i(i0)),
cons(pairIdentityIdentity(i^2(i0), i^3(i0)),
cons(pairIdentityIdentity(i^3(i0), i^2(i0)),
cons(pairIdentityIdentity(i^3(i0), i^4(i0)),
cons(pairIdentityIdentity(i^4(i0), i^3(i0)),
cons(pairIdentityIdentity(i^3(i0), i^5(i0)),
cons(pairIdentityIdentity(i^5(i0), i^3(i0)), empty))))))))
```

The second way is to not assign any value to the variable. This way it will be a free variable. Then, with the use of domain unfolding, AlPiNA will generate for that variable all the possible values within a given range. We will next explain how this works and the impact of such a choice.

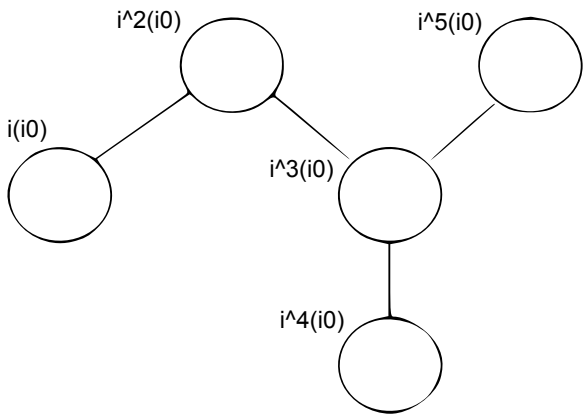


Fig. 2. An example of an ad hoc network topology

3.5 Using unfolding to model topology

Unfolding is used for the verification process in order to let the user define the part of the domain of a data type that will be taken into consideration when the state space is computed. For example, in our model, the *Identity* AADT is used for the identification of nodes. So when a certain operation must be done for all the nodes in the network, that operation is parameterized with a variable of type *Identity* for which no value is specified. Then the type *Identity* is unfolded to the number of nodes in the network. As a result, prior to building the state space, AlPiNA will unfold the Petri net by considering for that *Identity* variable all the possible values, up to the number of nodes in the network. Let us show how we can use this technique to model the topology of the ad hoc networks.

Topology AADT is actually a list of pairs of identities. Each pair of identities represents a direct connection in the ad hoc network and it is defined by the AADT *PairIdentityIdentity*. So the definition of the type *Topology* is based on the type *PairIdentityIdentity*, which is based on the type *Identity*. As a result, in order to unfold *Topology*, one needs to unfold also the other two types. Unfolding of a data type is specified by the name of the type, and the limit that will be considered for the domain. Here is an example of unfolding specification for *Topology* and for its dependencies.

```
Identity : TOTAL;
PairIdentityIdentity : TOTAL;
Topology : 3;
```

The type *Identity* is unfolded to the number of nodes in the network; the type *PairIdentityIdentity* is totally unfolded. That means that all the possible pairs that can be created with the identities of the nodes in the network will be taken into consideration. *Topology* is then unfolded to the desired depth. For example, if the bound is set to 3, AlPiNA will take into consideration all the lists

with three pairs that can be constructed with the pairs obtained by unfolding *PairIdentityIdentity* type. This way, we have actually defined all the topologies that a network can have with the given number of nodes, and in which there are three nodes which can communicate directly.

The number of topologies that will be taken into consideration in a non deterministic way through the above unfolding mechanism depends on the number n of nodes in the network, and on the number m of direct connections between them. This value represents the number of combinations of pairs that can be formed with n identities, taken m at a time. As the values for n and m increase, this value is rapidly increasing too. Unfortunately, the topology of the network cannot be abstracted, nor parameterized because of the way message exchange is done in wireless networks. In the case of a broadcast, the nodes which should receive the message can be determined only from the topology. Likewise, in the case of unicast or multicast, the topology is the only information regarding the fact that a node should receive the message or not. In conclusion, the topologies have to be taken into consideration explicitly.

Let us consider an example. If the ad hoc network has three nodes: A, B, and C, it means that for Identity all these three values will be considered. Next, because *PairIdentityIdentity* is totally unfolded, the following values will be considered for it: AB, AC, BA, BC, CA, and CB. As a result, Topology can have the following values:

```
(1) {},
(2) {AB}, {AC}, {BA}, {BC}, {CA}, {CB},
(3) {AB, AC}, {AB, BA}, {AB, BC}, {AB, CA}, {AB, CB},
...,
(4) {AB, BA, BC}, {AB, BA, CA}, {AB, BA, CB},
...
```

With (1) we consider the topology in which none of the nodes have direct wireless connections. With (2) we consider the possible topologies in which only two nodes can communicate directly, the third one being outside the communication range with each of the other two. With (3) we consider the possible topologies in which there are two groups of two nodes which can reach each other. And with (4) we consider all the topologies in which there are three groups of two nodes which can communicate with each other.

If the same value is considered for the topology for a whole protocol run, it means that after considering all these values, the protocol will be verified for all the possible topologies for three nodes. When different values are consider successively in the same protocol run, it means that the protocol is verified over a dynamic topology. So because of the fact that the order in which each of these values is considered is non deterministic, the verification will be made for all the topologies and for all the possible node movements in each of the topologies.

Because it is a list, *Topology* is an infinite data type. So unfolding its entire domain is impossible. But AlPiNA allows the partial unfolding up to a given bound on the number of elements, as we explained above. It is important to state that this second way of defining the topology of the network is particular

to AlPiNA and it works thanks to a special characteristic of the verification algorithm called partial net unfolding. Partial net unfolding means that it is not mandatory to unfold all the types, and the user can choose only the type that it needed to be unfolded ([1]).

When the topology is defined as a closed term, AlPiNA will compute the state space for the given algebraic Petri net N, starting from the initial marking. If M_0 is the initial marking, then the state space computed for a given topology can be written as:

$St_N(M_0)$.

When the topology is defined by unfolding, the algebraic Petri net is parameterized by a free variable of type Topology. If $\$tp$ is the name of this variable, then the parameterized algebraic Petri net can be written as:

$N(\$tp)$.

By unfolding, AlPiNA will instantiate the variable $\$tp$ with each of the possible values of the topology, as explained above, thus computing a set of algebraic Petri nets, one for each value:

$N = \cup_{x \in T_{\Sigma,Topology}} N(x)$.

When computing the state space, AlPiNA will actually compute the set of state spaces such that each state space corresponds to a value for the topology. We can write this as follows:

$St_N(M_0) = \cup_{x \in T_{\Sigma,Topology}} St_{N(x)}(M_0)$.

As it will be presented in section 4, the security properties that we have model checked with AlPiNA were expressed through an invariant property. In order to check such a property, AlPiNA starts by computing the state space of the algebraic Petri net provided as input. Then, it checks if the specified property is true for each of the states. If it is, then the property holds for the model. If not, the property does not hold for the model, and a counter-example is provided.

If the topology is defined as a closed term, checking a property for the model implies checking the property for the state space computed for the corresponding APN.

$St_N(M_0) \models invariantproperty$

If the topology is defined by unfolding, checking a property for all models implies checking it for the set of state spaces generated by instantiating the topology variable with all the possible values.

$St_N(M_0) \models invariantproperty \Leftrightarrow$

$\cup_{x \in T_{\Sigma,Topology}} (St_N(x)(M_0) \models invariantproperty)$

So we will check the invariant on all instances; finding a contradiction will mean there is one topology that contradicts the invariant. If the invariant is satisfied on the whole model it means that it is obviously satisfied in each instance.

3.6 Modeling the network

The message exchange in an ad hoc network has special characteristics, because all the nodes act like routers. When a node transmits a message, it is received only by the nodes which have a direct connection to that node. Then, each of the nodes which received the message, processes it according to the routing

protocol, and then retransmits it. This process continues until the message gets to the destination. Another aspect that must be taken into consideration is the fact that messages can be of unicast or broadcast type. If a message is unicast, it will be processed only by the node to which it is destined. If a message is broadcast, it should be processed by all the nodes which can receive it directly according to the topology of the network.

The messages transmitted by all the nodes are stored in the place called *Transmitted Packets* (Fig. 3). The network processes the messages from this place and then stores them in the place called *Received Packets* (Fig. 3), from where the nodes can take them for processing and so on.

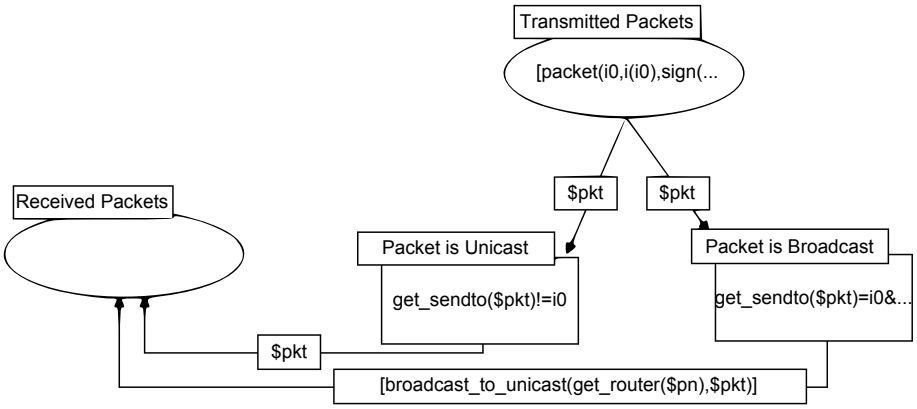


Fig. 3. The model for the ad hoc network operation

In order to have in the High level Petri net model the behavior presented above, we need to model accordingly two elements: the format of the messages exchanged by the nodes and the network itself. Regarding the format of the messages, besides the fields that a message has according to the considered routing protocol, we added two extra fields: a field that stores the identity of the previous node that transmitted it (*prev*), and a field that represents the identity of the node which should receive the message (*next*). If next field contains the value *i0*, then it means that the message is broadcast. Otherwise the message is unicast. The structure of the AADT *Packet* is provided in Fig. 4.

The modeling of the transmission/reception of a message is given in Fig. 3. All the messages transmitted by the nodes are stored in the place called *Transmitted Packets*. From here they are processed in order to provide the behavior explained in the previous paragraph. First we check if the message is unicast or broadcast. If it is unicast, no other processing is required (transition *Packet* is unicast) so the message is placed in the *Received Packets* place from where the destination node can pick it up for processing.

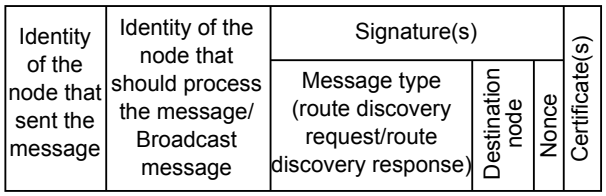


Fig. 4. The model for the ARAN messages

If the message is broadcast (transition *Packet* is broadcast), we search in the topology for all the identities of the nodes which can receive the message according to it, and we produce the same number of copies for the message, but with the next field filled with the corresponding identity. To verify in the APN if a certain node with identity i can receive a message, we search the variable of type *Topology* if it contains a pair of identities formed by the identity stored in *prev* and by i.

It is worth mentioning that this model of broadcast has an atomicity problem caused by some limitations of the Petri nets. Unfortunately there is no better way of modeling it with the current formalism. The problem is the fact that all the copies of the broadcasted message should reach all the destination nodes at the same time, as if they would be produced in the same transition. This is not possible to model, so, as a result, given the non determinism of the Petri net, other transition could be fired before all the copies reach the destination nodes. This could be solved by an extension of the Petri net, as the one proposed in [20]. The LLAMAS (Language for Advanced Modular Algebraic Systems) model proposed here is based on the old ideas of CO-OPN and it uses synchronization between the transitions in order to provide a better control of the atomicity. By using such synchronization it would be possible to force the correct transmission of a broadcast message by preventing any other transition to fire before the transition that handles the broadcast fires all the possible times. Such a mechanism will also have an impact over the combinatorial explosion by eliminating possibilities that have no meaning in the real ad hoc networks.

3.7 Modeling the adversary

The model that we used for the adversary was the Dolev-Yao model ([21]). In this model it is presumed that the adversary can perform the following operations:

- he can intercept all the messages transmitted in the network (1);
- he can generate new messages based on the knowledge he obtained from the intercepted messages (2);
- he can transmit messages (without modifying them) in the name of any node in the network (3);
- he can prevent a node from receiving a message that was meant for it, with the purpose of sending it another message (4).

Due to the state space explosion problem, we were unable to fully implement this kind of adversary in our model. We have only implemented attack types (1), (3) and (4). To implement attack type (1), the adversary was modeled as having access to all the messages exchanged in the network (places *Transmitted Packets* and *Received Packets* in Fig. 3). Thus he can perform the following actions over the messages: he can drop a message and thus preventing a node to receive it (implementation of attack type (4)) with the purpose of replacing the dropped message with another one, and he can retransmit a message (without modifying it) to another node than the node it was meant for (attack type (3)).

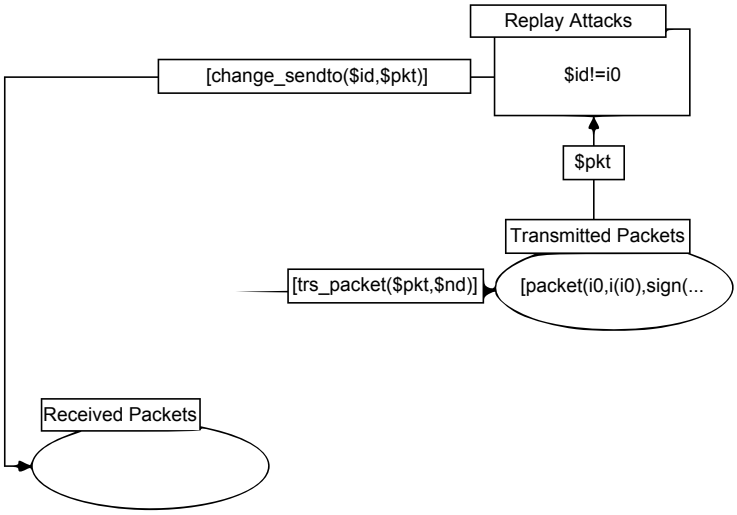


Fig. 5. The model for the adversary

As a consequence, cryptographic security properties like authentication, confidentiality and integrity cannot be checked. Correctness properties can be checked and we will present how in section 4.

3.8 ARAN operation

When modeling ARAN, we have focused on the most important part of the protocol which is the route discovery. As one can see from Fig. 6, the behavior of a node that participates in a route discovery process was modeled with two transitions. The transition *REP Packet at source* corresponds to the fact that the node that initiated the route request receives the response message. The transition *Packet processing* corresponds to all the other processing that a node has to do: broadcast of a route request message by an intermediary node, reception of the route request message by the targeted node, validation of the

digital signature(s) from the message, response to a route request message by the destination node, and the unicast of a response to a route discovery message. The actual behavior is implemented by axioms in the AADTs that define the nodes, the messages, the certificates, and the cryptographic operations. The conceptual difference between the two transitions is the presence of the place called *Witness Nodes I*. The purpose of *Witness Nodes I* will be explained in the following paragraph.

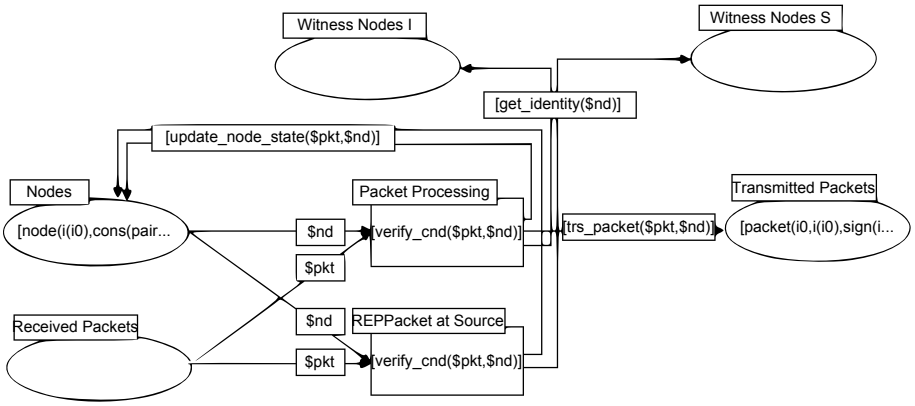


Fig. 6. The model for the node behavior in ARAN

4 Verification of security properties for ARAN

The security objectives of ARAN are to provide authentic and correct routing information for the nodes that issue a route request. Thus, the security properties that have to be verified are authentication of the nodes which participate in the route discovery, and integrity and correctness of the exchanged routing information. ARAN was already modeled and verified using different tools, and we will only cite the latest paper on the subject, [22]. ARAN is successful in assuring authentication and integrity, but an intruder can disturb it by replaying attacks and can propagate incorrect information about the topology of the network. In order to validate our method of modeling using AlPiNA, we wanted to see if we will obtain the same results as the ones already reported by previous research.

The security property that we have verified is correctness of routing information. Authentication and integrity were not considered for reasons explained in section 3 and there are no known attacks against these objectives.

To present what correctness of routing information means, let us consider the topology presented in Fig. 2. If *i(i0)* is the initiator node, and *i^5(i0)* is the

destination node, then the expected path between them that should be returned by the protocol is: *i(i0), i^2(i0), i^3(i0), and i^5(i0)*. In this case, we say that the protocol provided correct routing information, if and only if for each route discovery request made by node *i(i0)* for node *i^5(i0)*, the protocol will always return the above path. In all the other cases the routing information would not be correct.

In order to verify routing information correctness, we reduced the model of the intruder so that he will only use the possibility of replay attacks. Also, we added to the Petri net the places *Witness Nodes I*, and *Witness Nodes S*. Their role will be presented next. Each time an intermediary node along the routing path from the source node to the destination node processes a message related to the discovery process, its identity is stored in *Witness Nodes I*. The same thing will happen for the destination node too: when it will respond to the route discovery, its identity will be stored in *Witness Nodes I*. In the same manner, when the source node, the node that initiated the route discovery request, will receive the response from the destination node, its identity will be copied to the place called *Witness Node S*.

In the initial marking of the Petri net, the place called *Transmitted Packets* contains a route discovery message from node *i(i0)* for the destination *i^5(i0)*. The places *Witness Nodes S* and *Witness Nodes I* are empty. When generating the state space of the model, the place *Witness Nodes S* will eventually contain the identity of the source node *i(i0)*. This will mean the protocol run has finished, and the route to the destination was obtained. The identities of the nodes forming the returned route will be in the place *Witness Nodes I*.

To verify the correctness of the routing information, we need to compare the identities of the nodes from *Witness Nodes I* place with the identities of the nodes from the actual path in the considered topology. Using the property specification language available in AlPiNA, we have specified this property in the following way: If the number of nodes in the place *Witness Nodes S* is equal to one it implies that the number of nodes in the place *Witness Nodes I* is equal to the number of nodes in the path from the considered topology. Here is the specification of this property in AlPiNA's property specification language:

```
(card($x in WitnessNodesS) = 1 ) =>
       (card($y in WitnessNodesI) = value );
```

If the property holds when model checking is performed it means the protocol provided correct routing information. Otherwise, the routing information is incorrect and AlPiNA will display a counter-example: content for the place *Witness Nodes I* that contains a different number of nodes. Based on this counter-example we can reconstitute the attack performed by the intruder.

After performing the model checking we have seen that the protocol does not always provide correct routing information, meaning that the intruder was able to mount an attack on it (in concordance with [22]).

Returning to the example we have considered when explaining how the verification is done, when model checking the protocol for this topology, the place

Witness Nodes I, contains *{i^2(i0)}*, or *{i^3(i0)}*, or *{i^5(i0)}*, or *{i^2(i0), i^3(i0), i^5(i0)}*. Only the last value for *Witness Nodes I* corresponds to a correct run of the protocol. The other values represent incorrect routing information that the intruder manages to propagate in the network by replaying attacks. For example, if place *Witness Nodes I* contains *{i^5(i0)}*, it means that the intruder managed to replay the route discovery message sent by *i(i0)* to *i^5(i0)*, and prevented node *i^2(i0)* from receiving it. In this way *i^5(i0)* believes it has a direct connection with A, and responds accordingly. The intruder does the same with the route response message from *i^5(i0)*.

Table 1. Quantitative information

Tool name	Tool's performance for ARAN		
	Number of nodes	*Time (s)*	No of states
AlPiNA	4 (all nodes attacked)	0.95	436
	5 (all nodes attacked)	3.70	4655
	6 (all nodes attacked)	80.82	77239
	7 (6 nodes attacked)	110.95	79131
	8 (5 nodes attacked)	20.22	11637
	9 (5 nodes attacked)	32.92	15500
	10 (5 nodes attacked)	44.06	19363
AVISPA	4	0.05	-
	5	0.07	-

The table above presents quantitative information regarding the verification of routing information correctness, as previously described, in comparison with another model checker called AVISPA, used in [22], where the authors reported the same verification results as we have. The variable of the runs is the number of nodes, besides the adversary, in the topology of the ad hoc network that is taken into consideration. For some of the cases, the tool was unable to compute the state space for all the possible attacks. So we limited the number of nodes which were attacked to some maximum value, which is provided in the table between parentheses, in the same cell as the number of nodes.

AVISPA uses an on-the-fly model checking technique in which attacks are searched for without a prior computation of the whole state space. On the contrary, AlPiNA first computes the entire state space in a symbolic manner, and only then makes the search for attacks. As a consequence, the values provided for AVISPA represent the time of finding the replaying attack for the considered specification, while in the case of AlPiNA, the time column represents the time of computing the entire state space of the considered model. These values cannot be directly compared, but they reveal the fact that AlPiNA is capable of

handling the whole state space of the specifications verified with AVISPA, but with the limitation explained above. AlPiNA is capable of handling state spaces of 1-2 millions of states, but in this case, because of the atomicity problem presented at the end of subsection 3.6, starting with 7 nodes, all being attacked, the size of the state spaces goes directly to more millions of states than AlPiNA can handle. This is the reason of using these limitations and also the reason for the fact that the biggest size of the state space in the table is a little less then 80000.

In [22], the authors state they were unable to check the protocol for more than four and five nodes respectively, because of the state space explosion. But using AlPiNA, we managed to model check the protocol for 10 nodes.

5 Conclusions and Future Work

In this paper we have presented the use of algebraic Petri nets for modeling ad hoc networks and for verifying correctness properties for security protocols specially designed for this type of networks, with the use of AlPiNA, a symbolic model checker based on APNs. As far as we know this is the first report of using Petri nets for verifying security properties of the protocols designed for ad hoc networks.

As one can see from the figures we have provided, the Petri net that models the ad hoc network and the security protocol is very simple and clear and has a very small number of places. For example, the model for ARAN has six places. The heavy part of the model is represented by the AADTs that were defined. Thus AlPiNA combines the powerful symbolic model checking with the easy to use APN formalism, providing a good user experience, but also with the ability to master state space explosion.

The limitation of our approach refers to the fact that fabrication attacks were not considered. Fabrication refers to the ability of the intruder to create and send new messages, based on what he previously learned from the network. Our model for the adversary is capable of using the messages he learned from the network, but cannot create new messages. Because it is a symbolic model checker, when an attack is found, AlPiNA cannot provide attack traces. This makes it very difficult to model fabrication attacks, because of the lack of feedback from the tool. But we plan to address this limitation by developing a technique for inversing transitions in an APN, and thus providing attack traces and the necessary feedback.

The model and the verification performed for ARAN secure routing protocol discovered all the attacks that were previously reported for this protocol. This proves the validity of the method, but most importantly, it proves that AlPiNA can be used with success for verifying security protocols.

As future work, we have proposed to perform a quantitative comparison between CPN Tools and AlPiNA in order to see the actual performance improvement brought by the latter. Also we will work on proposing an extension to the current APN model, that will be more adequate to the modeling of distributed

protocols, in general, and which, in particular, will be capable of handling broadcast and similar operations in a correct manner. Another future work direction is to modify the modeling of the topology, such that equivalent topologies will be eliminated from the verification, thus reducing the state space and increasing the performance of the model checking.

References

1. Steve Patrick Hostettler, High-level Petri net model checking: the symbolic way, PhD thesis, University of Geneva, 2011.
2. Yongyuth Permpoontanalarp, Panupong Sornkhom, A New Colored Petri Net Methodology for the Security Analysis of Cryptographic Protocols, in The 10th Workshop and Tutorial on Practical Use of Colored Petri Nets and the CPN Tools, Denmark, pp. 81-100. 2009.
3. Didier Buchs, Steve Hostettler, Alexis Marechal, Matteo Risoldi, Alpina: A symbolic model checker, Applications and Theory of Petri Nets, pp. 287-296, 2010.
4. Steve Patrick Hostettler, Alexis Marechal, Alban Linard, Matteo Risoldi, Didier Buchs, High-Level Petri Net Model Checking with AlPiNA, Fundamenta Informaticae, IOS Press, Amsterdam, The Netherlands, vol. 113, no. 3-4, August 2011, ISSN, 0169-2968, pp. 229-264, 2011.
5. AlPiNA tool web page, http://alpina.unige.ch/, the 23 of December 2013.
6. Kimaya Sanzgiri, Bridget Dahill, A Secure Routing Protocol for Ad Hoc Networks, Proceedings of the 10th IEEE International Conference on Network Protocols, pp. 78-87, 2002.
7. L. Khoukhi, S. Cherkaui, Intelligent Solution for Congestion Control in Wireless Ad hoc Networks, in WONS 2006: Third Annual Conference on Wireless On-demand Network Systems and Services, pp. 10-19. 2006.
8. Tzu-Chiang Chiang, Zueh-Min Huang, Multicast Routing Representation in Ad Hoc Networks Using Fuzzy Petri Nets, Proceedings of the 18th International Conference on Advanced Information Networking and Application, vol. 2, pp. 420, 2004.
9. Congzhe Zhang, Mengchu Zhou, A Stochastic Petri Net Approach to Modeling and Analysis of Ad Hoc Network, in Proceedings of the International Conference on Information Technology: Research and Education, pp. 152-156, 2003.
10. Marco Beccuti, Massimiliano De Pierro, Andras Horvath, Adam Horvath, Karoly Farkas, A Mean Field Based Methodology for Modeling Mobility in Ad Hoc Networks, in Vehicular Technology Conference (VTC Spring), 2011, IEEE 73rd, pp. 1-5, 2011.
11. Piyush Prasad, Baltej Singh, Asish Kumar Sahoo, Validation of Routing Protocol for Mobile Ad Hoc Networks using Colored Petri Nets, bachelor thesis, National Institute of Technology, Rourkela, 2009.
12. Chaoyue Xiong, Tadao Murata, Jeffery Tsai, Modeling and Simulation of Routing Protocol for Mobile Ad Hoc Networks using Colored Petri Nets, Proceedings of the Conference on Application and Theory of Petri Nets: Formal Methods in Software Engineering and De-fence Systems, vol. 12, pp. 145-153, 2002.
13. Mohammad Ali Jabraeil Jamali, Tahere Khosravi, Validation of Ad Hoc On-demand Multipath Distance Vector Using Colored Petri Nets, International Conference on Computer and Software Modeling, Singapore, vol. 14, pp. 29-34, 2011.

14. Kristian L. Espensen, Mads K. Kjeldsen, Lars M. Kristensen, Modeling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks, Applications and Theory of Petri Nets: 29 International Conference, Lecture Notes in Computer Science Volume 5062, pp. 152-170, 2008.
15. Yongyuth Permpoontanalarp, Apichai Changkhanak, Security Analysis of the TMN Protocol by Using Colored Petri Nets: On-the-fly Trace Generation Method and Homomorphic Property, the 8th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 63-68, 2011.
16. Yang Xu, Modeling and Analysis of Security Protocols Using Colored Petri Nets, Journal of Computers, vol. 6, no. 1, pp. 19-27, 2011.
17. Ulrike Golas, Kathrin Hoffman, Hartmut Ehrig, Alexander Rein, Julia Padberg, Functional Analysis of Algebraic Higher-Order Net Systems with Applications to Mobile Ad-Hoc Networks, Bulletin of the EATCS, no. 101, pp.148-160, June 2010.
18. J. Padberg, H. Ehrig, L. Ribeiro, Formal Modeling and Analysis of flexible Processes in mobile ad-hoc networks, Bulletin of the EATCS, pp. 128-132, 2007.
19. Hartmut Ehrig, Bernd Mahr, Fundamentals of Algebraic Specification 1: Equations and Initial Semantics, Monographs in Theoretical Computer Science, An EATCS Series, Springer, 1985.
20. Alexis Ayar Marechal Marin, Unifying the syntax and semantics of modular extensions of Petri nets, PhD thesis, University of Geneva, 2013.
21. Danny Dolev, Andrew Yao, On the Security of Public Key Protocols, IEEE Transactions on Information Theory, vol. IT-29, nr.2, pp. 198–208, 1983.
22. Davide Benetti, Massimo Merro, Luca Vigano, Model Checking Ad Hoc Network Routing Protocols: ARAN vs. endairA, The 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 191-202, 2010.

Part III

PNSE'14: Short Presentations

Morphisms on Marked Graphs

Luca Bernardinello, Lucia Pomello, and Stefano Scaccabarozzi

Dipartimento di Informatica, Sistemistica e Comunicazione,
Università degli studi di Milano - Bicocca,
Viale Sarca, 336 - Edificio U14 - I-20126 Milano, Italia
luca.bernardinello@unimib.it

Abstract. Many kinds of morphisms on Petri nets have been defined and studied. They can be used as formal techniques supporting refinement/abstraction of models. In this paper we introduce a new notion of morphism on marked graphs, a class of Petri nets used for the representation of systems having deterministic behavior. Such morphisms can indeed be used to represent a form of abstraction on marked graphs, consisting in folding cycles and identifying chains. We will then prove that systems joined by these morphisms show behavioral similarities.

Keywords: Petri nets, marked graphs, morphisms, model abstraction, preservation of behavioral properties

1 Introduction

When working on concurrent and distributed systems, the dimensions and complexity of a model may lead to difficulties in the analysis of its features and properties. For this reason it is useful to have formal techniques allowing the decomposition of the entire model into separate modules which can be studied separately, then being recomposed maintaining their properties. Another way to reduce the dimension and complexity of a model is to use a multilevel approach to its analysis: we start working on a very abstract version of the model, then proceed through different levels of refinement by adding details to the model.

In order to obtain such functionalities we can use morphisms on Petri nets. In the literature (see, for example, [1], [2], [3], [4] and [5]) several kinds of morphism on different classes of Petri nets have been introduced. In this paper we propose a new definition of morphism on marked graphs, a class of Petri nets often used for representing systems having deterministic behavior. These so called F-morphisms and the subclass of $\hat{F}$-morphisms constitute a formal instrument which can be used to obtain a kind of abstraction of marked graphs.

Some kinds of morphisms defined in the literature, such as α-morphisms ([5]), allow to collapse part of the initial model on a single place or a single transition in order to obtain the abstract system. Differently, $\hat{F}$-morphisms map places on single places and transitions on single transitions, preserving the environment of each mapped element. Instead of collapsing portions of the detailed model into a single element, the abstraction is here obtained by "folding" cycles and

identifying chains and cycles. Both these elements still remain in the reduced model.

Such kind of abstraction preserves the behavior of the mapped part of the original system. This means that, whenever we apply a $\hat{F}$-morphism on a system, all the sequences of actions executable in the reduced version can be found in the original model.

In the last part of this paper, an analysis of preserved and reflected behavioral properties and invariants of marked graphs joined by $\hat{F}$-morphisms is performed.

In the next section, basic definitions related to Petri nets and their unfoldings are recalled. In Section 3 F- and $\hat{F}$-morphisms are introduced together with their main features. Then the relationship between the unfoldings of two marked graphs joined by a $\hat{F}$-morphism is explicated. Section 4 shows the results of the analysis of behavioral and structural properties preserved and reflected by $\hat{F}$-morphisms. The paper is closed by a short concluding section.

2 Preliminary definitions

In this section we recall basic definitions about marked graph theory and unfoldings. These notions will be used in the next chapters to study important aspects of F-morphisms.

2.1 Petri nets

We first start introducing the notion of net as seen in [6], with some adjustements.

Definition 1. *A net is a triple $N = (S, T, F)$, where*

- *S is a set of places,*
- *T is a set of transitions such that $S \cap T = \emptyset$,*
- *F is a set of directed arcs (flow relation), $F \subseteq (S \times T) \cup (T \times S)$.*

All places and transitions are said to be *elements* of N. A net is *finite* if the set of elements is finite.

For an element x of $S \cup T$, its *pre-set* is defined by

$$^\bullet x = \{y \in S \cup T \mid (y, x) \in F\}$$

while its *post-set* is defined by

$$x^\bullet = \{y \in S \cup T \mid (x, y) \in F\}.$$

A *directed path* (*path* for short) in a net N is a nonempty sequence $x_0 \dots x_k$ satisfying $x_i \in x_{i-1}^\bullet$ for each i $(1 \leq i \leq k)$. We say that this path *leads* from x_0 to x_k. The net is *strongly connected* if for each two elements x and y there exists a directed path leading from x to y.

An *undirected path* is a nonempty sequence $x_0 \dots x_k$ of elements satisfying $x_i \in {}^\bullet x_{i-1} \cup x_{i-1}^\bullet$ for each i $(1 \leq i \leq k)$. Such undirected path *leads* from x_0 to

x_k. The net is *weakly connected* if, for each two elements x and y, there exists an undirected path leading from x to y. In this paper, we will call *connected* a weakly connected net.

A *directed circuit* is a directed path $x_0 \dots x_k x_0$ such that, for each $i, j \in \mathbb{N}$, $i, j \leq k$, $i \neq j$, $x_i \neq x_j$ holds.

The states of a Petri net are defined by its *markings*. State changes are caused by the occurrences of transitions. A *marking* of a net $N = (S, T, F)$ is a mapping $M : S \to \mathbb{N}$. A place $s \in S$ is *marked* by a marking M if $M(s) > 0$.

A transition t is *enabled* at a marking M if M marks every place in ${}^\bullet t$. Then t can *occur*. Its occurrence transforms M into the marking M', defined for each place s as

$$M'(s) = \begin{cases} M(s) - 1 & \text{if } s \in {}^\bullet t \setminus t^\bullet, \\ M(s) + 1 & \text{if } s \in t^\bullet \setminus {}^\bullet t, \\ M(s) & \text{otherwise.} \end{cases}$$

In this case we write $M \xrightarrow{t} M'$. Notice that a place in ${}^\bullet t \cap t^\bullet$ is marked whenever t is enabled but does not change its token count by the occurrence of t. A marking is called *dead* if it enables no transition of N. A net N together with an *initial marking* M_0 constitutes a *Petri Net System* (also called *place/transition system*), denoted (N, M_0).

Let M be a marking of a net. A finite sequence $t_1 \dots t_k$ of transitions is called a *finite occurrence sequence, enabled at* M, if there are markings $M_1, \dots, M_k$ such that

$$M \xrightarrow{t_1} M_1 \xrightarrow{t_2} \dots \xrightarrow{t_k} M_k.$$

In this case we write $M \xrightarrow{\omega} M_k$, where $\omega = t_1 \dots t_k$. The empty sequence $\mathcal{E}$ is enabled at any marking M and satisfies $M \xrightarrow{\mathcal{E}} M$. A marking M' is said to be *reachable* from a marking M if there exists a finite occurrence sequence ω such that $M \xrightarrow{\omega} M'$.

In this paper we will mainly work on a particular kind of Petri nets, the *marked graphs*.

Definition 2. *A Petri net $N = (S, T, F, M_0)$ is a marked graph if, for every $s \in S$, $|{}^\bullet s| \leq 1$ and $|s^\bullet| \leq 1$.*

2.2 Behavioral properties

The presence of an initial marking M_0 allows to identify the *behavior* of the Petri net system (N, M_0), defined as the set of all markings reachable from M_0 together with the set of occurences of each transition which make the global state of the system change.

Properties of a net depending on the initial marking are known as *behavioral properties* of the net. We now introduce some behavioral properties ([7]) which will be used in the next sections.

Definition 3. *A Petri net (N, M_0) is said to be k-bounded or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e., $M(s) \leq k$ for every place s and every reachable marking M. (N, M_0) is said to be safe if it is 1-bounded.*

While boundedness implies the presence of a finite number of global states for a finite net, *liveness* ensures that every event can potentially occur in the future.

Definition 4. *A Petri net (N, M_0) is said to be live (or equivalently M_0 is said to be a live marking for N) if, no matter which marking has been reached from M_0, it is possible to ultimately fire any transition of the net by progressing through some further firing sequence.*

2.3 Incidence matrix and structural invariants

Definitions recalled in this section are taken from [7], with some adaptations.

Definition 5. *Let (N, M_0) be a Petri net with n transitions and m places. Its incidence matrix $A = [a_{ij}]$ is an $m \times n$ matrix of integers and its typical entry is given by*

$$a_{ij} = a_{ij}^{+} - a_{ij}^{-}$$

where $a_{ij}^{+} = 1$ if there is an arc of N going from transition j to its post-condition i, otherwise $a_{ij}^{+} = 0$, while $a_{ij}^{-} = 1$ if there is an arc to transition j from its pre-condition i, otherwise $a_{ij}^{-} = 0$.

Some properties of a Petri net can be studied through the incidence matrix and its invariants. A S-invariant associates weights to places in a way such that the weighted sum of tokens is the same in all reachable markings.

Definition 6. *Let N be a net and let A be its incidence matrix. A vector $\underline{I} : S \to \mathbb{Z}$ is a S-invariant for N iff it is a solution of: $\underline{I}A = \underline{0}$.*

T-invariants allow to identify possible cyclic behaviors in a Petri net.

Definition 7. *Let N be a net and let A be its incidence matrix. A vector $\underline{J} : T \to \mathbb{Z}$ is a T-invariant for N iff it is a solution of: $A\underline{J}^T = \underline{0}$.*

2.4 Branching processes and unfoldings

The *behavior* of a Petri net N can be represented in different ways. One of these is to use the so called *unfolding* of N. In order to understand what the unfolding of a net is, we first need to introduce some formal definitions. The theoretical notions we will relate in this subsection are all taken from [7]. From now on, we will only consider Petri nets such that, for every transition t, ${}^\bullet t$ and $t^\bullet$ are finite sets and, moreover, we assume them to be nonempty. Furthermore, we do not allow more than one token on a place in the initial marking. Such constraints do not result too restrictive with respect to the behavior of the studied systems.

Definition 8. *Let $N = (S, T, F, M_0)$ be a Petri net. For $x, y \in S \cup T$ we say that x precedes y if there is a (possibly empty) directed path from x to y in N. N is finitary if for every $y \in S \cup T$ the set $\{x \in S \cup T \mid x \text{ precedes } y\}$ is finite.*

The relation *precedes* defines a partial order on $S \cup T$, and $Min(N)$ is the set of minimal elements of that partial order. We now introduce the notion of *conflict*.

Definition 9. *Let $N = (S, T, F, M_0)$ be a Petri net. For $x_1, x_2 \in S \cup T$, x_1 and x_2 are in conflict, denoted $x_1 \# x_2$, if there exist distinct transitions $t_1, t_2 \in T$ such that ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$ and t_i precedes x_i, for $i = 1, 2$. For $x \in S \cup T$, x is in self-conflict if $x \# x$.*

The concept of conflict is used to define *occurrence net*.

Definition 10. *An occurrence net is a finitary acyclic net $N = (S, T, F, M_0)$ such that*

- *for every $s \in S$, $|{}^\bullet s| \leq 1$,*
- *no transition $t \in T$ is in self-conflict, and*
- *$M_0 = Min(N)$.*

We now define a particular kind of morphism called "folding" in [8]. Intuitively, a *homomorphism* from net N_1 to net N_2 formalizes the fact that N_1 can be folded onto a part of N_2, or, in other words, that N_1 can be obtained by partially unfolding a part of N_2.

Definition 11. *Let $N_i = (S_i, T_i, F_i, M_0^i)$ be nets, $i = 1, 2$. A homomorphism from N_1 to N_2 is a mapping $h : S_1 \cup T_1 \to S_2 \cup T_2$ such that*

- *$h(S_1) \subseteq S_2$ and $h(T_1) \subseteq T_2$,*
- *for every $t \in T_1$, the restriction of h to ${}^\bullet t$ is a bijection between ${}^\bullet t$ and ${}^\bullet h(t)$, and similarly for $t^\bullet$ and $h(t)^\bullet$, and*
- *the restriction of h to M_0^1 is a bijection between M_0^1 and M_0^2.*

The notions of homomorphism and occurrence net are necessary to formally define *branching processes*.

Definition 12. *Let $N = (S, T, F, M_0)$ be a net. A branching process of N is a pair (N', π), where $N' = (S', T', F', M_0')$ is an occurrence net and π is a homomorphism from N' to N, such that, for every $t_1, t_2 \in T$, if ${}^\bullet t_1 = {}^\bullet t_2$ and $\pi(t_1) = \pi(t_2)$, then $t_1 = t_2$.*

In [9], a notion of homomorphism between branching processes of the same net N is also defined. Injective homomorphisms define a partial order for the branching processes of N, called *approximation*. The set of the isomorphism classes of the branching processes of N, together with approximation, form a complete lattice. The least upper bound of such lattice is the *unfolding* of N.

3 A new class of morphisms on marked graphs

In this section we introduce a new kind of morphism on marked graphs, the *F-morphisms*. We will then focus on a subclass of such morphisms, the $\hat{F}$-*morphisms*, analysing some interesting features of theirs. Finally, we will study the relationship between the unfoldings of two marked graphs joined by a $\hat{F}$-morphism. In this paper we only consider a particular kind of marked graphs.

Remark *From now on, we only consider connected marked graphs without self-loops.*

It is now possible to introduce the main notion of this work.

Definition 13. *Let $N_i = (S_i, T_i, F_i, M_0^i)$, $i = 1, 2$, be two marked graphs. A F-morphism from N_1 to N_2 is a pair (σ, τ), where $\sigma : S_1 \to S_2$ and $\tau : T_1 \to T_2$ are partial surjective functions, such that:*

- *if $\tau(t_1)$ is undefined, then $\sigma({}^\bullet t_1) = \emptyset = \sigma(t_1^\bullet)$,*
- *if $\tau(t_1) = t_2$, then the restriction of σ to ${}^\bullet t_1$ is an injective and surjective partial function from ${}^\bullet t_1$ to ${}^\bullet t_2$ and, similarly, the restriction of σ to $t_1^\bullet$ is an injective and surjective partial function from $t_1^\bullet$ to $t_2^\bullet$,*
- *for every $s' \in S_2$*

$$M_0^2(s') = \sum_{s \in \sigma^{-1}(s')} M_0^1(s).$$

We define the composition of two F-morphisms $(\sigma_1, \tau_1) : N_1 \to N_2$ and $(\sigma_2, \tau_2) : N_2 \to N_3$ by using the notion of composition of functions, i.e., $(\sigma_1, \tau_1) \circ (\sigma_2, \tau_2) = (\sigma_2 \circ \sigma_1, \tau_2 \circ \tau_1) : N_1 \to N_3$. F-morphisms are closed by composition.

Theorem 1. *Let $N_i = (S_i, T_i, F_i, M_0^i)$ be marked graphs for $i = 1, \ldots, 3$. Let (σ_i, τ_i), $i = 1, 2$, be F-morphisms from N_i to N_{i+1}. The function $(\sigma, \tau) : N_1 \to N_3$, where $\sigma = \sigma_2 \circ \sigma_1$ and $\tau = \tau_2 \circ \tau_1$ is a F-morphism.*

This theorem is proved in [10]. The identity function $1_N = (id_S, id_T)$ is a F-morphism, where $id_S : S \to S$ and $id_T : T \to T$ are the total identity functions. The composition is associative. Hence, the family of F-morphisms, together with marked graphs, form a category which takes the name of *Marked Graph System*, denoted $\mathcal{MGS}$.

With these morphisms we allow to map chains on cycles, as shown in Figure 1, representing an example of F-morphism from N_1 to N_2. The labels suggest the arrows of the morphism. Notice that the cardinality of the pre-images of the elements labelled by `1`, `b` and `2` of N_2 is one, while the place labelled by `ac` has two elements in its pre-image.

By adding a further constraint to the definition of F-morphisms, we get a subclass of morphisms which preserve cycles and chains.

Definition 14. *Let $N_i = (S_i, T_i, F_i, M_0^i)$ be marked graphs for $i = 1, 2$. A $\hat{F}$-morphism from N_1 to N_2 is a F-morphism (σ, τ) with the following restriction:*

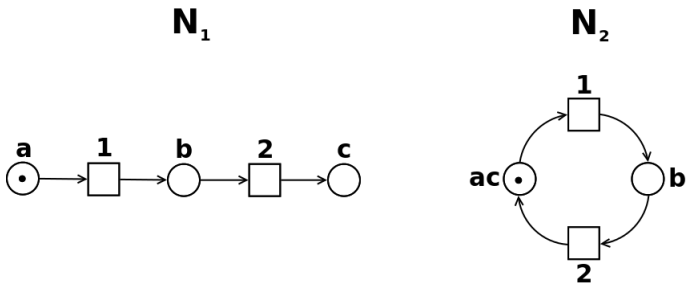


Fig. 1

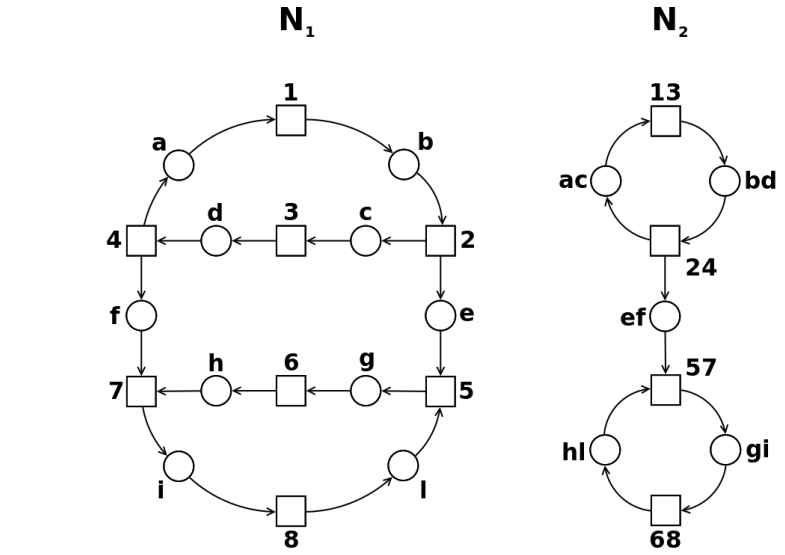
– *for all* $s_1 \in S_1$ *such that* $\sigma(s_1) = s_2$*, the restriction of* τ *to* ${}^\bullet s_1$ *is a bijection from* ${}^\bullet s_1$ *to* ${}^\bullet s_2$ *and, similarly, the restriction of* τ *to* $s_1^\bullet$ *is a bijection from* $s_1^\bullet$ *to* $s_2^\bullet$*.*

It is easy to see that $\hat{F}$-morphisms are closed by composition. In fact, since we already know that a $\hat{F}$-morphism (σ, τ) is a F-morphism, it is sufficient to prove that the additional constraint that characterizes $\hat{F}$-morphisms is preserved by composition. We prove it simply by observing that the composition of two bijections is also a bijection.

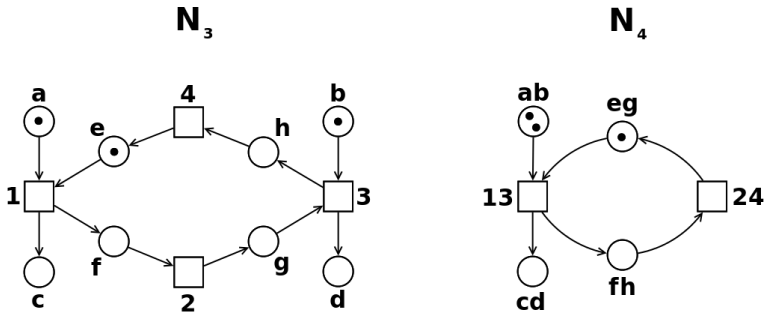
The example in Figure 1 shows a F-morphism (σ, τ) which is not a $\hat{F}$-morphism: let s_1 be the place of N_1 labelled with `c` and let $\sigma(s_1) = s_2$ (therefore, s_2 is the place of N_2 labelled with `ac`). The restriction of τ to $s_1^\bullet$ is not a bijection from $s_1^\bullet$ to $s_2^\bullet$, in fact we have that $s_1^\bullet = \emptyset \neq s_2^\bullet$.

In Figure 2 three examples of $\hat{F}$-morphisms are shown: the first two of them, $((\sigma_1, \tau_1) : N_1 \to N_2$ and $(\sigma_2, \tau_2) : N_3 \to N_4$, respectively, Figure 2a and Figure 2b), allow us to observe that, using $\hat{F}$-morphisms, it is possible to compress cycles and to identify chains; in the last one, $((\sigma_3, \tau_3) : N_5 \to N_6$, Figure 2c), an identification of cycles is represented.

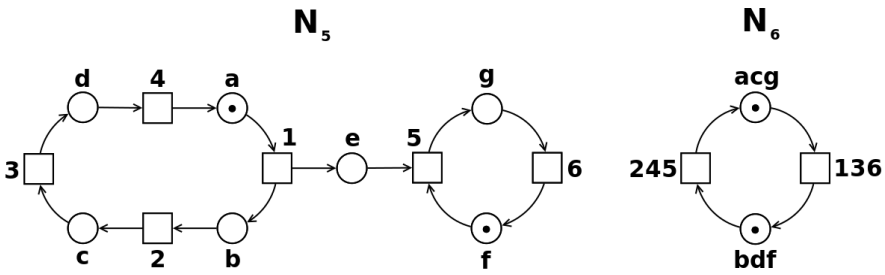
Let us now compare $\hat{F}$-morphisms with another kind of morphisms defined in [2], N-morphisms, corresponding to a kind of partial simulation. We want to do this since we will later show that we can always find a N-morphism between the unfoldings of two marked graphs joined by a $\hat{F}$-morphism. First of all, N-morphisms are defined on elementary net systems, while $\hat{F}$-morphisms are defined on marked graphs. N-morphisms define a relation between the places of the joined systems, such that its inverse is a partial function. Differently, $\hat{F}$-morphisms allow two places to have the same image. Furthermore, for $\hat{F}$-morphisms the mapping between events is surjective, while N-morphisms do not require such constraint. The last main difference is that, if two places s and s' of different elementary net systems are joined by a N-morphism, s belongs to the initial case of the first system if and only if s' is in the initial case of the second one, whereas whith $\hat{F}$-morphism a place of the starting system contain-


N1
1
a
b
d
3
c
4
2
f
e
h
6
g
7
5
i
l
8
N2
13
ac
bd
24
ef
57
hl
gi
68

(a)


N3
a
4
b
e
h
1
3
f
g
c
2
d
N4
ab
eg
13
24
cd
fh

(b)


N5
d
4
a
g
1
e
5
3
6
c
2
b
f
N6
acg
245
136
bdf

(c)

Fig. 2

ing no tokens in the initial marking can be mapped on a place containing tokens.

We now show some interesting features of $\hat{F}$-morphisms.

Theorem 2. *Let $N_i = (S_i, T_i, F_i, M_0^i)$ be marked graphs, for $i = 1, 2$, joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. Let A_1 and A_2 be the incidence matrices of, respectively, N_1 and N_2. Let $s' \in S_2$ be a place of N_2 such that $\sigma^{-1}(s') = \{s_1, s_2, \ldots, s_n\}$. For every transition $t \in T_1$ such that $\tau(t)$ is defined, the following equation holds:*

$$\sum_{i=1}^{n} A_1(s_i, t) = A_2(s', \tau(t)). \tag{1}$$

Proof. In order to prove the theorem, we need to compare the incidence matrices of N_1 and N_2. Let A_i, $i = 1, 2$, be the incidence matrices of, respectively, N_1 and N_2. Because of the structure of a marked graph, it is possible to say that every row of A_i contain one 1 or -1 value or both of them, while the remaining entries of that row contain 0 values. Let us now consider n distinct places $s_1, \ldots, s_n$ of N_1, such that $\sigma(s_i) = s'$, $1 \leq i \leq n$. For each $s_i \in \sigma^{-1}(s')$, if $|{}^\bullet s_i| = 1$ we denote t_{pre} the input transition of $|s_i|$ and, similarly, if $|s_i^\bullet| = 1$, we denote t_{post} the input transition of $|s_i|$. So, if such entries exist, $A_1(s_i, t_{pre}) = 1$ and $A_1(s_i, t_{post}) = -1$. For definition of $\hat{F}$-morphism, $A_2(s', \tau(t_{pre})) = 1$ and $A_2(s', \tau(t_{post})) = -1$. Furthermore, since we consider marked graphs without self-loops and σ defines an injective and surjective partial function between the pre-conditions of transitions joined by τ, for each $s_j \in \sigma^{-1}(s')$, $j \neq i$, we have $A_1(s_j, t_{pre}) = 0$ and $A_1(s_j, t_{post}) = 0$. This proof about one generic s' place of N_2 can be extended to all the places of N_2: so the theorem is proved.

The previous theorem allows us to introduce another interesting feature of $\hat{F}$-morphisms. Intuitively, if two marked graphs N_1 and N_2 are joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$, the pre-images of any element of N_2 contain the same number n of elements.

Theorem 3. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be marked graphs and let $(\sigma, \tau) : N_1 \to N_2$ be a $\hat{F}$-morphism. Every $x \in P_2 \cup T_2$ has pre-image containing the same number n of elements.*

Proof. Let A_i, $i = 1, 2$, be the incidence matrices of, respectively, N_1 and N_2. For every place $s' \in S_2$, if $|\sigma^{-1}(s')| = n$, then it is possible to find n distinct columns $t_1, \ldots, t_n$ of A_1 such that $A_1(s_i, t_i) = 1$ or $A_1(s_i, t_i) = -1$, with $s_i \in \sigma^{-1}(s')$. Let t' be the input or output transition of p'; it is easy to verify that $\tau^{-1}(t') = \{t_1, \ldots, t_n\}$. This means that, if the pre-image of a place of N_2 contains n elements, the pre-images of its input and output transitions also contain n elements. We can extend this proof to every place of N_2, thus proving the theorem.

We call n the *reduction factor* of (σ, τ). The $\hat{F}$-morphism shown in Figure 2b has reduction factor 2, while the one in Figure 2c has reduction factor 3.

3.1 $\hat{F}$-morphisms and behavioral relationships

We now want to show the relationship between the behaviors of two marked graphs joined by a $\hat{F}$-morphism. In this paper we assume that the behavior of a system can be entirely described by means of its *unfolding*, according to the definition given in [9]. For this reason, from now on, we will only consider marked graphs with one technical restriction: in the initial marking there should not be more than one token on each place.

Marked graphs are used to model deterministic systems. The absence of choices in the behavior of deterministic systems can be used to observe that the unfolding of a marked graph does not contain conflicts. In [9] the unfolding of a net N is formally defined as a pair (N', π), where N' is an occurrence net and π is a homomorphism from N' to N. An occurrence net containing no conflicts is called causal net, which is an acyclic marked graph.

Let us now consider N-morphisms defined in [2] for elementary net systems, and compared to $\hat{F}$-morphisms in the previous subsection. Causal nets, used to represent the unfoldings of marked graphs, form a subclass of elementary net systems. This allows us to explicit the relationship between the behaviors of two marked graphs joined by a total $\hat{F}$-morphism.

Theorem 4. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$ and let (N_1', π_1) and (N_2', π_2) be, respectively, the unfoldings of N_1 and N_2. Then, there exists a N-morphism $(\beta, \eta) : N_1' \to N_2'$ which makes the following diagram commute.*

$$\begin{array}{ccc} N_1 & \xrightarrow{\sigma,\tau} & N_2 \\ \uparrow{\scriptstyle \pi_1} & & \uparrow{\scriptstyle \pi_2} \\ N_1' & \xrightarrow{\beta,\eta} & N_2' \end{array}$$

In particular, β^{-1} is an injective partial function and, if (σ, τ) is total, (β, η) is an isomorphism.

The proof of this theorem can be found in [10], together with the necessary theoretical notions. Such proof uses an improved version of McMillan's unfolding algorithm (see [11]) with some modifications.

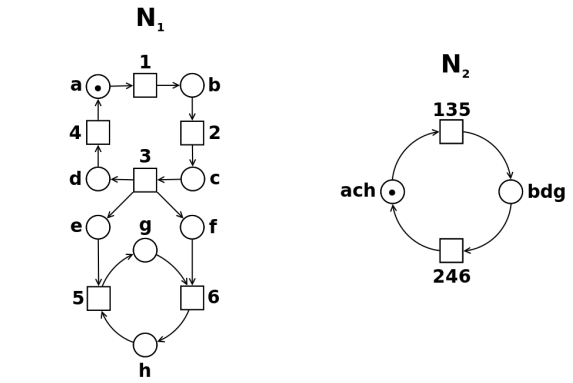
4 $\hat{F}$-morphisms and their properties

In this section we want to analyze some properties about liveness, boundedness, safeness, S and T-invariants of two marked graphs N_1 and N_2, joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. We will first analyze behavioral properties and then structural invariants.

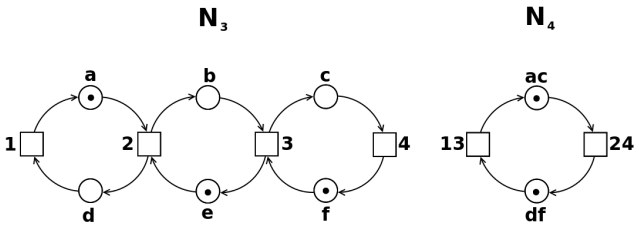
4.1 Analysis of behavioral properties

First of all, it is useful to observe that directed circuits are preserved by $\hat{F}$-morphisms. Intuitively, this means that, given two marked graphs N_1 and N_2 and a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$, if $\gamma = x_1 x_2 \dots x_k x_1$ is a directed circuit of N_1, $x_i \in S_1 \cup T_1$, (σ, τ) maps γ on a directed circuit of N_2.

In [7] marked graphs are defined as Petri nets $N = (S, T, F, M_0)$ in which, for each $s \in S$, it holds $|{}^\bullet s| = |s^\bullet| = 1$. Then, they prove that a marked graph N is live *iff* the initial marking places at least one token on each directed circuit in N. In this paper we consider a more general notion of marked graph: for each place s we have $|{}^\bullet s| \leq 1$ and $|s^\bullet| \leq 1$. It is well known (for example, see [7]) that, given a marked graph N such that $|{}^\bullet s| = 1$ for each place s, N is live if and only if the initial marking places at least one token on each directed circuit in N.



(a)



(b)

Fig. 3

The previous remarks allow to prove that $\hat{F}$-morphisms preserve liveness.

Theorem 5. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. If N_1 is live, then N_2 is also live.*

Generally, liveness is not reflected by $\hat{F}$-morphisms. In Figure 3a an example of $\hat{F}$-morphism from N_1 to N_2 is shown. N_2 is a live net, while N_1 is not live: transitions labelled with `5` and `6` are never enabled.

Since we proved that there is a N-morphism between the unfoldings of two marked graphs joined by a $\hat{F}$-morphism, it is easy to observe that $\hat{F}$-morphisms also preserve occurrence sequences.

Theorem 6. *Let $N_i = (S_i, T_i, F_i, M_0^i)$, $i = 1, 2$, be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. Let $\omega = t_1 \dots t_k$, be an occurrence sequence of N_1 enabled at the initial marking M_0^1. Therefore $\omega' = \tau(t_1) \dots \tau(t_2)$ is an occurrence sequence of N_2 enabled at M_0^2.*

From the definition of $\hat{F}$-morphism, it follows immediately that, if two marked graphs N_1 and N_2 are joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$, for each place s of N_2, the sum of the number of tokens placed by the initial marking of N_1 in the elements of the pre-image of s is equal to the number of tokens placed by the initial marking of N_2 in s. It is possible to extend this condition to every reachable marking of the two systems.

Theorem 7. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. Let $\omega = t_1 \dots t_k$ be an occurrence sequence of N_1 enabled at M_0^1 such that $M_0^1 \xrightarrow{\omega} M$. Then, $\omega' = \tau(t_1) \dots \tau(t_k)$ is an occurrence sequence of N_2 enabled at M_0^2 such that $M_0^2 \xrightarrow{\omega'} M'$ and, for each $s' \in S_2$, the following equation holds*

$$M'(s') = \sum_{s \in \sigma^{-1}(s')} M(s).$$

Using Theorem 7 it is easy to prove that boundedness is preserved by $\hat{F}$-morphisms.

Theorem 8. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. If N_1 is bounded, then N_2 is also bounded.*

So $\hat{F}$-morphisms preserve boundedness but, generally, they do not reflect it. The $\hat{F}$-morphism from N_1 to N_2 represented in Figure 3a does not preserve boundedness: N_2 is a 1-bounded net, while in N_1 the places labelled with `e` and `f` can be filled with an infinite number of tokens.

Note that the reflection of boundedness is obtained if (σ, τ) is total.

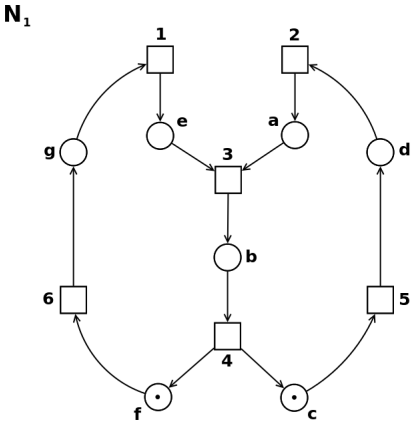
Theorem 9. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$ such that σ is total. If N_2 is bounded, then N_1 is also bounded.*

Proof. Each place of N_1 is mapped on a place of N_2. If N_2 is bounded, by theorem 7 it is easy to see that N_1 is also bounded.

Notice that, in general, safeness (1-boundedness) is not preserved. Let us consider the example shown in Figure 3b: there is a $\hat{F}$-morphism from N_3 to N_4 and, while N_1 is a safe net, N_2 is 2-bounded.

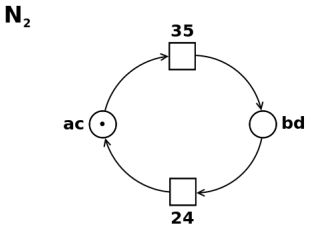
4.2 On structural invariants

We now focus on some properties about S and T-invariants of two marked graphs N_1 and N_2 joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. It is possible to prove that $\hat{F}$-morphisms reflect S-invariants. In order to obtain such result, we need to order the rows of the incidence matrix A_1 of N_1 in the following way. Let A_2 be the incidence matrix of N_2 and let n be the reduction factor of (σ). Given the first row of A_2, representing the place s of N_2, let us consider the n rows of A_1 corresponding to places of N_1 mapped by σ on s. We will put such rows in the first n positions of the matrix. The same procedure can be used to order the remaining rows of A_1. The rows corresponding to places not mapped by σ will occupy the last positions of A_1.



A_1

	1	2	3	4	5	6
a	0	1	-1	0	0	0
c	0	0	0	1	-1	0
b	0	0	1	-1	0	0
d	0	-1	0	0	1	0
e	1	0	-1	0	0	0
f	0	0	0	1	0	-1
g	-1	0	0	0	0	1



A_2

	35	24
ac	-1	1
bd	1	-1

Fig. 4

Theorem 10. *For $i = 1, 2$, let $N_i = (S_i, T_i, F_i, M_0^i)$ be two marked graphs joined by a $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$. Let A_1, A_2 and n be, respectively, the incidence matrices of N_1 and N_2, ordered as seen before, and the reduction factor of (σ, τ). If $\underline{I}_2 = (\alpha_1 \alpha_2 \dots \alpha_P)$, with $\alpha_j \in \mathbb{N}$ and $P = |S_2|$, is a S-invariant for*

N_2, then

$$\underline{I}_1 = (\overbrace{\alpha_1\alpha_1\ldots\alpha_1}^{n\ times}\overbrace{\alpha_2\alpha_2\ldots\alpha_2}^{n\ times}\ \ldots\ \overbrace{\alpha_P\alpha_P\ldots\alpha_P}^{n\ times}0\ldots0)$$

is a S-invariant for N_1.

The previous theorem is proved in [10]. Let us now consider the $\hat{F}$-morphism $(\sigma, \tau) : N_1 \to N_2$ shown in Figure 4, having reduction factor $n = 2$. The incidence matrix of N_1 is ordered as explained. $\underline{I}_2 = (11)$ is a S-invariant for N_2. The corresponding S-invariant for N_1 is built by taking n times each single value of $\underline{I}_2$ as the first components and adding 0s in the remaining positions. Thus, we obtain $\underline{I}_1 = (1111000)$.

$\hat{F}$-morphisms reflect S-invariants but do not preserve them. The S-invariant $\underline{I}_A = (0100111)$ for N_1 in Figure 4 can not be used to build a corresponding S-invariant for N_2. It is impossible to assign to each place of N_2 the weight of the elements of its pre-image. For example, let s be the place of N_2 labelled with `bd`: $\underline{I}_A$ assigns a different weights to the elements of $\sigma^{-1}(s)$. $\underline{I}_B$, built by assigning to each place of N_2 the sum of the weights of the elements of its pre-image, is not a S-invariant of N_2.

Regarding T-invariants, we observe that, in marked graphs, an occurrence sequence leads back to the initial marking if and only if it fires every transition an equal number of times. Then, since $\hat{F}$-morphisms are surjective, by Theorem 7 they preserve T-invariants.

In general, T-invariants are not reflected by $\hat{F}$-morphisms. For instance, let us consider the example in Figure 4. $\underline{J}_2^T = (11)$ is a T-invariant for N_2. For each transition t of N_2, we assign to the elements of its pre-image the weight given by $\underline{J}_2^T$ to t, and we use 0s for the other transitions of N_1. So, we obtain $\underline{J}_1^T = (011110)$, which is not a T-invariant for N_1.

5 Remarks and conclusions

We have introduced F- and $\hat{F}$-morphisms, new kinds of morphisms on marked graphs, a basic class of Petri nets. These morphisms can be used as a formal technique to deal with a kind of abstraction on marked graphs, consisting in the folding of cycles and the identification of chains. We have also proved that the unfoldings of two systems joined by a $\hat{F}$-morphism are joined by a N-morphism (see [2]). We have finally shown that liveness, boundedness and T-invariants are preserved by such morphisms, while S-invariants are reflected.

We now plan to define a new operation for the composition of marked graphs driven by $\hat{F}$-morphisms mapping the components on a net which works as an interface, similarly to what described in [12], [13] for $\hat{N}$-morphisms. We also intend to extend the theory related to F-morphisms to other classes of Petri nets, such as persistent, free choice and Place/Transition Petri nets, thus applying such functions to systems having conflicts. Finally, we want to apply $\hat{F}$-morphisms to models representing real systems having deterministic behavior (such as, for

example, manufacturing systems or cyclic processes) to formally analyze them by using a step-by-step approach based on different levels of refinement of the modelled system.

Acknowledgement.

This work was partially supported by MIUR and by MIUR-PRIN 2010/2011 grant 'Automi e Linguaggi Formali: Aspetti Matematici e Applicativi', code H41J12000190001.

References

1. Desel, J., Merceron, A.: Vicinity respecting homomorphisms for abstracting system requirements. Transactions on Petri Nets and Other Models of Concurrency **4** (2010) 1–20
2. Nielsen, M., Rozenberg, G., Thiagarajan, P.S.: Elementary transition systems. Theor. Comput. Sci. **96**(1) (1992) 3–33
3. Padberg, J., Urbásek, M.: Rule-based refinement of Petri nets: A survey. In Ehrig, H., Reisig, W., Rozenberg, G., Weber, H., eds.: Petri Net Technology for Communication-Based Systems. Volume 2472 of Lecture Notes in Computer Science., Springer (2003) 161–196
4. Winskel, G.: Petri nets, algebras, morphisms, and compositionality. Inf. Comput. **72**(3) (1987) 197–238
5. Bernardinello, L., Mangioni, E., Pomello, L.: Local state refinement and composition of elementary net systems: An approach based on morphisms. T. Petri Nets and Other Models of Concurrency **8** (2013) 48–70
6. Desel, J., Reisig, W.: Place/Transition Petri Nets. In Reisig, W., Rozenberg, G., eds.: Petri Nets. Volume 1491 of Lecture Notes in Computer Science., Springer (1996) 122–173
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (April 1989) 541–580
8. Winskel, G.: Event structures. In Brauer, W., Reisig, W., Rozenberg, G., eds.: Advances in Petri Nets. Volume 255 of Lecture Notes in Computer Science., Springer (1986) 325–392
9. Engelfriet, J.: Branching processes of Petri nets. Acta Inf. **28**(6) (1991) 575–591
10. Bernardinello, L., Pomello, L., Scaccabarozzi, S.: Morphisms on Marked Graphs (Extended Version). `http://www.mc3.disco.unimib.it/pub/bps2014ext.pdf` (2014)
11. Esparza, J., Römer, S., Vogler, W.: An Improvement of McMillan's Unfolding Algorithm. Formal Methods in System Design **20**(3) (2002) 285–310
12. Bernardinello, L., Monticelli, E., Pomello, L.: On preserving structural and behavioural properties by composing net systems on interfaces. Fundam. Inform. **80**(1-3) (2007) 31–47
13. Pomello, L., Bernardinello, L.: Formal tools for modular system development. In Cortadella, J., Reisig, W., eds.: ICATPN. Volume 3099 of Lecture Notes in Computer Science., Springer (2004) 77–96

A Petri Net Approach for Reusing and Adapting Components with Atomic and non-atomic Synchronisation

D. Dahmani[1], M.C. Boukala[1], and H. Montassir[2]

[1] MOVEP, USTHB, Algiers.
dzaouche,mboukala@usthb.dz,
[2] LIFC, Comp. Sci. Dept, Franche-Comté University
hmountassir@lifc.univ-fcomte.fr

Abstract. Composition of heterogeneous software components is required in many domains to build complex systems. However, such compositions raise mismatches between components. Software adaptation aims at generating adaptors to correct mismatches between components to be composed. In this paper, we propose a formal approach based on Petri nets which relies on mapping rules to generate automatically adaptors and check compatibilities of components. Our solution addresses both signature and behaviour level and covers both asynchronous and synchronous communication between components. State space of the Petri model is used to localise mismatches.

Keywords: Interface automata, components reuse, components adaptation Petri nets, synchronous and asynchronous communication.

1 Introduction

Component-based development aims at facilitating the construction of very complex and huge applications by supporting the composition of simple building existing modules, called components. The assembly of components offers a great potential for reducing cost and time to build complex software systems and improving system maintainability and flexibility. The reuse of a component and substitution of an old component by a new one are very promising solution [8, 9].
A component is a software unit characterised by an interface which describes the services offered or required by the component, without showing its implementation. In other terms, only information given by a component interface are visible for the other components. Moreover, interfaces may describe component information at signature level (method names and their types), behaviour or protocol (scheduling of method calls) and method semantics.

A software component is generally developed independently and is subject to assembly with other components, which have been designed separately, to create a system. Normally 'glue code' is written to realise such assembly. Unfortunately, components can be incompatible and cannot work together. Two components are incompatible if some services requested by one component cannot be provided by the other [1, 3]. The pessimistic approach considers two components compatible if they can always work together. Whereas, in the optimistic approach two components are compatible if they can be used together in at least one design [1].

Incompatibilities are identified: (i) at signature level coming from different names of methods, types or parameters, (ii) at behaviour or protocol level as incompatible orderings of messages, and (iii) at semantic aspect concerning senses of operations as the use of synonyms for messages or methods [3].

There exist some works aiming at working out mismatches of components which remain incompatible, even in the optimistic approach. These works generally use adaptors, which are components that can be plugged between the mismatched components to convert the exchanged information causing mismatches. For example, the approach proposed in [11] operates at the implementation level by introducing data conversion services. Similarly, in [2, 7] smart data conversion tools are deployed to resolve data format compatibility issues during workflow composition.
Other works are based on formal methods such as interface automata, logic formula and Petri nets which give formal description to software interface and behaviour [3, 5].
In [4], an algorithm for adaptor construction based on interface automata is proposed. Such adaptors operate at signature level and rely on mapping rules. The adaptors are represented by interface automata which aim at converting data between components according to mapping rules. However, the proposed approach allows not atomic action synchronization, but doesn't cover all possible behaviours. In [3], manual adaptation contracts are used cutting off some incorrect behaviours. They propose two approaches based on interface automata and Petri nets, respectively. However, unlike our approach, these works allow only asynchronous communications. In [6] the behaviour of interacting components is modelled by labelled Petri nets where labels represent requested and provided services. The component models are composed in such a way that incompatibilities are manifested as deadlocks in the composed model. In [13], OR-transition Colored Petri Net is used to formalize and model components where transitions can effectively represent the operations of the software component. Both [6] and [13] focus more on component composition than on adaptation.
In our approach, we propose Petri net construction to check compatibilities of components according to a set of matching rules without any behaviour restriction. Contrary to [3], we deal with both synchronous and asynchronous communications. We use state graph of the Petri net model to localise mismatches.
This paper contains five sections. Section 2 is consecrated to describe interface automata. The concept of mapping rules is given in section 3. In section 4, we

describe our component adaptation approach. Finally, we conclude and present some perspectives.

2 Interface automata

Interface automata are introduced by L.Alfaro and T.Henzinger [1], to model component interfaces. Input actions of an automaton model offered services by the component, that means methods that can be called or reception of messages. Whereas output actions are used to model method calls and message transmissions. Internal actions represent hidden actions of the component. Moreover, interface automata interact through the synchronisation of input and output actions, while internal actions of concurrent automata are interleaved asynchronously.

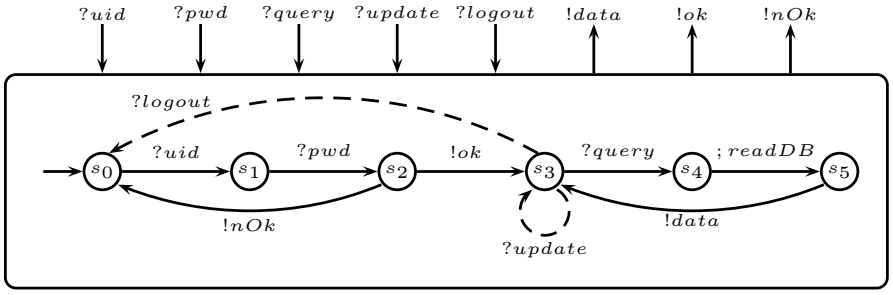
Definition 1 *(Interface automaton)*
An interface automaton $A = \langle S_A, S_A^{init}, \Sigma_A, \tau_A \rangle$ *where :*

- S_A *is a finite set of states,*
- $S_A^{init} \subseteq S_A$ *is a set of initial states. If* $S_A^{init} = \emptyset$ *then* A *is empty,*
- $\Sigma_A = \Sigma_A^O \cup \Sigma_A^I \cup \Sigma_A^H$ *a disjoint union of output, input and internal actions,*
- $\tau_A \subseteq S_A \times \Sigma_A \times S_A$.

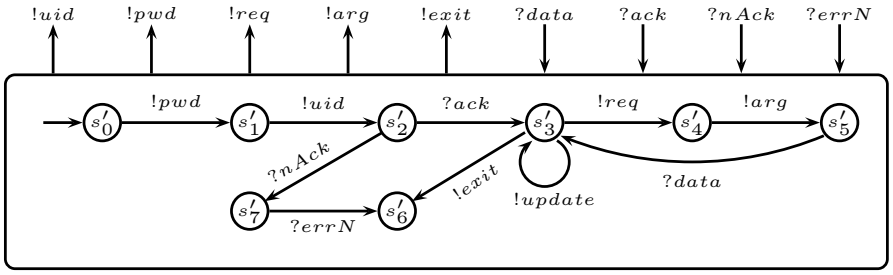
The input or output actions of automaton A are called external actions denoted by $\Sigma_A^{ext} = \Sigma_A^O \cup \Sigma_A^I$. A is closed if it has only internal actions, that is $\Sigma_A^{ext} = \emptyset$; otherwise we say that A is open. Input, output and internal actions are respectively labelled by the symbols "?", "!" and ";". An action $a \in \Sigma_A$ is enabled at a state $s \in S_A$ if there is a step $(s, a, s') \in \tau_A$ for some $s' \in S_A$.

Example 1 *Fig. 1 depicts a model of remote accesses to a data base. This example will be used throughout this paper. The system contains two components Client and Server which have been designed separately. On the one hand, Client issues an authentication message structured into a password* ($!pwd$) *and a username* ($!uid$). *If Client is not authenticated by Server* ($!nAck$ *and* $!errN$), *it exits. Otherwise, Client loops on sending read or update requests. A read request* ($!req$) *is followed by its parameters* ($!arg$), *then Client waits the result* ($?data$). *An update request is an atomic action* ($!update$). *At any moment, Client can exit* ($!exit$).
On the other hand, when Server receives a username ($?uid$) *followed by a password* ($?pwd$), *it either accepts the client access request* ($!ok$) *or denies it* ($!nOk$). *Afterwards, Server becomes ready to receive a read or update requests. If it receives a read request* ($?query$), *it performs a local action* ($;readDB$) *and sends the appropriate data* ($!data$). *Server can execute an update request* ($?update$).
Figure 1.a depicts interface automaton Server. It is composed of six states ($s_0, \ldots s_5$), *with state* s_0 *being initial, and nine steps, for instance* $(s_0, ?uid, s_1)$.

Some arcs are dashed, they will be referred in section 4. The sets of input, output and internal actions are given below:
- Σ^O_{Server} =$\{ok, nOk, data\}$,
- Σ^I_{Server} = $\{uid, pwd, logout, query, update\}$,
- Σ^H_{Server}= $\{readDB\}$.



(a) Server



(b) Client

Fig. 1: *Server* and *Client* interface automata

2.1 Composition of interface automata

Let A_1 and A_2 two automata. An input action of one may coincide with a corresponding output action of the other. Such an action is called a shared action. We define the set $shared(A_1, A_2) = (\Sigma^I_{A_1} \cap \Sigma^O_{A_2}) \cup (\Sigma^O_{A_1} \cap \Sigma^I_{A_2})$, e.g. set $Shared(Client, Server) = \{uid, pwd, update, data\}$.
The composition of two interface automata is defined only if their actions are disjoint, except shared input and output ones. The two automata will synchronize on shared actions, and asynchronously interleave all other actions [1].

Definition 2 *(Composable automata)*
Two interface automata A_1 and A_2 are composable iff

$$(\Sigma^H_{A_1} \cap \Sigma_{A_2} = \emptyset) \wedge (\Sigma^H_{A_2} \cap \Sigma_{A_1} = \emptyset) \wedge (\Sigma^I_{A_1} \cap \Sigma^I_{A_2} = \emptyset) \wedge (\Sigma^O_{A_1} \cap \Sigma^O_{A_2} = \emptyset)$$

Definition 3 *(Synchronous product)*
If A_1 and A_2 are composable interface automata, their product $A_1 \otimes A_2$ is the interface automaton defined by:

1. $S_{A_1 \otimes A_2} = S_{A_1} \times S_{A_2}$,
2. $S^{int}_{A_1 \otimes A_2} = S^{int}_{A_1} \times S^{int}_{A_2}$,
3. $\Sigma^H_{A_1 \otimes A_2} = (\Sigma^H_{A_2} \cup \Sigma^H_{A_1}) \cup shared(A_1, A_2)$,
4. $\Sigma^I_{A_1 \otimes A_2} = (\Sigma^I_{A_1} \cup \Sigma^I_{A_2}) \setminus shared(A_1, A_2)$,
5. $\Sigma^O_{A_1 \otimes A_2} = (\Sigma^O_{A_1} \cup \Sigma^O_{A_2}) \setminus shared(A_1, A_2)$,
6. $\tau_{A_1 \otimes A_2} = \{(v,u), a, (v',u) \mid (v,a,v') \in \tau_{A_1} \wedge a \notin shared(A_1,A_2) \wedge u \in S_{A_2}\}$
   $\cup \{(v,u), a, (v,u') \mid (u,a,u') \in \tau_{A_2} \wedge a \notin shared(A_1,A_2) \wedge v \in S_{A_1}\}$
   $\cup \{(v,u), a, (v',u') \mid (v,a,v') \in \tau_{A_1} \wedge (u,a,u') \in \tau_{A_2} \wedge a \in shared(A_1,A_2)\}$.

An action of $Shared(A_1, A_2)$ is internal for $A_1 \otimes A_2$. Moreover, any internal action of A_1 or A_2 is also internal for $A_1 \otimes A_2$ (3). The not shared input (resp. output) actions of A_1 or A_2 are input (resp. output) ones for $A_1 \otimes A_2$ (4, 5). Each state of the product consists of a state of A_1 together with a state of A_2 (1). Each step of the product is either a joint shared action step or a non shared action step in A_1 or A_2 (6).
In the product $A_1 \otimes A_2$, one of the automata may produce an output action that is an input action of the other automaton, but is not accepted. A state of $A_1 \otimes A_2$ where this occurs is called an illegal state of the product. When $A_1 \otimes A_2$ contains illegal states, A_1 and A_2 can't be composed in the pessimistic approach. In the optimistic approach A_1 and A_2 can be composed provided that there is an adequate environment which avoids illegal states [1].
The automata associated with *Client* and *Server* are composable since definition 2 holds. However, their synchronous product is empty, in fact (s_0, s'_0) is an illegal state: *Client* sends password ($!pwd$) while *Server* requires a username ($?uid$), causing a deadlock situation. Thus, *Client* and *Server* are incompatible. As mentioned in the introduction, two incompatible components can be composed provided that there exits an adaptor to convert the exchanged information causing mismatches. In particular, mapping rules are used to adapt exchanged action names between the components. Such rules may be given by designer. For more details, we refer reader to [11].

3 Mapping rules for incompatible components

A mapping rule establishes correspondence between some actions of A_1 and A_2. Each mapping rule of A_1 and A_2 associates an action of A_1 with more actions of A_2 (one-for-more) or vice versa (more-for-one).

Definition 4 *(Mapping rule)*
A mapping rule of two composable interface automata A_1 and A_2 is a couple $(L_1, L_2) \in (2^{\Sigma^{ext}_{A_1}} \times 2^{\Sigma^{ext}_{A_2}})$ such that $(L_1 \cup L_2) \cap shared(A_1, A_2) = \emptyset$ and if $|L_1| > 1$ (resp. $|L_2| > 1$) then $|L_2| = 1$ (resp. $|L_1| = 1$).

A mapping $\Phi(A_1, A_2)$ of two composable interface automata A_1 and A_2 is a set of mapping rules associated with A_1 and A_2.
We denote by $\Sigma_{\Phi(A_1,A_2)}$ the set $\{a \in \Sigma^{ext}_{A_1} \cup \Sigma^{ext}_{A_2} | \exists \alpha \in \Phi(A_1, A_2)$ s.t $a \in \Pi_1(\alpha) \cup \Pi_2(\alpha)\}$, with $\Pi_1(\langle L_1, L_2 \rangle) = L_1$ and $\Pi_2(\langle L_1, L_2 \rangle) = L_2$ are respectively the projection on the first element and the second one of the couple $\langle L_1, L_2 \rangle$. Observe that each action of $\Sigma_{\Phi(A_1,A_2)}$ is a source of mismatch situation between A_1 and A_2.

Example 2 *Consider again the components of example 1. In Client a read request is structured into two parts (!req and !arg), whereas it is viewed as one part (?query) in Server. A mapping rule is necessarily to map $\{!req, !arg\}$ to $\{?query\}$. The sets of mapping rules between Client and Server $\Phi_{(Client,Server)}$ and $\Sigma_{\Phi(Client,Server)}$ are defined as follows:*

- $\Phi_{(Client,Server)} = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$ *with :*
  $\alpha_1 = (\{!req, !arg\}, \{?query\})$,
  $\alpha_2 = (\{?ack\}, \{!ok\})$,
  $\alpha_3 = (\{?nAck, ?errN\}, \{!nOk\})$,
  $\alpha_4 = (\{!exit\}, \{?logout\})\}$
- $\Sigma_{\Phi(Client,Server)} =$
  $\{req, arg, query, ack, ok, nAck, nOk, errN, exit, logout\}$

4 Towards Components Adaptation

In $A_1 \otimes A_2$, the actions of $\Sigma_{\Phi(A_1,A_2)}$ are interleaved asynchronously since they are named differently in A_1 and A_2. In fact, $A_1 \otimes A_2$ doesn't deal with correspondence between actions of $\Sigma_{\Phi(A_1,A_2)}$. Moreover, the product $A_1 \otimes A_2$ doesn't accept shared actions which have incompatible ordering in A_1 and A_2. For instance *Client* sends a password followed by a user name, whereas *Server* accepts the last message and then the former one. It is obvious that $A_1 \otimes A_2$ cannot be used to check the compatibility of A_1 and A_2. In this context, an adaptor component, must be defined. Such an adaptor is mainly based on the set $\Phi(A_1, A_2)$ and is a mediator between A_1 and A_2. It receives the output actions specified in $\Sigma_{\Phi(A_1,A_2)}$ from one automaton and sends the corresponding input actions to the other. In case of incompatible ordering of shared actions, the adaptor works out such situations by receiving, reordering and sending such actions to their destination component.

Definition 5 *(Adaptation of A_1 and A_2)*
The automata A_1 and A_2 are adaptable according to $\Phi(A_1, A_2)$ if (i) A_1 and A_2 are composable, (ii) $\Phi(A_1, A_2)$ is not empty and (iii) there is a non empty automaton adaptor.

4.1 Petri Net Construction for Components Adaptation

Contrary to interface automata formalism, the Petri net model is well suited to validate interactions between components, especially whenever events reordering is required. In fact, Petri nets allow to store resources (e.g. messages) before their use. In this paper, we use a Petri net model to adapt two interface automata according to a set of mapping rules given by the user of the components. The approach we propose consists of building a Petri net which mimics the component interfaces. Furthermore, the Petri net also contains a set of transitions, one per matching rule, which represent the adaptor component. More details will be given below.

First, we give the basic definitions of a Petri net model. For more details, we refer reader to [12, 10].

Definition 6 *(Labeled Petri Net) A Petri net N is a tuple $\langle P, T, W, \lambda \rangle$ where :*

- *P is a set of places,*
- *T is a set of transitions such that $P \cap T = \emptyset$,*
- *W is the arc weight function defined from $P \times T \cup T \times P$ to $\mathbb{N}$.*
- *λ is a label mapping defined from T to an alphabet set $\Sigma \cup \{\epsilon\}$.*

A marking is a function M: $P \rightarrow \mathbb{N}$ where $M(p)$ denotes the number of tokens at place p. The firing of a transition depends on enabling conditions.

Definition 7 *(Enabling) A transition t is enabled in a marking M iff $\forall\ p \in P$, $M(p) \geq W(p, t)$.*

Definition 8 *(Firing rule in a Marking) Let t be a transition enabled in a marking M. Firing t yields a new marking M', $\forall p \in P$, $M'(p) = M(p) - W(p, t) + W(t, p)$.*

Definition 9 *(State Space) A state space, denoted by $\mathbb{S}(N, M_0)$, of a marked labelled Petri net (N, M_0) is an oriented graph of accessible markings starting from M_0. An arc $M \xrightarrow{t} M'$ of $\mathbb{S}(N, M_0)$ means that M' is obtained by firing t from M.*

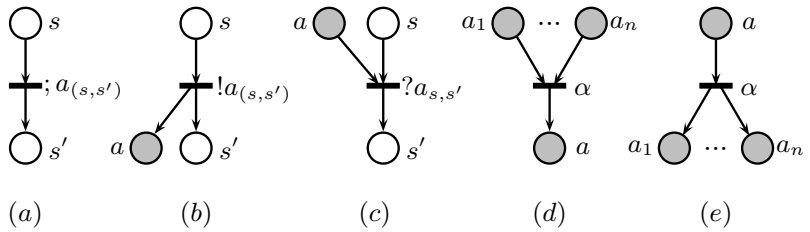


Fig. 2: Translation rules

The algorithm described below returns a marked labelled Petri net (N, M_0) composed of three parts dedicated for A_1, A_2 and a set of matching rules. These parts are glued by mean of places which model communication channels and are associated with external actions of A_1 and A_2, i.e. the actions of sets $Shared(A_1, A_2)$ and $\Sigma_{\Phi(A_1,A_2)}$.

For each state s (resp. external action a) of A_1 and A_2, the algorithm generates a corresponding place s (resp. a) in N. Furthermore, the places corresponding to initial states of interface automata will be initially marked in N.

Fig. 2.a, 2.b and 2.c show how to translate steps of A_1 and A_2. The gray full circles represent communication places. An internal action $s \xrightarrow{;a} s'$ is represented by a transition $;a_{(s,s')}$ which has an input place s and an output place s' (Fig 2.a). An output action $s \xrightarrow{!a} s'$ is represented by a transition $!a_{(s,s')}$ which has an input place s and two output places s' and a. A firing of $!a_{(s,s')}$ produces a token in place a modelling an emission of a message a (see Fig 2.b). Fig 2.c gives the translation of an input action $s \xrightarrow{?a} s'$, here each firing of $?a_{(s,s')}$ models a reception of a message a.

Fig. 2.d and 2.b show how to translate the mismatch rules. For each mismatch rule $\alpha = (\{!a\}, \{?a_1, \ldots, ?a_n\})$ of $\Phi(A_1, A_2)$, a transition α is added. The input places of α are $a_1 \ldots a_n$ and its output place is a. Each firing of α models the receptions of $a_1 \ldots a_n$ and the emission of a (see Fig 2.d). The same pattern is applied for a rule $\alpha = (\{?a_1, \ldots, ?a_n\}, \{!a\})$. Fig 2.e shows the translation of a rule $\alpha = (\{!a_1, \ldots, !a_n\}, \{?a\})$ or $\alpha = (\{?a\}, \{!a_1, \ldots, !a_n\})$. In this case, each firing of α models the reception of a and the emissions of $a_1 \ldots a_n$. These transitions simulate the adaptor.

For analysis requirement (next section), transitions associated with mismatch rules are labelled by the rule names and the others by the corresponding action names.

Algorithm 1 *BuildPetriNet*

`Inputs` $A_1 = \langle S_1, A_1, I_1, T_1 \rangle$`,` $A_2 = \langle S_2, A_2, I_2, T_2 \rangle$ `and` $\Phi(A_1, A_2)$ `a set of rules`
`Output`

A labelled Petri Net $N = \langle P, T, W, \lambda \rangle$ and its initial marking M_0
Initialization $P = \emptyset$, $T = \emptyset$
Begin
// Generation of places corresponding to the states of A_1 *and* A_2
for each state $s \in S_1 \cup S_2$ do
  add a place s to P
  If $s \in S_{A_1}^{init} \cup S_{A_2}^{init}$ then $M_0(s) = 1$ else $M_0(s) = 0$
  Endif
endfor
// Places simulating direct communication between A_1 *and* A_2
for each action $a \in Shared(A_1, A_2)$ do
  add a place a to P
endfor
// Places simulating indirect communication between A_1 *and* A_2
for each action $a \in \Sigma_{\Phi(A_1,A_2)}$ do
  add a place a to P
endfor
// Transitions simulating steps of A_1 *and* A_2
for each transition $s_1 \xrightarrow{\delta a} s_2 \in T_1 \cup T_2$, (with $\delta \in \{!, ?, ;\}$)
  add a transition $\delta a_{s_1,s_2}$ to T
  $\lambda(\delta a_{s_1,s_2}) = \delta a$
  add the arcs $s_1 \rightarrow \delta a_{s_1,s_2}$ and $\delta a_{s_1,s_2} \rightarrow s_2$ to W
  case:
    $\delta = '!'$: add the arc $!a_{s_1,s_2} \rightarrow a$ to W
    $\delta = '?'$: add the arc $a \rightarrow ?a_{s_1,s_2}$ to W
  endcase
endfor
// Transitions simulating adaptor of A_1 *and* A_2
for each $\alpha \in \Phi(A_1, A_2)$ do
  add a transition α to T
  $\lambda(\alpha) = \alpha$
  case:
    $\alpha \in \{(\{!a\}, \{?a_1, \ldots, ?a_n\}), (\{\{?a_1, \ldots, ?a_n\}, \{!a\})\}$:
      add the arcs $a_i \rightarrow \alpha$, $i \in 1 \ldots n$, and $\alpha \rightarrow a$ to W,
    $\alpha \in \{(\{!a_1, \ldots, !a_n\}, \{?a\}), (\{?a\}, \{!a_1, \ldots, !a_n\})\}$:
      add the arcs $a \rightarrow \alpha$ and $\alpha \rightarrow a_i$, $i \in 1 \ldots n$, to W
  endcase
endfor
return (N, M_0)
End

Fig. 3 gives a labelled and marked Petri net N associated with *Client* and *Server* according to the set of rules $\Phi(Client, Server)$ (which are defined in example 2). For sake of clarity, communication places are duplicated and transitions are represented by their labels. Moreover, transitions ?*update*, !*update*

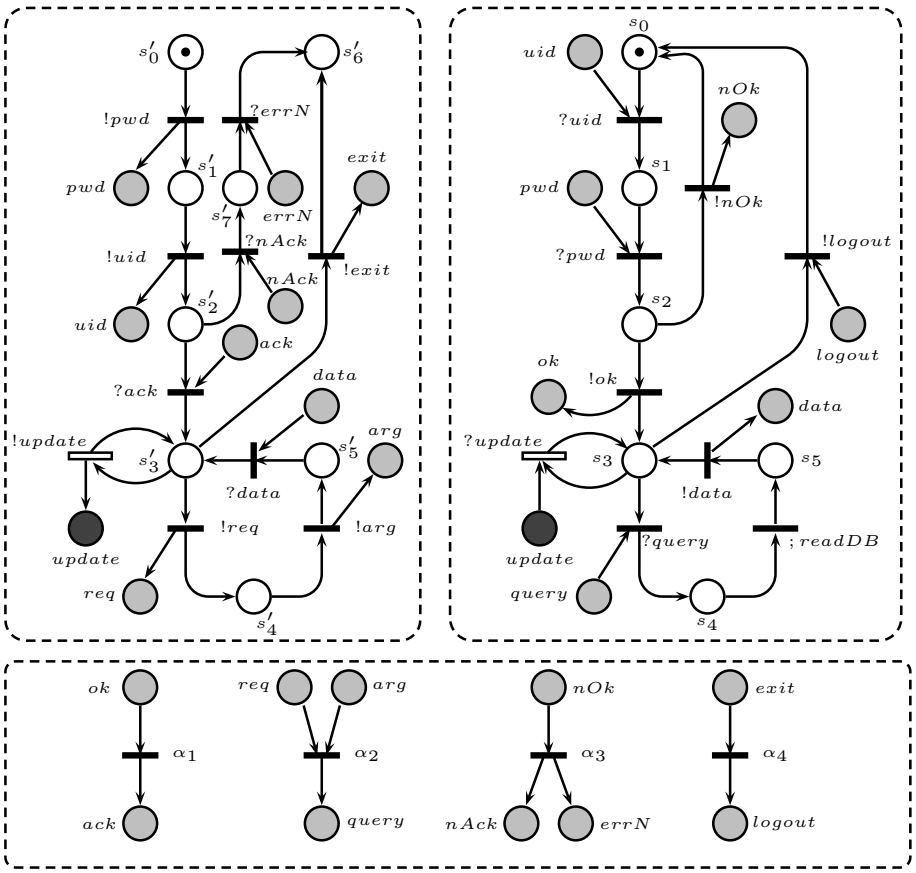


Fig. 3: A Petri net for adaptation of *Server* and *Client*

and place *update* are represented differently, a special attention will be accorded to them in the next section. The left and right parts of the net are respectively dedicated to *Client* and *Server*, they are glued by mean of communication places *uid*, *pwd*, *update* and *data*. These latter correspond to the actions of $Shared(Client, Server)$ and are used to simulate direct communications between *Client* and *Server*.

The lower part of the net represents the *adaptor*, it contains four transitions $\alpha_1, \alpha_2, \alpha_3$ and α_4, each one represents a rule of $\Phi(Client, Server)$. The communication places *req*, *arg*, *query*, *ack*, *ok*, *nAck*, *nOk*, *errN*, *exit* and *logout* are used to link the three parts and correspond to the actions of $\Sigma_{\Phi(Client,Server)}$.

Places s_0 and s'_0 are initially marked in N, they translate the initial places of *Client* and *Server* automata.

4.2 Synchronisation Semantics between Components

At this level, the Petri net construction models only asynchronous communication between two components. Such kind of communication may be source of incoherence as illustrated by the following scenario:

- *Client*: Authentication,
- *Server*: Okay message,
- *Client*: update request,
- *Client*: read request,
- *Server*: response for the read request,
- *Server*: data base update.

It is worth noting that the result of the read request may be incorrect. This occurs whenever the required information is concerned by the *update* operation. To work out this problem, *Client* and *Server* must synchronise on *update* action. Therefore, we propose to enrich the Petri net construction to strengthen synchronisation between transitions which are related to critical shared actions (e.g. *update* action): (1) such transitions *must be fired by pair* (w.r.t some critical action, one for an output step and the other for an input step). (2) The communication places of critical external actions are not useful since here messages are not stored. (3) The set of critical actions, denoted by *Synch*, is an input of the algorithm. The set of transitions related to *Synch* is denoted by T_{Synch}. For instance, to avoid the previous scenario, action *update* is considered as critical, so transitions !*update* and ?*update* must be fired simultaneously. Place *update* is omitted, $Sync = \{update\}$ and $T_{Sync} = \{?update, !update\}$.

4.3 Building and analysing state space

In order to model synchronous communication between components, transitions of T_{Synch} are fired by pair. Further conditions are necessary to fire simultaneously a pair of transitions t and t' belonging to T_{Synch} from a state s :

- $\lambda(t) = \delta a$ and $\lambda(t') = \overline{\delta} a$.
- Both t and t' are enabled in s.

As mentioned in the introduction, the compatibility control of components is made by using the state graph. In order to do this, we adapt the notion of illegal state for our approach. We use the classical definition [1] where an illegal state indicates that some service is required by one component but cannot be offered by the other one.

Definition 10 *(Illegal state)*
Let s be a state of $\mathbb{S}(N, M_0)$, s is an illegal state if:
- s has no successor and contains at least a marked communication place,
- or there is an enabled transition t of T_{synch}, with $\lambda(t) = !a$ but no enabled transition t' with $\lambda(t') = ?a$ in s.

The state graph of the marked Petri net shown in Fig. 3 contains no illegal state, therefore *Client* and *Server* can be composed according to the set of rules $\Phi(Client, Server)$.

Example 3

Consider again the example of Fig. 2 and let us omit the dashed arcs. The corresponding state graph contains two illegal states. Fig. 4 exhibits a particular sequence of the state graph, containing the two illegal states (gray states):

1. In $(s_3 s'_3)$, transition !*update* is enabled but cannot be fired since transition ?*update* is not enabled within the state. This means that *Client* issues an update request which is not assumed by *Server* at this state.
2. State $(s_3 s'_6, exit)$ has no successor in the state graph and a marked communication place (*exit*). Such a mark means that *Client* has sent an *exit* request which will not be covered by *Server*.

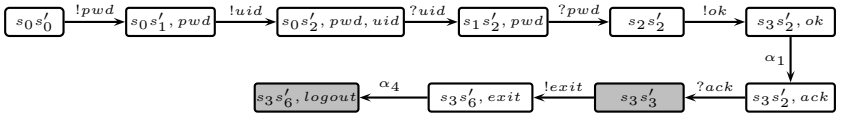


Fig. 4: A firing sequence

5 Conclusion

Software Adaptation is widely used for adapting incompatible components, viewed as black boxes. In this paper, we have presented a Petri net construction for software adaptation at signature and behavioural levels based on mapping rules. These latter are used to express correspondence between actions of components. The Petri net construction reflects the structure of component interface automata to assemble and their corresponding mapping rules. The proposed construction is incremental, e.g. rules can be easily added or replaced. Our approach allows both synchronous and asynchronous communications, unlike the other approaches referred in this paper. In our future work, we plane to extend our Petri net construction to take into account adaptation of components with temporal constraints.

References

1. L. Alfaro and T.A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM*, pages 109–120. Press, 2001.
2. S. Bowers and B. Ludascher. An ontology-driven framework for data transformation in scientific workflows. *DATA INTEGRATION IN THE LIFE SCIENCES, PROCEEDINGS*, 2994:1–16, 2004.
3. C. Canal, P. Poizat, and G. Salaun. Model-based adaptation of behavioral mismatching components. *IEEE Transactions on Software Engineering*, 34(4):546–563, 2008.
4. S. Chouali, S. Mouelhi, and H. Mountassir. Adapting components behaviours using interface automata. In *SEAA'10, 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 119–122, Lille, France, September 2010. IEEE Computer Society Press.
5. S. Chouali, S. Mouelhi, and H. Mountassir. Adapting components using interface automata strengthened by action semantics. In *FoVeoos 2010, int. conf. on Formal Verification of Object-oriented software*, pages 7–21, Paris, France, June 2010.
6. D. C. Craig and W. M. Zuberek. Petri nets in modeling component behavior and verifying component compatibility. In *Int. Workshop on Petri Nets and Software Engineering, in conjunction with the 28-th Int. Conf. on Applications and Theory of Petri Nets and Other Models of Concurrency*, 2007.
7. W. Kongdenfha, H.R. Motahari Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, 2009.
8. C.W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.
9. L. Kung-Kiu and W. Zheng. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.
10. T. Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
11. H.R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati. Semi-automated adaptation of service interactions. In *WWW*, pages 993–1002, 2007.
12. W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 31 July 2013. 230 pages; ISBN 978-3-642-33277-7.
13. Yong Yu, Tong Li, Qing Liu, and Fei Dai. Modeling software component based on extended colored petri net. In Ran Chen, editor, *Intelligent Computing and Information Science*, volume 135 of *Communications in Computer and Information Science*, pages 429–434. Springer Berlin Heidelberg, 2011.

Observable Liveness

Jörg Desel[1] and Görkem Kılınç[1,2]

[1] Fakultät für Mathematik und Informatik, FernUniversität in Hagen, Germany
[2] Universitá degli Studi di Milano-Bicocca, Italy

Abstract. Whereas the traditional liveness property for Petri nets guarantees that each transition can always occur again, observable liveness requires that, from any reachable marking, each observable transition can be forced to fire by choosing appropriate controllable transitions; hence it is defined for Petri nets with distinguished observable and controllable transitions. We introduce observable liveness and show this new notion generalizes liveness in the following sense: liveness of a net implies observable liveness, provided the only conflicts that can appear are between controllable transitions. This assumption refers to applications where the uncontrollable part models a deterministic machine (or several deterministic machines), whereas the user of the machine is modeled by the controllable part and can behave arbitrarily.

1 Introduction

Liveness and boundedness have turned out to be the most prominent behavioral properties of Petri nets – a Petri net is considered to behave well if it is live and bounded. This claim is supported by many publications since decades, and in particular by the nice correspondences between live and bounded behavior of a Petri net and its structure, see e.g. [4, 11]. Nowadays workflow Petri nets receive a particular interest, and with them the behavioral soundness property. However, as shown in [16], soundness of workflow nets is identical to the combination of liveness and boundedness of the net obtained by addition of a feedback place (between the final and the initial transition) to a workflow net. This way, these behavioral properties are also applied to models of processes, that have a start and an end action.

This paper concentrates on liveness, but looks at yet another scenario: Petri nets with transitions that can be observable or unobservable (silent transitions), and can be controllable or not. These nets are inspired by Petri net applications in control theory [8, 2], but can also be seen as a generalization of Petri nets with silent transitions. We provide a notion of liveness which is tailored for Petri nets with observable and controllable transitions, or for the systems modeled by these nets. Observable liveness of a model of a software system (embedded or not) with a user interface roughly means liveness from the user's perspective.

The standard definition of liveness for traditional Petri nets reads as follows:

> A transition t is live if, for each reachable marking m, there is a marking m' reachable from m that enables t. A net is live if all its transitions are live.

We consider Petri net models of software systems where only some activities are observable, and only a subset of these can be controlled by a user (like a vending machine, which has a user interface and an internal behavior). Our liveness notion applies to such nets, which also have observable transitions and, among them, controllable ones. This liveness notion still follows the idea that, no matter which marking m was reached, an occurrence sequence can be constructed which includes a given transition t. However, in contrast to the traditional definition,

- we only consider observable transitions t (i.e., if a transition cannot be observed then we do not care about it),
- we assume that instead of constructing the entire sequence, we (i.e., the user) can only control the net by choosing controllable transitions whenever they are enabled, whereas the net is always free to fire uncontrollable transitions arbitrarily. In particular, if a controllable transition is in conflict with an uncontrollable one, the controllable one might fire but cannot be enforced by the user.

This paper consists of two main parts with two different aims: In the first part of the paper we motivate observable liveness notion for observable software system models. The second part concentrates on the special case where the uncontrollable part of the considered software system behaves deterministically, that means conflict situation can only occur between two controllable transitions. We show that liveness implies observable liveness if no uncontrollable part ever is in conflict with any other transition. This assumption refers to applications where the uncontrollable part models a deterministic machine, whereas the user of the machine is modeled by the controllable part and can behave arbitrarily.

The paper is organized as follows. In Section 2, we introduce our setting and illustrate a simple example. Section 3 is devoted to basic definitions. In Section 4 , we introduce the notion of observable liveness. Section 5 discusses some properties of the new notion and relate it with the traditional liveness. Section 6 is devoted to the case of deterministic uncontrollable behavior. We finish the paper with conclusions, related work and further ideas.

2 The Setting

When defining observable liveness, several design decisions had to be made. We had a particular setting of a modeled system in mind, that motivated our choices. This section aims at explicating this setting and motivating our design decisions.

The generic software system to be modeled consists of a machine (or several machines), a user interface to this machine, and perhaps of activities and conditions which do not belong to the machine. The user can observe and control all activities outside the machine, he can neither control nor observe any activities inside the machine. Concerning the user interface, there are activities that the user can only observe but not control, whereas other interface activities might be both observable and controllable.

One might argue that instead of activities, only local states of machines are observable, for example a light which can be on or off. Then, instead of observing this state, in our setting we observe the activities that cause the changes of the state. In terms of nets, instead of observing a place, we observe the (occurrences of) transitions in the pre- or post-set of the place.

Controllable activities can be those not connected to the machine or can be activities of the interface. Whereas a controllable activity outside the machine is clearly also observable, one might argue that this is not obvious for controllable interface activities. In fact, if the activity can be caused by pressing a button, the user cannot be sure that with every use of this button the activity takes place. An additional prerequisite is that the activity is enabled by the machine, whereas buttons can always be pressed. So we implicitly assume that the user sees whether a controllable transition is enabled or not and can thus distinguish activities from non-activities caused by buttons.

Assume that a user wants to enforce an observable activity a after some previous run of the system. Then, depending on what he has observed so far, he should have a strategy to control activities in such a way that eventually he can observe a. By translating activities to transitions, the same holds for the Petri net model. The strategy is formalized by a function that maps an arbitrary sequence of observable transitions to a set of controllable transitions: if a sequence was observed, then one of these controllable transitions can be fired. Since the domain of this function is infinite in general, and its co-domain finite (theoretically exponential in the number of controllable transitions, but usually linear), different sequences are mapped to the same set. We assume that the user can effectively compute this function by using, e.g., only a finite history or an automata based approach. For generality of our approach, we nevertheless consider a strategy an arbitrary function as above.

There might be states in which controllable activities and uncontrollable ones are enabled, i.e., both the machinery and the user can do something. In such a state, we cannot expect that the user is able to do his controllable activity first. This means that, in case of competition between activities, the user does not have control if not only controllable activities are involved.

For an observably live activity, we want that the user can enforce the occurrence of this activity. Therefore, we provide an appropriate behavioral model of the net. Clearly, the user can only enforce any reaction from the machine if the machine obeys some progress assumption: we do not consider runs in which an uncontrollable transition is enabled, does not occur, and is not in conflict with any other occurring transition. Progress is only assumed for controllable transitions if they are persistently chosen by the response function and moreover concurrent to uncontrollable ones.

Throughout the paper, a controllable transition is illustrated via a black filled rectangle, an observable transition is illustrated by a bold rectangle, while unobservable ones are drawn by not bold rectangles. The incoming and outgoing arcs which are not connected to any place or transition are used when only a part of a net is shown.

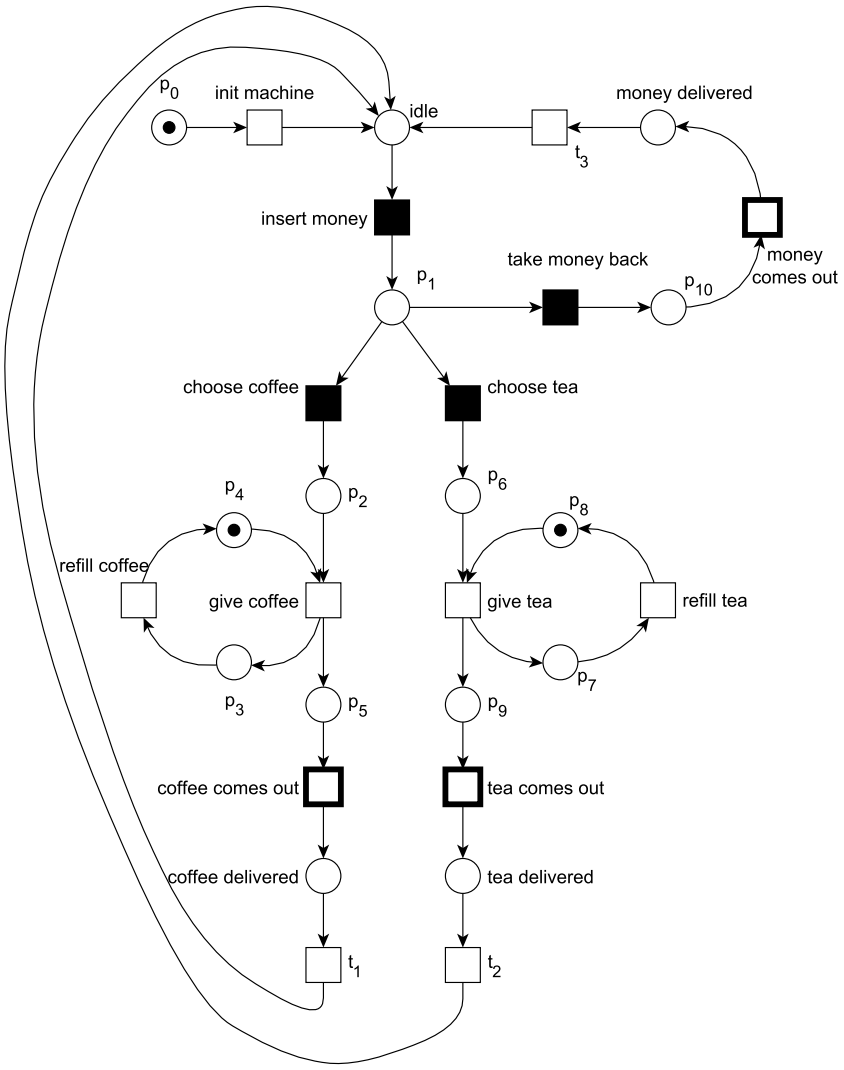


Fig. 1. An observably live net which represents a vending machine.

The example net shown in Fig. 1 models a vending machine with coffee and tea options. The user can operate the machine by inserting a coin and using three buttons (*insert coin, choose coffee, choose tea* and *take money back* are controllable transitions). Using these controllers, the user can take coffee, take tea or take his money back. The transitions *coffee comes out*, *tea comes out* and *money comes out* are observable, and the user can always force these transitions to occur by using the controllable ones. In other words, each of the observable transitions in the net is observably live and so the entire net is observably live. In case that there is no more coffee or tea, the machine needs a refill operation. In this case the user has to wait until the refill operation is done. Regarding the progress assumption, the refill operation will be done since *refill coffee* and *refill tea* transitions will fire eventually, and they are not in conflict with any transitions which can disable them. Note that the entire net is not live since the unobservable part includes a transition which can only fire once (*init machine*). However, this behavior does not affect our notion of observable liveness since the observable transitions can still be forced to fire. Considering such a machine, observable liveness is a useful notion to express the serviceability of a machine via an interface. We can generalize this for models of all kinds of software systems with a user interface. In this case, observable liveness expresses the liveness of a software system from the user's point of view.

3 Basic Definitions

An (initially marked) place/transition net N consists of a finite and non-empty set of places P, a finite and non-empty set of transitions T with $P \cap T = \emptyset$, a set of arcs $F \subseteq (P \times T) \cup (T \times P)$ and an initial marking $m_0 \colon P \to \mathbb{N}$. For a place or transition x, we denote its pre-set by ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$. Similarly, the post-set of x is denoted by $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$.

A marking m is an arbitrary mapping $m \colon P \to \mathbb{N}$. It enables a transition t if each place $p \in {}^\bullet t$ satisfies $m(p) > 0$. If it enables t then t can fire, which leads to the successor marking m', defined by

$$m'(p) = \begin{cases} m(p) + 1 & \text{if } p \in t^\bullet, p \notin {}^\bullet t \\ m(p) - 1 & \text{if } p \in {}^\bullet t, p \notin t^\bullet \\ m(p) & \text{otherwise} \end{cases}$$

We denote this by $m \xrightarrow{t} m'$.

The set of reachable markings of the net N, $\mathcal{R}(N)$, is the smallest set of markings that contains the initial marking m_0 and satisfies

$$[m \in \mathcal{R}(N) \ \wedge \ m \xrightarrow{t} m'] \implies m' \in \mathcal{R}(N).$$

The place/transition net is called *bounded* if $\mathcal{R}(N)$ is finite. Equivalently, it is bounded if and only if there exists a bound b such that each marking $m \in \mathcal{R}(N)$ satisfies for each place p: $m(p) \leq b$. It is called 1-bounded if this condition holds for $b = 1$.

If $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} m_3 \xrightarrow{t_3} m_4 \cdots$, then $t_1\, t_2\, t_3\, t_4 \ldots$ is called *occurrence sequence* (enabled at marking m_1). If an occurrence sequence σ is finite, i.e. $\sigma = t_1\, t_2 \ldots t_n$, then we write $m_1 \xrightarrow{\sigma} m_{n+1}$.

The place/transition net is *live* if, for each reachable marking m and each transition t, there exists a marking m' reachable from m that enables t. Equivalently, it is live if and only if for each transition t and each finite occurrence sequence σ enabled at m_0 there exists a transition sequence τ such that $\sigma\tau t$ is an occurrence sequence enabled at m_0. Note that in order to append two sequences, the left hand one is supposed to be finite. In turn, when writing $\sigma\tau$ we implicitly express that σ is finite.

Transitions can be observable or non-observable, and they can be controllable or non-controllable. We denote by $O \subseteq T$ the set of observable transitions and by $C \subseteq O$ the set of controllable ones.

A place/transition net with observable and controllable transitions is called *observable place/transition net* $N = (P, T, F, m_0, O, C)$. Given an occurrence sequence σ of the place/transition net, its projection $\overline{\sigma}$ to the observable transitions is called observable occurrence sequence. Conversely, a sequence $t_1\, t_2\, t_3 \ldots$ of observable transitions is an observable occurrence sequence if and only if there are finite sequences $\sigma_0, \sigma_1, \sigma_2, \ldots$ of unobservable transitions such that $\sigma_0\, t_1\, \sigma_1\, t_2\, \sigma_2\, t_3 \ldots$ is an occurrence sequence.

An infinite occurrence sequence $t_1\, t_2\, t_3 \ldots$ enabled at some marking m is called *weakly unfair* w.r.t. some transition t if, for some $k \in \mathbb{N}$, $t_1\, t_2 \ldots t_k\, t$ is enabled at m and, for each $j > k$, we have ${}^\bullet t_j \cap {}^\bullet t = \emptyset$ (after some finite initial phase, t is persistently enabled and not in structural conflict with any occurring transition). Notice that this definition is slightly weaker than the usual definition of weak fairness which only demands that t is persistently enabled. The occurrence sequence is *weakly fair* w.r.t. t if it is not weakly unfair w.r.t. t. By this definition, every finite occurrence sequence is weakly fair w.r.t. to all transitions.

There are many different fairness notions for Petri nets (and previously for other models). Our notion - often also called progress assumption - was first mentioned in [12]. It is particularly obvious for partially ordered behavior notions such as occurrence nets and can now be viewed as a standard notion.

4 Observable Liveness

In order to give the definition of observable liveness, we first stick to observable liveness of a single transition, which apparently has to be observable, and later define observable liveness of observable place/transition nets as observable liveness of all observable transitions.

So consider a single observable transition t which might be moreover controllable or not. If the net reaches from the initial marking m_0 a marking m by the occurrence of an arbitrary occurrence sequence σ_0, an agent wants to enforce transition t by selecting appropriate controllable, enabled transitions. If this is always (for each reachable marking m) possible, then we call t observably live.

From the marking m, the net first proceeds arbitrarily and autonomously, i.e., some occurrence sequence σ_1 without controllable transitions occur. This sequence can be

a) finite and lead to a deadlock,
b) finite and lead to a marking that enables controllable and uncontrollable transitions,
c) finite and lead to a marking that enables only controllable transitions,
d) or infinite.

For the infinite case we demand weakly fair behavior w.r.t. all uncontrollable transitions, i.e. there is progress in all concurrent parts of the net.

For cases b) and c), the agent fires a controllable transition and then proceeds as before with a next autonomous sequence σ_2, and so on. This will lead to either an infinite sequence σ_i, or eventually to case a) or case d).

Our liveness notion should express that – in case of observable liveness – there always is (at least one) controllable transition after any sequence σ_i in case c). To formalize this, (and to avoid an infinite alternation of $\forall$ and $\exists$) we introduce a response function φ, which delivers a set of possible controllable transitions as a response of the agent to the sequence observed so far. Notice that an observed sequence does not determine the reached marking because unobservable transitions might occur, changing the marking but not effecting the observed sequence. In turn, different observed sequences might lead to the same marking.

We call the transition t observably live if, for some such response function, we eventually observe t in the sequence created this way.

More formally, the definition reads as follows:

Definition 1. *Let $\varphi\colon O^* \to 2^C$ be a response function and let $m_0 \xrightarrow{\sigma_0} m$ be an occurrence sequence. We call an occurrence sequence σ, enabled at marking m, φ-maximal if it is either an infinite composition $\sigma = \sigma_1\, t_1\, \sigma_2\, t_2\, \sigma_3\, t_3\, \ldots$ or a finite composition $\sigma = \sigma_1\, t_1\, \sigma_2\, t_2 \ldots \sigma_k\, t_k\, \mu$, where $k \geq 0$, satisfying the following:*

a) All σ_i are finite and can be empty, μ is finite or infinite.
b) For each t_i we have $t_i \in \varphi(\overline{\sigma}_0\, \overline{\sigma}_1\, t_1\, \overline{\sigma}_2\, t_2 \ldots \overline{\sigma}_i)$, i.e., t_i is a response to the sequence observed so far.
c) No σ_i contains a controllable transition ($i \geq 1$), and the same holds for μ.

Only for the second variant:
d) μ is weakly fair w.r.t. all non-controllable transitions. μ is moreover weakly fair w.r.t. all controllable transitions t satisfying $t \notin \varphi(\overline{\sigma}_0 \overline{\sigma'})$ for only finitely many prefixes $\sigma' of \sigma$.
e) If μ is finite then all transitions enabled after σ are controllable and do not belong to $\varphi(\overline{\sigma_0}\, \overline{\sigma})$ (this includes deadlocks).

Lemma 1. *Assume that σ_0 leads from m_0 to a marking m and σ is a φ-maximal occurrence sequence enabled at m. If $\sigma = \sigma_1\, \sigma_2$ and $m \xrightarrow{\sigma_1} m_1$, then σ_2 is a φ-maximal occurrence sequence enabled at m_1.*

Proof. The claim follows immediately from the definition of φ-maximal occurrence sequence. □

Some comments: All σ_i in Definition 1 are finite and succeeded by a controllable transition, chosen by the response function. If we get stuck in a deadlock, this is the case of a finite μ. We do not expect that after some σ_i only controllable transitions are enabled. Therefore, there might be situations where the user can fire a controllable transition but also the net can proceed autonomously. If liveness can only be enforced by passivity of the user in this case, the response function yields the empty set for the observed sequence.

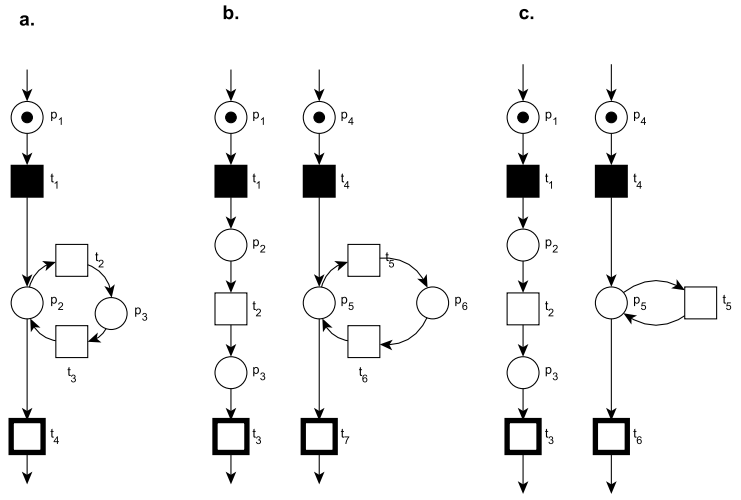


Fig. 2. Some example nets.

Figures 2.a, 2.b, and 2.c illustrate the weak fairness notion employed in our definition of φ-maximal occurrence sequence.

In the net shown in Fig. 2.a., after the controlled occurrence of t_1 the system can choose between t_2 and t_4. It can even always prefer t_2, and t_4 never occurs. Only strong fairness would imply that eventually t_4 can be observed, but our chosen notion of weak fairness does not. So t_4 is not observably live.

In Fig. 2.b., the net of Fig. 2.a. is extended by a concurrent sequence. Our weak fairness assumption implies that the left branch proceeds even if the right stays in an infinite loop. So transition t_3 is observably live.

Figure 2.c. illustrates the difference between our weak fairness and the one usually used in the literature, e.g. [13]. We do not expect that t_6 eventually occurs although it remains enabled at each marking reached after the occurrence of t_4.

However, since t_5 and t_6 share the input place p_5 we do have a conflict here. So again, t_3 is observably live and t_6 is not.

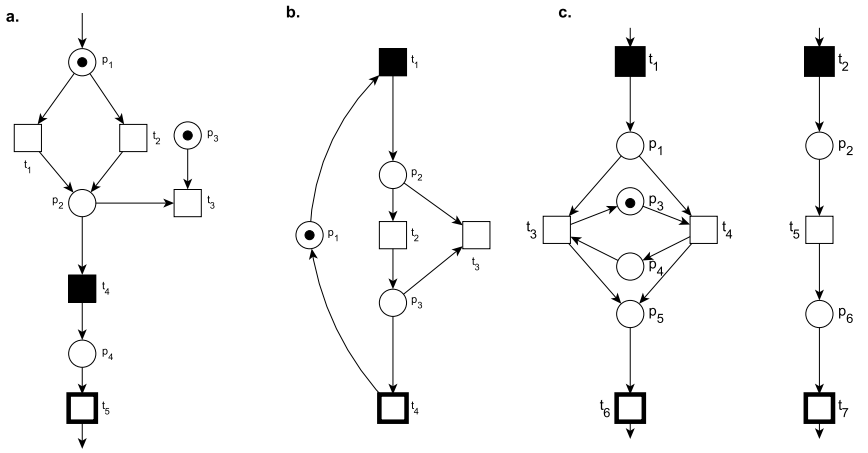


Fig. 3. Example nets.

In the net shown in Fig. 3.a, there is a conflict between t_3 and t_4. In this situation, even if the response function φ tells us to fire t_4 after t_1, we cannot be sure that t_4 will stay enabled since the unobservable transition t_3 might also fire. Since we cannot force t_4 to fire, t_5 is not observably live.

Now we define observable liveness as follows:

Definition 2. *An observable transition t of an observable place/transition net is* observably live *if there is a response function $\varphi_t\colon O^* \to 2^C$ such that, for each $m_0 \xrightarrow{\sigma_0} m$, each φ_t-maximal occurrence sequence enabled at m contains an occurrence of t. An observable place/transition net is observably live if all its observable transitions are observably live.*

In this definition, "an occurrence of t" can be replaced by "infinitely many occurrences of t", as in the definition of traditional liveness.

Theorem 1. *An observable transition t of an observable place/transition net is* observably live *if and only if there is a response function $\varphi_t\colon O^* \to 2^C$ such that, for each $m_0 \xrightarrow{\sigma_0} m$, each φ_t-maximal occurrence sequence enabled at m contains infinitely many occurrences of t.*

Proof. Clearly we only have to prove $\Rightarrow$, because each occurrence sequence with infinitely many occurrences of t has at least one t-occurrence.

So assume observable liveness of t, i.e., a response function $\varphi_t \colon O^* \to 2^C$ such that, for each $m_0 \xrightarrow{\sigma_0'} m'$, each φ_t-maximal occurrence sequence enabled at m' contains an occurrence of t (notice that we replaced σ_0 by σ_0' and m by m').

Let $m_0 \xrightarrow{\sigma_0} m$ and let σ be a φ_t-maximal occurrence sequence enabled at m. We have to show that σ contains infinitely many occurrences of t. By assumption we know that σ contains at least one occurrence of t. Let σ_1 be the prefix of σ that ends after the first occurrence of t and let $\sigma = \sigma_1\,\sigma_2$. Then $m_0 \xrightarrow{\sigma_0\sigma_1} m_1$ for some marking m_1. This marking m_1 enables the φ_t-maximal occurrence sequence σ_2 by Lemma 1. Again using the assumption, σ_2 contains an occurrence of t.

The arbitrary repetition of this argument yields arbitrarily many occurrences of t in σ, whence this sequence must have infinitely many t-occurrences. □

5 Properties and Relations with Traditional Liveness

In this section, we provide some properties of observable liveness and relations to traditional liveness.

Lemma 2. *For each response function φ and each $m_0 \xrightarrow{\sigma_0} m$, there is a φ-maximal occurrence sequence enabled at m.*

Proof. In order to construct a φ-maximal occurrence sequence, we proceed iteratively. Assume that we constructed a finite sequence σ', enabled at m, in accordance with a), b) and c) of Def. 1 and let $m \xrightarrow{\sigma'} m'$. If m' enables an uncontrollable transition t or a controllable one which is in the current response set $\varphi(\overline{\sigma_0}\overline{\sigma'})$, then we append t to σ'. If there is more than one such candidate, we choose the least recently chosen such transition in order to ensure weak fairness.

If this is not possible then all transitions enabled after σ' are controllable and do not belong to $\varphi(\overline{\sigma_0}\overline{\sigma'})$, whence then σ' is a φ-maximal occurrence sequence by e) of Def. 1. □

Proposition 1. *Each observably live transition t is live.*

Proof. Since t is an observably live transition there is a response function φ_t such that for each $m_0 \xrightarrow{\sigma_0} m$, each φ_t-maximal occurrence sequence enabled at m includes t. By Lemma 2 there exists a φ_t-maximal occurrence sequence. This implies that, for each reachable marking m, there exists an occurrence sequence which enables t, and so t is live. □

Corollary 1. *An observably live net is live if all transitions are observable.* □

Notice that Cor. 1 does not hold without the assumption that all transitions are observable. The net shown in Fig. 3.b is not live since t_3 can never occur, but it is observably live.

The converse of Prop. 1 does not hold in general. Figure 2.a, if t_4 is assumed to be connected to t_1, shows a live net which is not observably live. However, if

all transitions are controllable then liveness of t implies its observable liveness, as shown next:

Proposition 2. *If $O = C = T$ then observable liveness of a transition t coincides with its liveness.*

Proof. By Prop. 1, we only have to show the implication $\Leftarrow$.

Assume that t is live. We have to show that there is a response function $\varphi_t \colon O^* \to 2^C$ such that, for each $m_0 \xrightarrow{\sigma_0} m$, each φ_t-maximal occurrence sequence enabled at m contains an occurrence of t. Since t is live, there exists an occurrence sequence σ' enabled at m such that t is enabled after σ'.

Let $\sigma_0\, \sigma'\, t = \overline{\sigma_0 \sigma' t} = t_1 t_2 t_3 \ldots t_k$ and $m_0 \xrightarrow{\sigma_0 \sigma' t}$. We choose any response function with $\varphi_t(t_1 t_2 \ldots t_i) = \{t_{i+1}\}$ for $i = 0, 1, \ldots, k-1$. Since all transitions are controllable, the unique φ_t-maximal occurrence sequence consists of only controllable transitions. The σ_i (for $i = 1, 2, 3, \ldots$) given in Def. 1 are thus empty sequences, and so there is only one φ_t-maximal occurrence sequence for each m. □

Corollary 2. *If $O = C = T$, then observable liveness of a net coincides with liveness of the net.* □

Proposition 3. *Assume that in an observable net there is an infinite and weakly fair occurrence sequence σ without controllable transitions. Then each observable transition which does not appear in σ infinitely often is not observably live.*

Proof. Let $m_0 \xrightarrow{\sigma_0} m$ and assume that t is an observably live transition. There is a response function φ_t such that each φ_t-maximal occurrence sequence enabled at m contains an occurrence of t. So an infinite weakly fair occurrence sequence without controllable transitions σ which is enabled at some marking m' such that $m_0 \xrightarrow{\sigma_0} m \xrightarrow{\sigma'} m' \xrightarrow{\sigma}$ has to include t to be observably live. Since the sequence σ does not include any instance of t, t cannot be observably live. □

Corollary 3. *If an observable net without controllable transitions has an infinite and weakly fair occurrence sequence which does not include all the observable transitions then the net is not observably live.* □

6 Deterministic Uncontrollable Behavior

As seen before, a live net is not necessarily observably live. The main reason is that, for proving liveness, we can always choose an appropriate occurrence sequence enabling some transition t whereas for observable liveness this choice is only possible for controllable transitions (which are not in conflict with unobservable ones) and the net behaves arbitrarily elsewhere.

In this section, we show that the situation is different if the only choices to be made are among controllable transitions. This is not an unrealistic setting; the automated part of a system often behaves deterministically (but still concurrently), whereas the user model might allow for alternatives.

Formally, deterministic behavior is given in terms of the conflict-free property, to be defined next. Intuitively, a transition is conflict-free if it is never in conflict with any other transition; if both are enabled then they are enabled concurrently. Since "never" refers to reachable markings, the definition applies to a net with an initial marking and its state space and not only to its structure. However, each two transitions that are ever in conflict necessarily share an input place which is thus forward branching. With concurrent behavior we mean that two transitions do not compete for tokens. If a place carries more than one token, one could argue that two transitions in its post-set still can occur concurrently (see [17]). We take the stricter view that every two enabled transitions with a common input place (which can carry one or more tokens) are considered in conflict and not concurrent.

Definition 3. *A Petri net is conflict-free w.r.t. a transition u if, for each reachable marking m enabling u, every other transition v enabled at m is concurrent to u, i.e., ${}^\bullet u \cap {}^\bullet v = \emptyset$.*

Figure 3.c shows a net fragment which is conflict-free w.r.t. all its unobservable transitions. Notice that there is concurrency between these transitions. Notice also that forward branching places are possible, provided every reachable marking enables at most one output transition of a branching place. The following lemma will be used frequently in the sequel. It follows immediately from the occurrence rule.

Lemma 3. *Assume two transitions u and v of a net, both enabled at some marking m, such that ${}^\bullet u \cap {}^\bullet v = \emptyset$. Then m enables $u\,v$ as well as $v\,u$, and both sequences lead to the same marking.* □

A well-known result for conflict-free nets [10] is given by the following lemma. We provide a proof for the sake of self-containment, and since our lemma refers to a single conflict-free transition only.

Lemma 4. *If a Petri net is conflict-free w.r.t. a transition u, and some reachable marking m enables u as well as a sequence $\sigma\,u$ where u does not appear in σ, then m also enables the sequence $u\,\sigma$, and the occurrences of $\sigma\,u$ and of $u\,\sigma$ lead to the same marking.*

Proof. By induction on the length of σ.

Base: If σ is the empty sequence then nothing has to be shown.

Step: Assume $\sigma = v\,\sigma'$. We have $u \neq v$ because u does not appear in σ. By conflict-freeness w.r.t. u and since m enables both u and v, these transitions are concurrent. Therefore, and by Lemma 3, m also enables the sequences $v\,u$ and $v\,\sigma'\,u$. Let $m \xrightarrow{v} m'$.

The induction hypothesis can be applied to the marking m', enabling u and $\sigma'\,u$, yielding the sequence $u\,\sigma'$ enabled at m'. So $v\,u\,\sigma'$ is enabled at m. Again since u and v are concurrent and by Lemma 3, m also enables $u\,v\,\sigma'$, which is identical with $u\,\sigma$.

Since each transition occurs in $\sigma\, u$ and in $u\, \sigma$ the same number of times, and by the occurrence rule, the occurrences of these sequences lead to the same marking. □

Lemma 5. *If a Petri net is conflict-free w.r.t. a transition u, and some reachable marking m enables u as well as a sequence σ where u does not appear in σ, then m also enables the sequence $\sigma\, u$.*

Proof. By induction on the length of σ.

Base: If σ is the empty sequence then nothing has to be shown.

Step: Assume $\sigma = v\, \sigma'$. We have $u \neq v$ because u does not appear in σ. By conflict-freeness w.r.t. u and since m enables both u and v, these transitions are concurrent. Therefore, and by Lemma 3, m also enables the sequence $v\, u$. Let $m \xrightarrow{v} m'$.

The induction hypothesis can be applied to the marking m', enabling u and σ', yielding the sequence $\sigma'\, u$ enabled at m'. So $v\, \sigma'\, u$ is enabled at m. We have $v\, \sigma' = \sigma$, which finishes the proof. □

The following theorem constitutes the main result of this paper. It applies only to nets where the only possible conflicts occur between controllable transitions, i.e., to nets which are conflict-free w.r.t. all uncontrollable transitions. This rules out conflicts between two uncontrollable transitions as well as conflicts between controllable and uncontrollable transitions.

As a preparation, we need a couple of definitions and lemmas.

Definition 4. *An occurrence sequence σ enabled at a marking m is called* minimal towards t, *where t is a transition, if σ ends with t, contains no other occurrence of t, and no transition in σ can be postponed, i.e., $\sigma = \sigma'\, t$, t does not occur in σ', and σ cannot be divided as $\sigma = \mu'\, u\, \mu''$ for some transition u, $u \neq t$, such that $\mu'\, \mu''$ is enabled at m, too.*

A transition u can only occur if its input places carry tokens, and another transition v might have to occur before because it produces the token consumed by u. We then call the occurrence of v a causal predecessor of the occurrence of u. A minimal occurrence sequence towards a transition t contains one occurrence of t, its causal predecessors, the predecessors of these predecessors etc., and nothing else. In partially ordered runs, where causal dependence between transition occurrences is explicitly modeled by means of a partial order, this corresponds to a run containing the occurrence of t and all transition occurrences that precede t.

Definition 5. *Given a sequence σ, any deletion (i.e, replacement by the empty sequence) of elements in σ yields a* subsequence *of σ. Its* complementary sequence *is the sequence obtained from σ by deleting all elements that appear in the subsequence.*

This definition captures the case $\sigma = \sigma'\, \sigma''$ where σ' is a subsequence and σ'' is its complementary sequence (and vice versa), but is more general. For example, if $\sigma = t_1, t_2, \ldots, t_{2n}$, the sequence $t_1, t_3, \ldots, t_{2n-1}$ is a subsequence, and $t_2, t_4, \ldots t_{2n}$ its complementary sequence.

Lemma 6. *Assume a conflict-free net with a reachable marking m, a transition t and an occurrence sequence σ enabled at m that contains an occurrence of t. Then there exists a subsequence σ' of σ, enabled at m, which is minimal towards t. Moreover, if σ'' is the complementary subsequence, m enables $\sigma'\,\sigma''$.*

Proof. Define μ as the prefix of σ which ends with the first occurrence of t, and let $\overline{\mu}$ be the rest of σ. Clearly, μ is finite.

Assume that μ can be divided as $\mu = \mu'\,u\,\mu''$ such that $\mu'\,\mu''$ is enabled at m and u does not occur in μ''. By Lemma 5, we can shift u behind μ'' and thus obtain the sequence $\mu'\,\mu''\,u$. Still t occurs only once, being the last transition in μ''.

If u_1 is the rightmost transition (transition occurrence, respectively) in μ for which such a division is possible, we obtain from $\mu\,\overline{\mu}$ the sequence $\mu'_1\,\mu''_1\,u_1\,\overline{\mu}$. Let $\mu_2 = \mu'_1\,\mu''_1$. Now let u_2 be the rightmost transition with the same property for the sequence μ_2 and let $\mu_2 = \mu'_2\,u_2\,\mu''_2$. The same argument as above yields the sequence $\mu'_2\,\mu''_2\,u_2\,u_1\overline{\mu}$. Exhaustive repetition of this procedure yields smaller and smaller sequences μ_i to be considered and eventually the sequence

$$\mu'_k\,\mu''_k\,u_k\,u_{k-i}\ldots u_1\,\overline{\mu}$$

such that no further transition to be postponed can be found in $\mu'_k\,\mu''_k$. So this sequence is minimal towards t. By construction, it is a subsequence of σ, and $u_k\,u_{k-i}\ldots u_1\,\overline{\mu}$ is the complementary subsequence. □

Starting with the next lemma, we additionally require 1-boundedness, i.e., we assume that no reachable marking assigns more than one token to a place.

Lemma 7. *Consider a 1-bounded and conflict-free Petri net with an arbitrary transition t. All initially enabled occurrence sequences which are minimal towards t lead to the same marking.*

Proof. Consider two occurrence sequences μ_1 and μ_2, both enabled at the initial marking, and both minimal towards t. We proceed by induction on the length of μ_1.

Base: The sequence μ_1 has only one element if and only if $\mu_1 = t$. So then t is initially enabled, and hence $\mu_1 = \mu_2 = t$.

Step: Assume that t is not initially enabled. We claim that there is an initially enabled transition u which appears in μ_1 as well as in μ_2, i.e., $\mu_1 = \mu'_1\,u\,\mu''_1$ and $\mu_2 = \mu'_2\,u\,\mu''_2$. When this claim is proven, we know by conflict-freeness that there are also initially enabled occurrence sequences $u\,\mu'_1\,\mu''_1$ and $u\,\mu'_2\,\mu''_2$. By the induction hypothesis applied to the (new initial) marking obtained by firing u and to the sequences $\mu'_1\,\mu''_1$ and $\mu'_2\,\mu''_2$, both sequences lead to the same marking, and we are finished.

So it remains to prove the claim, that some initially enabled transition occurs in μ_1 and in μ_2. We proceed indirectly and assume the contrary.

We again divide μ_2 as $\mu'_2\,\mu''_2$, now such that no transition of μ'_2 occurs in μ_1 and the first transition in μ''_2, say v, occurs in μ_1. By assumption, v is not

initially enabled. The sequence μ_2'' is not empty because both μ_1 and μ_2 contain t. We divide μ_1 as $\mu_1' \mu_1''$ such that μ_1'' begins with the first occurrence of v in μ_1.

Since v is not enabled initially, some place $s \in {}^\bullet v$ is initially unmarked. Since v is enabled after μ_1' and after μ_2', s carries a token after the occurrence of μ_1' and after the occurrence of μ_2'. By conflict-freeness and since the sets of occurring transitions in μ_1' and μ_2' are disjoint, we can also fire both, i.e. $\mu_1' \mu_2'$, from the initial marking. This yields a marking with two tokens on the place s, contradicting 1-boundedness. □

The proof of the above lemma also shows that all minimal sequences towards t have the same length, whence these sequences are exactly the sequences with minimal length containing an occurrence of t.

Now we are ready for the main result: liveness of a 1-bounded net implies observable liveness, provided the only conflict that can appear are between controllable transitions. Although this result might seem obvious at first sight, its proof is surprisingly involved. The core argument of the proof is that, in a live Petri net, for each transition t, every reachable marking m enables an occurrence sequence σ_m that includes an occurrence of t. If t is observable, then observable liveness requires that we can force t to occur by only providing a suitable response function φ_t which controls the behavior whenever there is a conflict. So an obvious idea is to define φ_t in such a way that always the next transition in σ_m is responded, if this transition is controllable. However, φ_t depends not on markings, but on observed sequences. That means, instead of t the user only knows the sequence of observable transitions of the initially enabled occurrence sequence σ_0 that leads to m. For this observed sequence, there might exist many sequences including unobservable transitions, and hence many different reached markings m, and so also many different occurrence sequences σ_m. Instead of the unknown occurrence sequence σ_0 we consider the set of all occurrence sequences μ_0 satisfying $\overline{\mu_0} = \overline{\sigma_0}$. Among these sequences we concentrate on the minimal ones. We will show that, if the net is 1-bounded, all these minimal occurrence sequences lead to the same marking which we call $m_{\overline{\sigma_0}}$. We will moreover show that m, the marking reached by the occurrence of σ_0 is reachable from $m_{\overline{\sigma_0}}$. However, these results only hold for conflict-free nets, and our considered net is not necessarily conflict-free. Since until now we only consider the behavior given by the observed transitions of σ_0, since all controllable transitions are observable and since conflicts only appear among controllable transitions, we can transform the considered net into a conflict-free one, without spoiling the relevant behavior. By liveness (of the original net), $m_{\overline{\sigma_0}}$ enables an occurrence sequence σ containing t. First, we look at the first observable transition in σ. Since there are no conflicts, every occurrence sequence starting at $m_{\overline{\sigma_0}}$ possessing a weak fairness assumption eventually has to enable u. If u is controllable, it might be in conflict with some other transition. In this case we set $\varphi_t(\overline{\sigma_0} = \{u\})$ so that, if u is controllable or not, also u eventually occurs. Fortunately, the distance between this marking and a marking enabling t is smaller than the distance between m and a marking enabling t, where distance is defined in terms of the number of needed observable transitions to reach one marking from the other. So we can repeat the

above considerations, this way defining φ_t on the fly, until we eventually force t to occur.

Theorem 2. *If a 1-bounded observable Petri net, which is conflict-free w.r.t. all uncontrollable transitions, is live, then it is observably live.*

Proof. Consider a 1-bounded live observable Petri net which is conflict-free w.r.t. all uncontrollable transitions. We have to prove observable liveness, i.e., observable liveness of each observable transition t. So let t be an observable transition. To show observable liveness of t, we have to provide a response function φ_t such that, for each $m_0 \xrightarrow{\sigma_0} m$, each φ_t-maximal occurrence sequence σ enabled at m eventually contains t.

The considered net is only partially conflict-free, because there might be conflicts between controllable transitions. To be able to apply the previous lemmas, we make the net conflict-free for a given initially enabled sequence μ_0:

For each observable transition v we add a fresh place s_v, and an arc from s_v to v. Then v can only occur when s_v is marked. Now consider the sequence $\overline{\mu_0} = v_1 v_2 \dots v_k$. For each transition v_i in this sequence except the last (v_k) we add an arc from v_i to $s_{v_{i+1}}$. The place s_{v_1} gets an initial token, the other new places remain unmarked initially.

By construction, every reachable marking of this extended net marks at most one of the new places. Since each observable transition has such a place in its preset, always at most one observable transition is enabled. Since conflicts are only possible between controllable transitions and since each controllable transition is observable, thus no conflict can appear. Therefore, this extended net is conflict-free. By construction, the new initial marking enables μ_0 in the extended net.

The following claim also refers to an arbitrary initially enabled occurrence sequence μ_0 and to the net extended with the places as mentioned above. It generalizes Lemma 7:

Claim: All minimal occurrence sequences μ enabled at m_0 which satisfy $\overline{\mu} = \overline{\mu_0}$ lead to the same marking.

Proof of Claim: by induction on the length of $\overline{\mu_0}$.

Base: If $\overline{\mu_0}$ is empty then the only minimal sequence μ satisfying $\overline{\mu} = \overline{\mu_0}$ is the empty sequence.

Step: Let μ_1, μ_2 be minimal occurrence sequences enabled at m_0 which satisfy $\overline{\mu_1} = \overline{\mu_2} = \overline{\sigma_0}$.

Let $\mu_1 = u_1\, u_2\, \dots\, u_k$ and let u_i be the first observable transition in μ_1. Similarly, let $\mu_2 = v_1\, v_2\, \dots\, v_l$. Then the first observable transition v_j in μ_2 satisfies $u_i = v_j$.

We apply Lemma 6 to both sequences and thus obtain minimal subsequences towards u_i (v_j, respectively). By Lemma 7, both subsequences lead to the same marking. The induction hypothesis applies to the two complementary sequences. This ends the proof of the claim.

The unique (for a given μ_0) marking reached by a minimal sequence μ satisfying $\overline{\mu} = \overline{\mu_0}$ will be called m_{μ_0} in the sequel. Abusing notation, we call the same marking of the original net also m_{μ_0}, ignoring the additional places.

In the following, it will be useful to assume an arbitrary fixed total order $\prec$ on the set of observable transitions, i.e., if u and v are distinct observable transitions then either $u \prec v$ or $v \prec u$.

By liveness of the original net, for each initially enabled occurrence sequence μ_0 there exists (at least one) occurrence sequence μ_0' ending with t which is enabled by m_{μ_0} (in the original net). We assume that μ_0' has a minimal number of observable transitions among all sequences with the above property, i.e., $\overline{\mu_0'}$ has minimal length. Among these minimal sequences we assume moreover that the first observable transition in μ_0' is minimal w.r.t. $\prec$.

Now we define φ_t as follows: For each initially enabled occurrence sequence μ, we set $\varphi_t(\overline{\mu}) = \{u\}$ if $\overline{\mu'}$ begins with u and u is controllable, and $\varphi_t(\overline{\mu}) = \emptyset$ if $\overline{\mu'}$ begins with u and u is not controllable. Notice that $\overline{\mu'}$ contains t as its last transition and is hence not empty.

We now come back to the core of this proof and consider an arbitrary initially enabled occurrence sequence σ_0 which leads to a marking m. We have to show that each φ_t-maximal occurrence sequence enabled at m eventually contains t.

We consider a conflict-free variant of the net as before, but instead of considering only the sequence σ_0 we add places according to the sequence $\sigma_0\, \varphi_t(\sigma_0)$, i.e., we allow to fire the observable transition $\varphi_t(\sigma_0)$ after σ_0.

We proceed by induction on the number of observable transitions in σ_0' (which is defined above as an occurrence sequence ending with t enabled at m_{σ_0} with a minimal number of observable transitions).

Base: Assume that $\overline{\sigma_0'} = t$. Then there is an occurrence sequence σ_0', enabled at m_{σ_0} which eventually contains t (and no other observable transition). Since m is reachable from m_{σ_0} by Lemma 6, for each φ_t-maximal occurrence sequence enabled at m there is a suitable prefix yielding a φ_t-maximal occurrence sequence from m_{σ_0}. By conflict-freeness of the extended net and by weak fairness, each φ_t-maximal occurrence sequence enabled at m_{σ_0} eventually contains t. Hence this holds in particular for those passing through m.

Step: Assume that $\overline{\sigma_0'} = u_1\, u_2 \ldots u_k\, t$, $k \geq 1$. Arguing as in the Base case, there is an occurrence sequence σ_0', enabled at m_{σ_0} which eventually contains u_1 (and no other observable transition). Since m is reachable from m_{σ_0} by Lemma 6, for each φ_t-maximal occurrence sequence enabled at m there is a suitable prefix yielding a φ_t-maximal occurrence sequence from m_{σ_0}. By conflict-freeness of the extended net and by weak fairness, each φ_t-maximal occurrence sequence enabled at m_{σ_0} eventually contains u_1. Hence this holds in particular for those passing m. So each φ_t-maximal occurrence sequence σ enabled at m can be divided as $\sigma_1 u_1 \sigma_2$ where σ_2 is again φ_t-maximal, and $\overline{\sigma_2}$ is shorter than $\overline{\sigma}$. By the induction hypothesis, σ_2 contains t, and therefore so does σ. □

In Fig. 4, we see one net with a conflict and a conflict-free net. The net shown in Fig. 4.a includes a conflict between a controllable transition and an uncontrollable transition (which is also unobservable). Although the net is live, since we cannot force t_1 to fire, both t_1 and t_3 are not observably live and so the net is not observably live. When the conflict in Fig. 4.a is resolved, we get the net shown in Fig. 4.b which is both live and observably live.

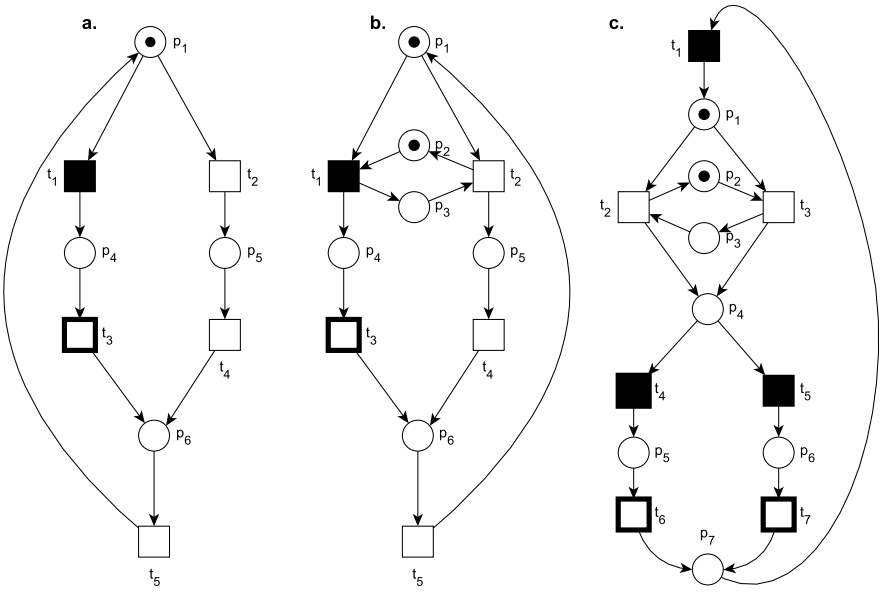


Fig. 4. a: a net with a conflict, **b**: a conflict-free net, **c**: a net which is conflict-free w.r.t. its uncontrollable transitions.

The net shown in Fig. 4.c is conflict-free w.r.t. all its uncontrollable transitions. Notice that there is a conflict between two controllable transitions t_4 and t_5. We can choose the related controllable transition in order to observe the occurrence of any observable transitions. The only choice is ours to make, the uncontrollable part of the machine behaves deterministically. This net is both live and observably live.

7 Conclusion and Related Work

Petri nets are widely used in software engineering for modeling and verifying software systems [3]. In this work, we provide a novel liveness notion which expresses the serviceability of a software system via an interface.

We considered a variant of Petri nets with observable transitions, where an observable transition can also be controllable. For further information about controllability and observability in Petri nets and using Petri nets in control theory, see [2, 15].

In analogy to the usual definition of liveness of a Petri net, we provided a notion for observable liveness, which roughly means that a user can always enforce the occurrence of any observable transition, only by stimulating the net by choosing appropriate enabled controllable transition. Therefore it is necessary to assume that also the uncontrollable part of a net proceeds, i.e., we assume

that the net behaves weakly fair. A similar notion, *T-liveness*, yet for different motivations, is represented in [9]. One of the main differences is that only the fully controllable and observable nets are considered.

In general, liveness does not imply observable liveness and neither the opposite direction holds. This paper proves that for 1-bounded Petri nets with transitions that can be observable or additionally controllable, liveness implies observable liveness, where the latter means that control can force every transition to fire eventually from an arbitrary reachable marking – provided the net model behaves deterministically in its uncontrollable part. This control can only select enabled controllable transitions and is based only on the sequence of transitions observed so far. This way the result generalizes the obvious observation, that in a fully deterministic net a transition is live if and only if it eventually fires.

A future consideration refers to possible generalizations of our result. It clearly still holds when there is some limited nondeterminism in the uncontrolled part. For example, if two alternative uncontrollable transitions cause the same marking transformation, the result is not spoiled. More generally, we aim at defining an equivalence notion on nets, based on the respective observed behavior, which preserves observable liveness. Reduction rules, as defined e.g. in [1], [6] and [4] but also in many other papers, could be applied to the uncontrollable part leading to simpler but equivalent nets. However, there are obvious additional rules. For example, a rule that deletes a dead transition is sound w.r.t. the equivalence because dead uncontrollable transitions do not contribute to the observable liveness or non-liveness of the considered net.

As a future work, we plan to consider an automata approach for the implementation of the response function. The domain of the response function is defined infinite. In order to decide which controllable transitions can be fired next, an arbitrary history of observed transitions has to be considered. Often, a finite amount of the history is enough for this decision. If this is the case, an automata based approach can be used for the realization of the response function: the response then only depends on a state (of finitely many) of this automaton.

Concerning behavior, each run has an alternation between free choices of the machine (where in analysis all possibilities must be considered) and particular choices of the user. Therefore, describing the behavior with AND/OR-trees seems promising, maybe in combination with unfolding approaches. The partial order view would have obvious advantages to capture the progress assumption (that we called weak fairness) in a natural way [5, 14].

A final remark concerns the relation to Temporal Logics. Since liveness and all reachability questions in traditional Petri nets use existential quantification on paths (of the reachability graph), and therefore require Branching Time concepts, our approach explicates *reasons* for desired activities, i.e., transition occurrences. More precisely, as in the discussion of liveness in this paper, we distinguish uncontrollable alternatives and controllable choices, to be able to express that a certain activity (of a user) leads to the eventual occurrence of an event, no matter how the uncontrollable activities behave (but assuming they do not refuse work

at all). This is clearly a Linear Time property. So, very roughly speaking, we translate Branching Time properties to Linear Time properties, and at the same time add details about controllability and observability to the system model. Future work aims at these transformations not only in the context of liveness properties but for arbitrary properties expressed by logical formulae. A related work has been done by Haddad et al. in [7].

Acknowledgements

The authors thank to Lucia Pomello and Luca Bernardinello for their valuable comments. This work was partially supported by MIUR and by MIUR - PRIN 2010/2011 grant 'Automi e Linguaggi Formali: Aspetti Matematici e Applicativi', code H41J12000190001.

References

1. Gérard Berthelot. Transformations and decompositions of nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 254 of *Lecture Notes in Computer Science*, pages 359–376. Springer, 1986.
2. Christos G. Cassandras and Stephane Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
3. Giovanni Denaro and Mauro Pezzè. Petri nets and software engineering. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 439–466. Springer Berlin Heidelberg, 2004.
4. J. Desel and J. Esparza. *Free Choice Petri Nets*. Cambridge tracts in theoretical computer science. Cambridge University Press, 1995.
5. Jörg Desel, Hans-Michael Hanisch, Gabriel Juhás, Robert Lorenz, and Christian Neumair. A guide to modelling and control with modules of signal nets. In Hartmut Ehrig, Werner Damm, Jörg Desel, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 270–300. Springer, 2004.
6. Serge Haddad. A reduction theory for coloured nets. In Grzegorz Rozenberg, editor, *European Workshop on Applications and Theory in Petri Nets*, volume 424 of *Lecture Notes in Computer Science*, pages 209–235. Springer, 1988.
7. Serge Haddad, Rolf Hennicker, and MikaelH. Møller. Specification of asynchronous component systems with modal i/o-petri nets. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, Lecture Notes in Computer Science, pages 219–234. Springer International Publishing, 2014.
8. Lawrence E. Holloway, Bruce H. Krogh, and Alessandro Giua. A survey of petri net methods for controlled discrete event systems. *Discrete Event Dynamic Systems*, 7(2):151–190, 1997.
9. Marian V. Iordache and Panos J. Antsaklis. Design of t-liveness enforcing supervisors in petri nets. *IEEE Trans. Automat. Contr.*, 48(11):1962–1974, 2003.
10. L. H. Landweber and E. L. Robertson. Properties of conflict-free and persistent petri nets. *J. ACM*, 25(3):352–364, July 1978.
11. T. Murata. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, pages 541–580, April 1989.

12. Wolfgang Reisig. Partial order semantics versus interleaving semantics for csp-like languages and its impact on fairness. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 403–413, London, UK, UK, 1984. Springer-Verlag.
13. Wolfgang Reisig. *Elements of distributed algorithms: modeling and analysis with Petri nets*. Springer, 1998.
14. Wolfgang Reisig. *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.
15. Manuel Silva. Half a century after carl adam petri's ph.d. thesis: A perspective on the field. *Annual Reviews in Control*, 37(2):191 – 219, 2013.
16. Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
17. Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke. On causal semantics of petri nets. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2011.

Real-Time Property Specific Reduction for Time Petri Net

Ning Ge

LAAS-CNRS
7 Avenue du Colonel Roche, Toulouse
Ning.Ge@laas.fr

Marc Pantel

University of Toulouse, IRIT-CNRS
2 Rue Charles Camichel, Toulouse
Marc.Pantel@enseeiht.fr

Abstract. This paper presents a real-time property specific reduction approach for Time Petri Net (TPN). It divides TPN models into sub-nets of smaller size, and constructs an abstraction of reducible ones, which exhibits the same property specific behavior, but has less transitions and states. This directly reduces the amount of computation needed to generate the whole state space. This method adapts well to the verification of real-time properties in asynchronous systems. It should be possible to apply similar methods to other families of properties.

Keywords: Real-time property specific reduction, Time Petri net

1 Introduction

The key issue that prevents a wide application of model checking in the industry is the scalability with respect to the size of the target system. A realistic system usually has thousands and even millions of states and transitions. Although a huge part of impossible firing sequences of transitions are eliminated during the building of system's behavior, the interleaving of all others is still a very large number that will easily lead to combinatorial state space explosion. Classic verification methodologies usually encounter scalability issues very quickly along with the growth of system scale, because they follow an implicit purpose: many different kind of properties will be assessed relying on the same state space graph (reachability graph). Indeed, once the reachability graph has been generated, it can be reused to verify different kinds of properties, just by revising the assessed logic formulas. This consideration requires to build the reachability graph preserving precise and sufficient information for the assessment of properties. The existing state space reduction methods, partial order reduction [1, 2], compositional reasoning [3, 4], symmetry [5, 6], abstraction techniques [7], on-the-fly model checking [8, 9], etc., usually follow the same philosophy to produce a complete state space that preserves the mandatory semantics. These generic reduction methods have effectively improved the efficiency of model checking techniques. But their improvement is becoming more and more difficult. We thus might put aside the universality of the semantics expressed in the state space graph, and take into account property specific reduction methods.

This work proposes a real-time property specific state space reduction approach for Time Petri Net (TPN). It divides the TPN model into sub-nets of

smaller size, and constructs an abstraction of reducible sub-nets, which exhibit the same property specific behavior, but has less transitions and states. The real-time property specific behavior (called real-time behavior for short in the following parts) of TPN sub-nets is an abstraction of the whole state-transition traces that only preserves real-time behaviors from the viewpoint of observations. This method adapts well to the verification of real-time properties in asynchronous systems. It could be possible to apply similar methods to other families of properties.

This paper is organized as follows: Section 2 presents some related works; Section 3 introduces real-time properties and Time Petri Net; Section 4 gives an overview of property specific reduction methods; Section 5 defines two real-time behavior regularities for this work; Section 6 details the proposed reduction method; Section 7 provides experimental results; Section 8 discusses the behavior coverage issue; Section 9 gives some concluding remarks.

2 Related Works

Several existing works [10–13] defined reducible sub-net patterns for Petri nets, Time Petri nets or Colored Petri nets, based on the idea of fusing redundant places and transitions. They provide in fact simple behavior equivalent patterns. The state space reduced by these patterns is rather limited.

The idea of our approach is similar to the partial order reduction [14, 2] and the state space abstraction techniques applied in the TINA toolset.

The partial order reduction is usually used in asynchronous concurrent systems, where most of the activities in different processes are performed independently, without a global synchronization. Its main idea is to construct a reduced state class graph by analyzing the dependencies between the transitions and exploiting the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. A set of non-reducible transitions are preserved in the reduced state class graph. The reduced behavior is a subset of the behavior of the full state class graph. Compared to the partial order reduction, the proposed property specific reduction exploits the commutativity of TPN sub-nets, which result in the same property specific behavior.

The TINA toolset provides various state space abstractions for TPN when generating state class graphs, following the techniques proposed in [15, 9]. Depending on the abstraction options, the construction can preserve the traces required by the verification of markings, states, LTL, or ctl* properties. This work relies on the state class graph preserving markings to verify the real-time properties in TPN. Even with this highest abstraction, the state space still rapidly increases along with system scale. Therefore, more abstract state class graphs dedicated to one type of properties (in our case real-time properties) is needed.

3 Preliminaries

3.1 Time Petri Net

Time Petri nets [16] extends Petri Nets with timing constraints on the firing of transitions. Here we use the formal definition of TPN from [17] to explain its syntax and semantics.

Definition 1 (Time Petri Net). *A Time Petri Net (TPN)* $\mathcal{T}$ *is a tuple* $\langle P, T, {}^\bullet(.), (.)^\bullet, M_0, (\alpha, \beta)\rangle$*, where:*

- $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of places;*
- $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of transitions;*
- ${}^\bullet(.) \in (\mathbb{N}^P)^T$ *is the backward incidence mapping;*
- $(.)^\bullet \in (\mathbb{N}^P)^T$ *is the forward incidence mapping;*
- $M_0 \in \mathbb{N}^P$ *is the initial marking;*
- $\alpha \in (\mathbb{Q}_{\geq 0})^T$ *and* $\beta \in (\mathbb{Q}_{\geq 0} \cup \infty)^T$ *are respectively the earliest and latest firing time constraints for transitions.*

Following the definition of enabledness in [18], a transition t_i is enabled in a marking M iff $M \geq {}^\bullet(t_i)$ and $\alpha(t_i) \leq v_i \leq \beta(t_i)$ (v_i is the elapsed time since t_i was last enabled). There exist a global synchronized clock in the whole TPN, and $\alpha(t_i)$ and $\beta(t_i)$ correspond to the local clock of t_i. The local clock of each transition is reset to zero once the transition becomes enabled. The predicate $\uparrow Enabled(t_k, M, t_i)$ in the following equation is satisfied if t_k is enabled by the firing of transition t_i from marking M, and false otherwise.

$$\uparrow Enabled(t_k, M, t_i) = (M - {}^\bullet(t_i) + (t_i)^\bullet \geq {}^\bullet(t_k)) \wedge ((M - {}^\bullet(t_i) < {}^\bullet(t_k)) \vee (t_k = t_i)) \quad (1)$$

Time Petri Net is widely used to formally capture the temporal behavior of concurrent real-time systems due to its easy-to-understand graphical notation and the available analysis tools, such as TINA, INA, Roméo, etc.

3.2 Real-Time Property Verification

The safety and reliability of real-time systems strongly depend on the satisfaction of its real-time requirements, in both qualitative and quantitative aspects.

Dwyer et al. initially proposed qualitative temporal property patterns for finite-state verification in [19]. Konrad created in [20] mappings of quantitative requirements into timed logics MTL, TCTL, and RTGIL, and defined a pattern template to ease the reuse. From the viewpoint of property verification, the real-time requirements expressed by Dwyer's and Konrad's patterns are not atomic. We thus defined a minimal set of atomic patterns, which allows to specify the same time requirements as Dwyer's and Konrad's patterns do, to ease the property verification based on observers. We have defined 12 event-based and 4 state-based observers and verified real-time requirements using the reachability assertions. Some early results about the observer-based verification approach are presented in [21, 22].

4 Approach Overview

Let's first see an example benefiting from property specific reduction method.

Example 1 (Example of Property Specific Reduction). When generating the reachability graph preserving markings for the TPN model in Fig. 1 by TINA, it contains **177 states** and **365 transitions**. This system is identified as two sub-nets A and B: A is the structure in dotted box, and B is the other parts. The transition t_4 is the only portal transition between A and B. From the viewpoint of A through t_4, A does not know the inner structure and inner behavior of B, only two informations are observable: how many times t_4 will be fired and the time range for each firing occurrence of t_4.

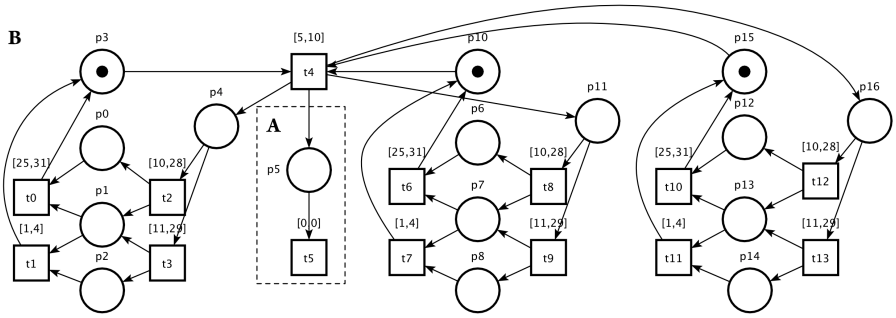


Figure 1. Example of Property Specific Reduction

We provide these informations based on the real-time property verification method presented in the previous section. t_4 is fired infinitely. The time ranges for each firing occurrence are shown in Table 1. For each firing occurrence n ($n \in \mathbb{N}$) of t_4, the time range $[t_n^{min}, t_n^{max}]$ is $[5 + 17(n-1), 10 + 69(n-1)]$. The behavior regularity in this case is that, except the first occurrence, the time difference between current occurrence and the previous one is always in [17, 69].

A sub-net B' conforming to this regular pattern is constructed to replace original sub-net B, as shown in Fig. 2. Sub-net A is kept as before. The reachability graph of the reduced TPN only contains **3 states** and **3 transitions**, but exhibits the same real-time behavior as before from the viewpoint of A.

To summarize the main objective of this work from the above example, we aim to find the regularity of the real-time behavior for the TPN sub-nets from the viewpoint of observations. As we only observe TPN transitions, the real-time behavior from the viewpoint of observed transitions concerns both the firing occurrence times and the time range of each firing occurrence. A reducible sub-net must be independent of its surrounding behavioral context. It means that

Occurrence	Time $[t_i^{min}, t_i^{max}]$	Time Diff $[t_i^{min} - t_{i-1}^{min}, t_i^{max} - t_{i-1}^{max}]$
0	[0, 0]	-
1	[5, 10]	[5, 10]
2	[22, 79]	[17, 69]
3	[39, 148]	[17, 69]
...	...	...
n	[5+17(n-1), 10+ 69(n-1)]	[17, 69]

Table 1. Real-Time Behavior

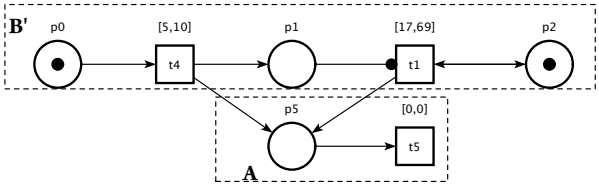


Figure 2. Example Result of Behavioral Equivalence

whether it is "knocked out" from the system or not, it will exhibit exactly the same behavior whenever it is measured, in terms of occurrence times of the portal transition and its time range of each firing occurrence.

An overview of the approach is illustrated in Fig. 3. First, some reducible sub-nets like A, B, and C are identified from the whole TPN model using the *Identification* functions. These sub-nets contain either none incoming transition and one unique outgoing transition such as A, denoted as *one-way-out pattern*; or one incoming and one outgoing transitions such as B and C, denoted as *generic pattern*. The regularity of real-time behaviors for each reducible sub-nets A, B and C are searched using *Reduction* functions relying on observer-based property verification method. If the regularity is founded, reduced sub-nets (A', B', and C') are constructed to replace the original ones after their soundness is assessed by the *Refinement* functions, which also rely on the observer-based property verification method. As the *one-way-out pattern* and the *generic pattern* rely on different identification functions but similar reduction and refinement functions, for the page limit, we only develop our discussion based on the *one-way-out pattern*.

5 Regularity of Real-Time Behavior

The regularity of real-time behavior depends on the characteristics of a system. Fig. 4 illustrates two possible regularities of real-time behavior from the viewpoint of observed transitions. The TPN in Fig.4 (a) has 3 sub-nets: A, B and C. A (resp. B) has a unique portal transition T_A (resp. T_B) to C, and produces tokens via T_A (resp. T_B) periodically or sporadically. From the viewpoint of C,

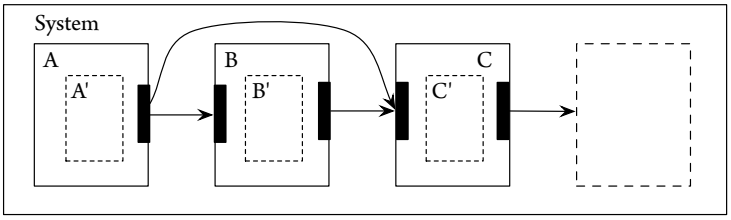


Figure 3. Overview of Behavior Equivalence Approach

regardless the complex inner behaviors of the A and B, they can be seen as single transitions that may fire regularly under a pattern to feed C by tokens. There exists thus an opportunity to abstract and redefine this *regularity* to a reduced `TPN` A' (resp. B') that may contains less states and transitions than the original one.

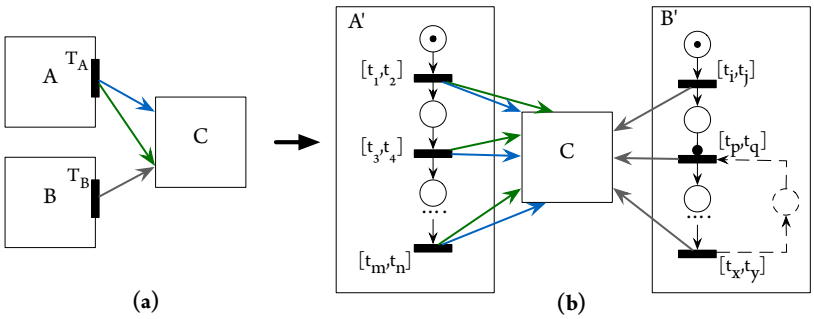


Figure 4. Reduction pattern

When the observation is performed on a `TPN` transition, the regularity of its firing occurrence is either finite or infinite. The time range of each firing occurrence can be measured using observers if the time ranges are bounded.

Fig. 4 (b) shows two kinds of possible regularity. Assume that we observe the firing time of transitions T_A and T_B for each firing occurrence. The occurrence of T_A/T_B can be either finite (A) or infinite (B). An infinity observer can be added on a transition to check its infinity. Each occurrence T_i has a bounded time range $[t_i^{min}, t_i^{max}]$. These ranges are derived by adding `BCET` (Best Case Execution Time) and `WCET` (Worst Case Execution Time) observers on T_A and T_B.

Finite Firing Occurrence If the occurrence is finite, the sub-net A can be represented by a finite sequential section of transitions $\mathbb{T}_{seq} = \{T_i\}$ $(i \in \mathbb{N})$ with adapted time range $[T_i.min, T_i.max]$, where $T_i.min = t_i^{min} - t_{i-1}^{min}$, and

$T_i.max = t_i^{max} - t_{i-1}^{max}$ and $t_0^{min} = t_0^{max} = 0$. It is possible that the regularity of A contains several control modes that lead to several branches with finite sequential transitions.

Infinite Firing Occurrence If the occurrence is infinite, as we focus on finite-state systems, the states in sub-net B must be finite. In other words, there must exist a repeating pattern in B. Depending on system's behavior, there are several possible repeating patterns, such as single loop pattern, nested loop pattern, etc. In this paper, we only discuss one of them: the pattern that is composed of an eventual finite sequential section of transitions $\mathbb{T}_{seq} = \{T_i\}$ $(i \in \mathbb{N})$ and a loop section of transitions $\mathbb{T}_{loop} = \{T_j\}$ $(j \in \mathbb{N})$. The other patterns are under study. Therefore, for now, if the system does not behave the infinite regularity with an eventual sequential section and a loop section, it is considered as non-reducible.

6 Real-Time Property Specific Reduction

The property specific state space reduction method follows three steps (functions): *identification*, *reduction* and *refinement*, which rely on the real-time property specification and observer-based verification approaches presented in [21, 22]. This section details the algorithms for the above functions for the *one-way-out pattern*.

6.1 Identification Function for One-Way-Out Pattern

We first define a symbolic system to ease the discussion:

- t^+ and t^-: for a given transition t, represent respectively the outgoing and incoming arcs of t.
- p^+ and p^-: for a given place p, represent respectively the outgoing and incoming arcs of p.
- $\mathbb{T}^{R(N)}$ and $\mathbb{P}^{R(N)}$: for a given `TPN` N, represent respectively the sets of reducible transitions and places.

We distinguish the reducible and non-reducible `TPN` structure. Non-reducible elements include those structures directly associated with properties, including observer structures, structures directly linked to observers and places/transitions referred to by reachability assertions. The other parts are considered as reducible.

Before performing property specific reduction, some property-irrelevant structures can be directly removed from the reducible net. They are the structures that have causality to the observers. The exact causality can be measured using the reachability graph of the whole system. The paradox exists here: if the whole reachability graph can be generated, we may not need any reduction method. Therefore, to ensure the safety of the removal, we rely on the dependency analysis in `TPN` as a over-approximation. The detailed dependency algorithm is trivial thus will not be presented here. Now assume the set of $\mathbb{T}^{R(N)}$ and $\mathbb{P}^{R(N)}$ are available after the removal.

Identification function $\mathbf{F}(N) = <A, T_{out}>$ identifies, for a given `TPN` N, the enclosed sub-net A that could be possibly reduced (necessary condition), and the unique portal outgoing transition T_{out}:

- A is a connected graph, $A \subset N$, $T_{out} \in A$
- $\forall p \in A, (p \in \mathbb{P}^{R(N)}) \wedge (p^+ \subset A) \wedge (p^- \subset A)$
- $\forall t \in A, (t \in \mathbb{T}^{R(N)}) \wedge (t^- \subset A)$
- $(T_{out} \in A) \wedge (T_{out}^+ \cap \overline{A} \neq \emptyset)$

6.2 Reduction Function

Reduction function $\mathbf{G}(A, t) = <N_S, N_L>$ extracts, for a given sub-net A and the outgoing portal transition t, the behavioral equivalent sequential section N_S for the finite cases, or an eventual sequential section N_S and the loop section N_L for the infinite cases. It first checks the infinity of t in sub-net A using an infinity observer. In both cases, the bounding time range $[t_i^{min}, t_i^{max}]$ is measured using predefined `BCET` and `WCET` observers for the i^{th} firing occurrence of t.

Building Sequential Section In the finite case, there is only a sequential section N_S. The set of sequential transitions $\mathbb{T}_{seq} = \{T_i\}$ $(i \in \mathbb{N})$ in N_S is built using $[t_i^{min}, t_i^{max}]$. Each transition T_i in $\mathbb{T}_{seq}$ is associated with a time range $[T_i.min, T_i.max]$. The algorithm for building N_S from A using the transition t is described in Algo. 1. Initially, t_o^{min} and t_0^{max} are set as 0. N_S starts from an initial place with one token. Whether t^i has occurred is checked using `tHasOcc(i)` function relying on an occurrence observer. For each occurrence (i) of fired t, a pair of `BCET` and `WCET` observers are added to t in the sub-net A to compute the t_i^{min} and t_i^{max}. Then the time range $[T_i.min, T_i.max]$ is associated to the transition T_i. T_i is added in N_S, and an associated new place without token is also added in N_S.

Data: A, t
Result: N_S
$t_o^{min} := 0$, $t_0^{max} := 0$;
N_S`.add(new Place(1))` ;
i := o ;
while *`tHasOcc(i++)`* **do**
  t_i^{min} := `getOccBCET(A,t,i)` ;
  t_i^{max} := `getOccWCET(A,t,i)` ;
  $T_i.min = t_i^{min} - t_{i-1}^{min}$;
  $T_i.max = t_i^{max} - t_{i-1}^{max}$;
  N_S`.add(`T_i`, new Place(0))` ;
end

Algorithm 1: Building Sequential Section

Building Loop Section In the infinite case, the key issue is to identify the firing occurrence of t that divides the sequential section N_S and the loop section N_L. The Algo. 2 is proposed to build the N_S and N_L sections by searching for the loop starting transition ($loopStartIndex$) and the length of loop ($loopLength$).

Data: $A, t, occThreshold, loopThreshold$
Result: N_S, N_L
$t_0^{min} := 0$, $t_0^{max} := 0$;
N_S.add(new Place(1)) ;
occ := 0 ;
while *occ++* $\leq$ *occThreshold* **do**
  t_{occ}^{min} := `getOccBCET(A,t,occ)`; t_{occ}^{max} := `getOccWCET(A,t,occ)` ;
  for *loopStartIndex = 0; loopStartIndex < occ; loopStartIndex ++* **do**
    for *loopLength = 1; loopLength* $\leq$ *occ - loopStartIndex; loopLength ++* **do**
      match : = 0 ;
      for *index = loopStartIndex; index* $\leq$ *occ - loopLength; index++* **do**
        if *isSame(*$<t_{index}^{min}, t_{index}^{max}>$, $<t_{index+loopLength}^{min}, t_{index+looplength}^{max}>$*)* **then**
          match++ ;
        end
        else break;
      end
      if *match* $\geq$ *loopThreshold* **then**
        for *k = 1; k < loopStartIndex; k++* **do**
          $T_k.min = t_k^{min} - t_{k-1}^{min}$; $T_k.max = t_k^{max} - t_{k-1}^{max}$;
          N_S.`add(`T_k`, new Place(0))` ;
        end
        for *k = loopStartIndex; k < loopStartIndex + loopLength; k++* **do**
          $T_k.min = t_k^{min} - t_{k-1}^{min}$; $T_k.max = t_k^{max} - t_{k-1}^{max}$;
          N_L.`add(`T_k`, new Place(0))` ;
          N_L.`connect(lastPlace,` $T_{loopStartIndex}$`)` ;
        end
        return ;
      end
    end
  end
end

Algorithm 2: Building Loop Section

As the firing occurrence of t is infinite, an occurrence bound value is predefined as $occThreshold$ to stop the algorithm. As the Identification function $\mathbf{F}(N)$ uses necessary conditions, the identified sub-net A is considered as non-reducible if the loop section cannot be found using $occThreshold$. Another bound value $loopThreshold$ judges whether the $loopStartIndex$ and the $loopLength$ are

found. If the loop pattern holds for *loopThreshold* times, it is considered that this division of N_S and N_L is statistically correct. It is obvious that no matter how big the *loopThreshold* is, the assurance cannot reach 100%, because the loop execution is infinite. In order to make sure that the reduced net refines exactly the same behavior as before, a pre-check (refinement function) must be performed before accepting the reduced structure.

6.3 Refinement Function

The refinement function verifies the behavioral equivalence between the reduced sub-net and the original one. Fig. 5 shows the principle of this function: comparing the time range of each firing occurrence between the nets B and B'. It is realized by adding time interval observers between the transition T_B in B and the transitions T_i in B'. Although the firing occurrence is infinite, under the repeating pattern, the number of T_i is finite. If the refinement fails, it means the system does not fit the behavior regularity, and thus the reduction method cannot be applied.

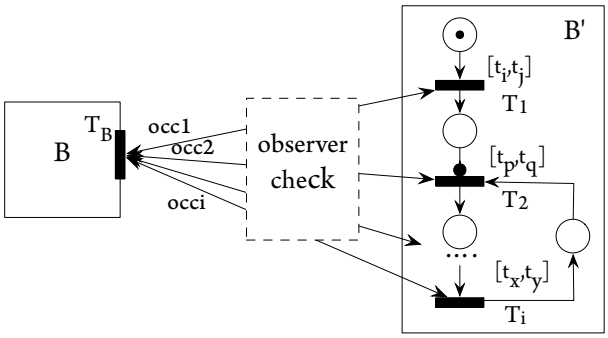


Figure 5. Refinement Function

It is possible that the observed time range do not fully refine the original behavior because of possible "time holes" in this range. For example, a transition can fire during [10,15] or [20,30], but never during]15,20[. If [10,30] is directly used as the time range, the original real-time behavior of the system is extended. Therefore a detailed observation must be introduced to detect the time holes.

For a given observed range [min, max] of transition T, at its i^{th} occurrence, the assertion $check_k$ "*exist* T_i *between k and k+1*" will be checked for all $min \leq k < max$. If $check_k$. If the check does not pass, the time range will be broken into two sections: [min, k] and [k+1,max]. To be more general, if $check_{k_1}$, $check_{k_2}$, ... $check_{k_n}$ do not pass, the final refined equivalent time ranges of this occurrence will become [min, k_1], [$k_1 + 1$, k_2], ..., [$k_n + 1$, max]. Accordingly, the sequential transition of the equivalent sub-net will be refined to a sub-structure which

contains branches representing all possible firing time range after removing those impossible ranges. An example in Fig. 6 (a) shows that the transition T in the reduced sub-net A exhibits a firing time range $[t_3, t_4]$. But there exists time holes on this time range, and thus the real time behavior is $[t_3, t'_3] \cup [t'_4, t_4]$, where $t'_3 < t'_4$. The transition T should be replaced by the sub-range structure (grey part in Fig. 6 (b)).

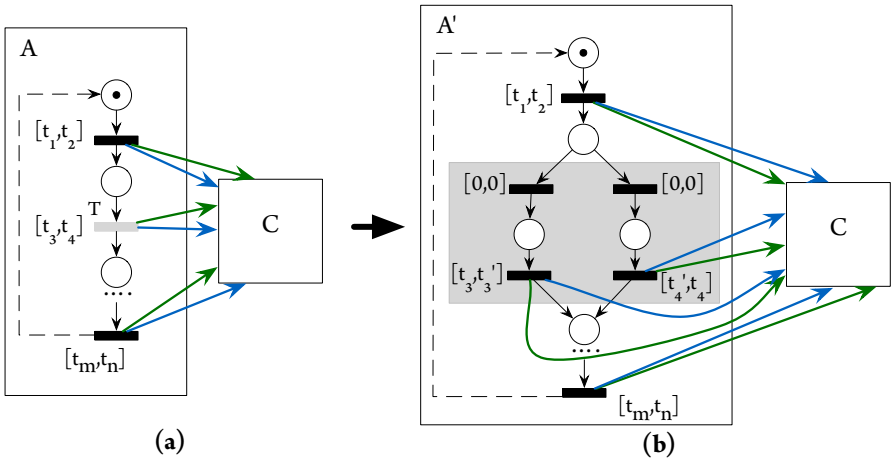


Figure 6. Deal with Holes on Time Range

7 Experimental Results

To experiment the property specific reduction method, we use an avionic case study investigated by M. Lauer et al. [23], which is a part of a flight management system (`FMS`). The `FMS` consists of two units, a control display unit and a computer unit. The control display unit provides human/machine interface for data entry and information display. The computer unit provides both computing platform Integrated Modular Avionics (`IMA`) and various interfaces to other avionics. The communication between modules is implemented by Avionic Full DupleX(*AFDX*). `FMS` uses redundant implementation of its functions.

The latency requirement is assessed in the case study. It depends on the functional chain in Fig. 7. At any time, the pilot can request some information on a given waypoint. The KU_1 (*Keyboard and Cursor Control Unit*) controls the physical device used by the pilot to enter his requests. When KU_1 receives a request (req_1), it broadcasts $wpid_1$ and $wpid_2$ to the *Flight Managers* FM_1 and FM_2 respectively. The *FMs* manage the flight plan, i.e., the trajectory between successive waypoints. When a request occurs, both query the *NDB* (*Navigation Database*) by sending $query_1$ (resp. $query_2$) to retrieve the static information

on the waypoint such as the latitude and the longitude. The *NDB* separately answers each *FM* by sending a message $answer_1$ (resp. $answer_2$) containing the expected data. Upon reception of this message, each *FM* computes two complementary dynamic data: the distance to the waypoint, and the *ETA* (*Estimated Time of Arrival*). These data ($wpInfo_1$ and $wpInfo_2$ resp.) are periodically sent to respective *MFDs* (*Multi Functional Display*) which periodically elaborate the pages to be displayed on the screens. The KU_1, *FMs*, *NDB*, *MFDs* are asynchronous functional modules.

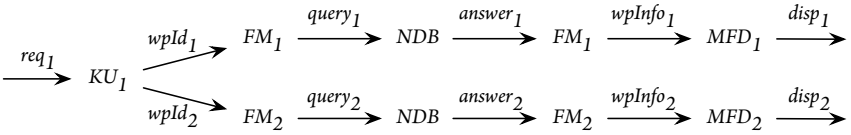


Figure 7. Functional Chain: Sporadic Response to Request

The latency requirement guarantees that the system responds quick enough to a request. It corresponds to the time elapsed between pilot's request (req_1) and the first occurrence of the display signal depending on req_1 ($disp_1$). Therefore, the real-time property here is the worst case time (`WCT`) and best case time (`BCT`) between req_1 and the first occurrence of $disp_1$ depending on req_1.

We model the functional chain in `TPN`. The `WCT` and `BCT` observers are added respectively to the `TPN`. A binary search algorithm is used to search for the bound values. The computation results (verified under MacOS 10.6.8 with a processor 2.4 GHz Intel Core 2 Duo) are shown in Table 2. The `WCT` (resp. `BCT`) is 450.4 (reps. 75.2) `ms`. By applying the reduction approach, the state space is significantly reduced. Take the `WCT` for example, compared to the verification time 278.313 s before reduction, the verification time is reduced to 2.484 s.

Table 2. Real-Time Property Verification Results

Property		**Property Value (ms)**	**State/Transition Number**		**Execution Time (s)**	
			Before Reduc.	After Reduc.	Before Reduc.	After Reduc.
Latency	System	`N/A`	9378/23250	`N/A`	`N/A`	`N/A`
	`WCT`	450.4	67105/145024	9/10	278.313	2.484
	`BCT`	75.2	11162/28922	8/9	43.781	3.719

To test the scalability, the functional chain is enlarged by increasing the number of *NDB*. Each functional chain traverses P *NDB*, i.e. $2P + 3$ functions.

$$L_1 = \xrightarrow{req_1} KU_1 \xrightarrow{wpId_1} FM_1 \xrightarrow{query_1} NDB_1 \xrightarrow{query_2} \ldots \xrightarrow{query_{P-1}} NDB_{P-1} \xrightarrow{query_P} NDB_P \xrightarrow{answer_P} NDB_{P-1} \xrightarrow{answer_{P-1}} \ldots \xrightarrow{answer_2} NDB_1 \xrightarrow{answer_1} FM_1 \xrightarrow{wpInfo_1} MFD_1 \xrightarrow{disp_1} \quad (2)$$

Before apply this reduction method, the state space begins to explode even the *NDB* number is 2 under the test environment. By increasing P from 1 to 11, we give out the state/transition number, reduction time, model checking (MC) time and solving time after applying the reduction method in Table 3. The reduction result is prominent. The solving time is almost linear with respect to the system's scale. This case study shows that after reduction, the explosive systems can be analyzed, if the systems conform to the behavioral regularities.

Table 3. Scalability Test for Latency Property

***NDB*/Fun.**	**State/Tran (after Red.)**		**Reduction Time (s)**	**MC Time (s)**		**Solving Time (s)**	
	WCT	BCT		WCT	BCT	WCT	BCT
1/7	9/10	8/9	38.049	2.484	1.860	40.533	39,909
2/8	9/10	8/9	57.876	2.656	1.883	60.532	59,759
3/9	9/10	6/5	79.813	2.812	2.079	82.625	81,892
4/10	9/10	6/5	102.500	2.906	2.079	105.406	104,579
5/11	9/10	6/5	124.987	3.015	2.102	128.002	127,089
6/12	9/10	6/5	149.359	2.891	2.196	152.250	151,555
7/13	9/10	6/5	169.607	2.953	2.227	172.560	171,834
8/14	9/10	6/5	193.329	3.031	2.250	196.360	195,579
9/15	9/10	6/5	216.239	3.000	2.211	219.239	218,45
10/16	9/10	6/5	239.953	3.047	2.195	243.000	242,148
11/17	9/10	6/5	263.049	3.188	2.195	266.237	265,244

8 Computation Complexity & Applicability

This method turns the combination problem of $O(N \cdot M)$ into a divide-and-conquer problem of $O(t_{iden} + n \cdot N + M \cdot N')$, where

- N is the state unfolding complexity of the target sub-net,
- M is the complexity of the other parts of the TPN,
- N' is the state unfolding complexity of the reduced sub-net, $1 \leq N' \leq N$. It is expected that $1 \leq N' \ll N$ if the system conforms to the behavioral regularity.
- t_{iden} is the time for identification, it is $O(N_S^2)$, where N_S is the number of places and transitions in the TPN system.
- n is unfolding times of A by the reduction, refinement and cavity detection
  - Finite case reduction: $2N_B^4 \cdot A_{obs}$, N_B is the defined bound value of occurrence times, A_{obs} is the unfolding time of A with observer.
  - Infinite case reduction: $2N_B^4 \cdot A_{obs}$, N_B is the defined bound value of occurrence times.
  - Refinement: $(n_S + n_L) \cdot A_{obs}$, n_S is the length of sequential section, n_L is the length of loop section.
  - Cavity Detection: $\sum_{i=1}^{n_S+n_L} (max_i - min_i) \cdot A_{obs}$

This method relies on the observers, it may thus take time to search for the bound values of time ranges. In some cases, if the system does not conform to the behavioral regularity, it can only be known after performing the reduction and refinement methods. As our purpose is to reduce the state space of model checking, the trade-off between computation time and the state space is acceptable, except that the computation time is out of the predefined thread-hold value. This is then an engineering problem.

9 Conclusion

This paper proposes a real-time property specific reduction approach for `TPN` based model checking. We illustrate the reduction method for the *one-way-out pattern*. More generic pattern with one incoming portal transition and one outgoing transition uses different identification function, but similar reduction and refinement functions. This method makes the verification more scalable for systems conforming to some behavioral regularities. It makes a trade-off between the state space and the solving time, and allows to verify large scale systems that will easily encounter combinatorial explosion problem, especially for the asynchronous real-time systems. The case study shows that after reduction, the explosive systems can be analyzed, if the systems conform to the behavioral regularities. The reduction and refinement functions rely on the real-time property specification and observer-based verification approaches. For now, we have defined two behavioral regularities for the finite and infinite firing occurrence, and provided reduction methods for the pattern with an eventual sequential section and a loop section. Other real-time behavioral regularities are under study. Similar approaches can be studied to reduce the state space for verifying other families of properties.

Acknowledgment

This work was funded by the FUI P and OpenETCS projects. We also wish to thank Michaël Lauer and Frédéric Boniol for the sharing of the avionic case study.

References

1. Valmari, A.: A stubborn attack on state explosion. In: Computer-Aided Verification, Springer (1991) 156–165
2. Godefroid, P., van Leeuwen, J., Hartmanis, J., Goos, G., Wolper, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Volume 1032. Springer Heidelberg (1996)
3. Misra, J., Chandy, K.M.: Proofs of networks of processes. Software Engineering, IEEE Transactions on (4) (1981) 417–426
4. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3) (1994) 843–871

5. Clarke, E.M., Enders, R., Filkorn, T., Jha, S.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design **9**(1-2) (1996) 77–104
6. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal methods in system design **9**(1-2) (1996) 105–131
7. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(5) (1994) 1512–1542
8. Holzmann, G.: On-the-fly model checking. ACM Computing Surveys (CSUR) **28**(4es) (1996) 120
9. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool tina - construction of abstract state spaces for Petri nets and time Petri nets. International Journal of Production Research **42**(14) (2004) 2741–2756
10. Sloan, R.H., Buy, U.: Reduction rules for time Petri nets. Acta Informatica **33**(7) (1996) 687–706
11. Berthelot, G.: Transformations et analyse de réseaux de Petri: application au protocoles. Rapports de recherche / Université de Paris-Sud, Laboratoire de recherche en informatique. LRI (1983)
12. Berthelot, G., et al.: Checking properties of nets using transformations. In: Advances in Petri Nets 1985. Springer (1986) 19–40
13. Haddad, S.: A reduction theory for coloured nets. In Rozenberg, G., ed.: Advances in Petri Nets 1989. Volume 424 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1990) 209–235
14. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT press (1999)
15. Berthomieu, B., Vernadat, F.: State class constructions for branching analysis of time Petri nets. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2003) 442–457
16. Merlin, P., Farber, D.: Recoverability of communication protocols–implications of a theoretical study. Communications, IEEE Transactions on **24**(9) (1976) 1036 – 1043
17. Cassez, F., Roux, O.H.: Structural translation from time Petri nets to timed automata. JSS **79**(10) (October 2006) 1456–1468
18. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE Trans. Softw. Eng. **17**(3) (March 1991) 259–273
19. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering. ICSE '99, ACM (1999) 411–420
20. Konrad, S., Cheng, B.H.: Real-time specification patterns. In: Proceedings of the 27th international conference on Software engineering, ACM (2005) 372–381
21. Ge, N., Pantel, M.: Time properties verification framework for UML-MARTE safety critical real-time systems. In: Modelling Foundations and Applications. Springer (2012) 352–367
22. Ge, N., Pantel, M., Crégut, X.: Formal specification and verification of task time constraints for real-time systems. In: Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. Springer (2012) 143–157
23. Lauer, M.: Une méthode globale pour la vérification d'exigences temps réel - Application à l'Avionique Modulaire Intégrée. PhD thesis, INPT (juin 2012)

Visual Language Plans - Formalization of a Pedagogical Learnflow Modeling Language

Kerstin Irgang[1] and Thomas Irgang[2]

[1] Human-Centered Information Systems, Clausthal University of Technology,
kerstin.pfahler@tu-clausthal.de
[2] Department of Software Engineering and Theory of Programming, Fernuniversität Hagen,
thomas.irgang@fernuni-hagen.de

Abstract. In this paper we present an approach to support selfregulated learnflows in the collaborative environment Metafora. In this environment students construct Visual Language Plans. Those plans model workflows of learning activities, which the students execute to solve complex learning scenarios across different tools.
Visual Language plans were already used in the context of different pedagogical studies but have no formal syntax or semantics, yet. In this paper, we present the syntax of Visual Language Plans and develop a mapping from Visual Language Plans to Petri net defining semantics. With the help of this semantics, the environment can support the students executing their learnflows. If students execute activities given in a Visual Language Plan which are not enabled in the corresponding Petri net, feedback messages occur guiding the students. Students can refine their Visual Language Plan during execution. If a plan changes the corresponding Petri net model also changes. Analyzing the newly generated Petri net model can help to uncover faulty states of the learnflow model.

1 Introduction

On the one hand, most schools and universities use traditional eLearning systems like Moodle or Ilias. These eLearning systems, which work like Groupware systems in industry, support workflows and offer a lot of evaluation tools, but miss most of the Web 2.0 features and collaboration. On the other hand the students use interactive and collaborative Web 2.0 systems like Facebook or Twitter. New pedagogical approaches try to close this gap and benefit from supporting collaboration between students.

The Metafora project [1] developed a *Computer Supported Collaborative Learning* (CSCL) system [2], which takes up the advantages of social networking technologies. The project was co-funded by the European Union under the 7th Framework Program for R&D, with several partners on the technological and pedagogical side. In the Metafora system, students between 12 to 16 years learn math and science in an enjoyable and selfregulated way, working collaboratively in groups of 3 to 6 members on a complex challenge they have to solve. One aspect of the project was to developed a so-called *Visual Language* [3] for planning and executing of learning activities. One of the main goals of this language is to serve the *Learning to learn together* (L2L2) [4] approach (see e.g. [5]). The publications [6,7] describe the L2L2 approach in detail.

In the Metafora system a group of students gets a challenge which they solve collaboratively. Therefore the students build a Visual Language Plan, consisting of different cards and arcs. This plan describes and documents their approach to the challenge. Teachers can create new challenges. A step of creating a challenge is to select the set of available cards [8]. The Metafora system supports the students in modeling and executing their Visual Language Plan. When the students execute their plan they use it to join collaborative instances of the used tools and document the current state of their execution. When the students finished their plan they reflecting about their process with the help of the plan. Figure 1 shows an example of a Visual Language Plan.

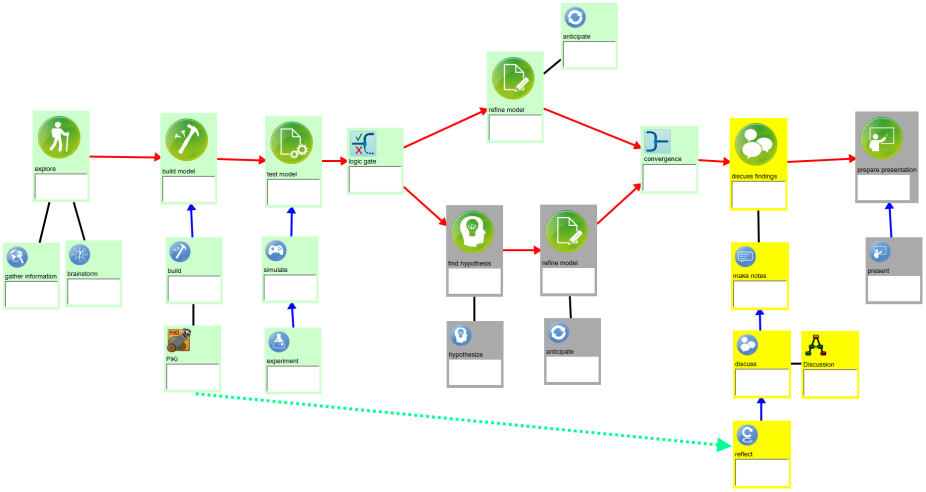


Fig. 1. An example of a Visual Language Plan.

The Visual Language Plans have a graphical representation and consists of nodes and arcs. This nodes are cards. Different types of arcs connecting these cards. A card represents a learning activity. Cards are in one of the three states idle, started or finished. The Visual Language Plan depicted in Figure 1 was developed by students solving the challenge *The bouncing cannon ball* [9]. The students see the bouncing of a cannon ball and simulate how changing variables like angle and speed change the trace of the ball. Students explore the phenomenon with the integrated, game-based domain tool *PiKI* (Pirates of the Kinematics Island). In PiKI students fire cannon balls from a pirate ship to an island trying to hit treasures. To solve the challenge the students create a plan and follow it step by step. In case of the plan depicted in Figure 1 they start with exploring the challenge by gathering for information on bouncing effects and brainstorming ideas. Afterwards the students build a model with their ideas in the domain-tool *PiKI*. In the next step, the students test their model with simulating and experimenting on it. Depending on the results of their test, the students rather go ahead directly to refining their model, or looking for a new hypothesis on the effect of changing variables of the ball if the test failed. Nevertheless, the students discuss their findings afterwards, by reflecting

on the previous results in PiKI. The students use the integrated, graphical discussion environment LASAD [10] for their discussion and make notes about the findings. Finally, the students create a presentation and present their results to other groups.

Although the Visual Language is very intuitive, it does not have a formal syntax or semantics. Both are essential requirements to support the students by modeling a Visual Language Plan. The papers [5] and [11] introduce the core elements of the Visual Language, but they give no construction rules. First syntactic rules were already defined in [12]. Now, we fine grain and formalize the syntax of the Visual Language and develop semantics rules. To define the semantics of a Visual Language Plan we give a mapping to a Petri net and exploit the occurrence rule of Petri nets. There are other approaches using Petri nets to model learnflows. The paper [13] considering the teacher as the expert who models a learnflow which students execute. This approach already had the advantage of an executable model, which allowed simulation and usage of a workflow engine, but teachers did not accept it and the students did not understand the model. To overcome this, the Metafora project developed Visual Language Plans as an intuitive graphical representation for learnflows and examined it in classrooms with teachers and students.

We organized the paper as follows. In Section 2, we give the required definitions of Petri nets and in Section 3 we develop a mathematical founded syntax for Visual Language Plans. Section 4 describes the mapping of Visual Language Plans to Petri nets and Section 5 shows our use cases for the new semantics of Visual Language Plans. In Section 6 we sum up our work.

2 Petri nets and their occurence rule

In this paper we use the following notations. With $\mathbb{N}_0$ we denote the non-negative integers, i.e. $\mathbb{N}_0 = 0, 1, 2, \ldots$ and with $|S|$ we denote the cardinality for a finite set S. Petri nets [14] are bipartite graphs of places and transitions which are a good tool to model concurrent systems:

Definition 1 (Petri net). *A Petri net is a 4-tuple $N = (T, P, F, m_0)$, where T is a finite set of transitions, P is a finite set of places, $F \subseteq (T \times P) \cup (P \times T)$ is a finite set of edges and $m_0 : P \to \mathbb{N}_0$ is a marking . The sets T and P are disjoint, i.e. $T \cap P = \emptyset$.*

In graphical representations, we draw the transitions as squares and the places as circles. If an edge between a place and a transition exists, we draw an arrow. We show the marking for a place with small dots drawn in that place. To define the semantics of a Petri net, we use the preset and postset of a transition.

Definition 2 (Pre- and Postset). *Given a Petri net $N = (T, P, F, m_0)$. The preset of a node $n \in (T \cup P)$ is the set $\bullet n := \{n' \in (T \cup P) \mid (n', n) \in F\}$. The postset of a node $n \in (T \cup P)$ is the set $n\bullet := \{n' \in (T \cup P) \mid (n, n') \in F\}$.*

In a Petri net, enabled transition has only marked places in its preset. Only enabled Transitions can occur. If a transition occurs the marking of the Petri net changes. The transition consumes marks from its preset and produces new marks in its postset.

Definition 3 (Occurrence Rule for Petri nets). *Given a Petri net $N = (T, P, F, m)$. A transition $t \in T$ is enabled, iff for all places $p \in {}^\bullet t : m(p) > 0$ holds. The occurrence of t yields the new marking $m' : P \to \mathbb{N}_0$. This marking is:*

$$m'(p) := \begin{cases} m(p) - 1 \text{ , if } p \in {}^\bullet t \text{ and } p \notin t^\bullet \\ m(p) + 1 \text{ , if } p \notin {}^\bullet t \text{ and } p \in t^\bullet \\ m(p) \quad \text{ , else} \end{cases}$$

3 Visual Language Plans

In this section we will introduce and explain the elements of Visual Language Plans. During the Metafora project pedagogues and psychologists of the University of Exeter developed this Visual Language Plans and pedagogues and teachers of the Hebrew University of Jerusalem and the National and Kapodistrian University of Athens evaluated it. There are several pedagogical studies [15,16,17,18,19,20,21,22] using Visual Language Plans, but none of these studies defines a consistent syntax for it. [5] presents an overview of the used definitions of Visual Language Plans during the Metafora project. The unpublished *Guideline for the Visual Language* [23] gives an informal description of the syntax and we will formally define it in this section.

To solve Metafora challenges students build their own Visual Language Plan which describes their approach to the problem. Such a Visual Language Plan consist of nodes and different types of arcs. Nodes of a Visual Language Plan are cards. The meaning of a card depends on their label and this label belongs to the Visual Language. At the moment, the Visual Language has about 60 labels and it allows teachers to add further labels. The Visual Language divides those labels into 7 disjoint categories and this categories belong to different detail levels. We use the categories to define the syntax and semantics of the Visual Language. The 7 categories of the Visual Language are Activity Stage, Gate, Activity Process, Resource, Role, Attitude and Other.

Activity Stage Cards labeled with an Activity Stage model main steps. Some available Activity Stage labels are *explore*, *define questions* and *find hypothesis*.

Gate Cards labeled with a Gate direct the control flow between cards labeled with an Activity Stage. The Visual Language contains gates for *and* and *xor*.

Activity Process Cards labeled with an Activity Process model the actions which students do. Some available Activity Process labels are *simulate*, *discuss* and *make notes*. This cards model the activities required for solving a card labeled with an Activity Stage.

Resource Cards labeled with a Resource label are links to persistent instances of microworlds and tools integrated in Metafora. For example, there are labels for the physics pirate game *PiKI*, the algebraic pattern tool *eXpresser* and for the graphical discussion environment LASAD [10].

Role Cards labeled with a Role annotate required roles for other cards. Some available Role labels are *note taker*, *evaluator* and *manager*.

Attitude Cards labeled with an Attitude annotate required mind-sets. Some available Attitude labels are *rational*, *critical* and *creative*.

Other Cards labeled with Other labels annotate domain specific information. At the moment the only available Other labels are the generic labels *text card* and *blank card*.

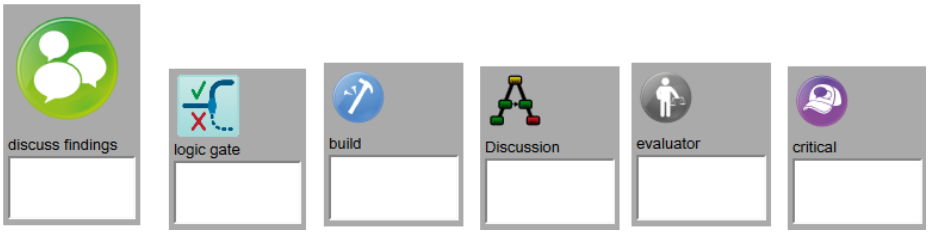


Fig. 2. Examples for cards labeled with labels of the different categories. From the left to the right a card labeled with an *Activity Stage* label, a card labeled with a *Gate* label, a card labeled with an *Activity Process* label, a card labeled with a *Resource* label, a card labeled with a *Role* label and a card labeled with an *Attitude* label.

Figure 2 shows example cards. The category of a cards label is visible through the size, style and coloring of the cards icon. The icons of cards labeled with an *Activity Stage* are larger than other labels to visualize the higher granularity and importance of these cards. To puzzle out the model, the students create labeled card instances and connect it with each other. The students can use 4 different arcs to connect the cards. These arcs are:

is next to The directed *is next to* relation, shown as red arc, models a time sequence of cards. It connects cards labeled with an Activity Stage or a Gate to model the abstract learnflow of a Visual Language Plan. It also orders cards within the set of cards labeled with an Activity Process, within the set of cards labeled with a Resource, within the set of cards labeled with a Role and within the set of cards labeled with an Attitude.

is needed for The directed *is needed for* relation, shown as blue arc, models a time sequence with propagation of resources. It connects cards labeled with an Activity Process, a Role, an Attitude or a Resource with cards labeled with an Activity Process or an Activity Stage. If this relation ends with a card labeled with an Activity Stage the students must finish the other card before they finish the card labeled with an Activity Stage. It also connects cards labeled with a Role or an Attitude with cards labeled with a Role, an Attitude or a Resource and within the set of cards labeled with a Resource.

is input for The directed *is input for* relation, shown as dashed green arc, combines the meaning of the *is next to* and *is needed for* relations. It models propagation of resources to later cards, e.g. for reflection. This relation connects cards labeled with an Activity Process, a Role, an Attitude or a Resource.

is linked to The symmetric *is linked to* relation, shown as black line, models an undirected relation between cards. It associates cards labeled with an Attitude or a Role

to cards modeling activities. This relation connects cards labeled with an Activity Process, a Role, an Attitude or a Resource with each other or with cards labeled with an Activity Stage. If it connects a card with a card label with an Activity Stage the students must finish this card before they finish the card labeled with the Activity Stage.

The collaborative web application *Planning Tool* implements the Visual Language and is part of the Metafora system [2]. To solve a challenge students use the Planning Tool and create a Visual Language Plan to model their approach. With the help of their Visual Language Plan the students document their work and access persistent instances of the used mircoworlds and tools. At the moment, we develop a tool which aims to support the students in creating and executing their Visual Language Plan. Therefore, we need a mathematical syntax for Visual Language Plans, based on the given description. To define this syntax, we first define a mathematical structure for Visual Language Plans.

Definition 4 (Visual Language Plan Structure). *A Visual Language Plan consists of a finite set C of labeled cards and 4 relations.*

The set C is the union of the pairwise disjoint sets C_{AS}, C_G, C_{AP}, C_{Re}, C_{Ro}, C_{At} and C_O, where C_{AS} is a set of cards labeled with an Activity Stage, C_G is a set of cards labeled with a Gate, C_{AP} is a set of cards labeled with an Activity Process, C_{Re} is a set of cards labeled with a Resource, C_{Ro} is a set of cards labeled with a Role, C_{At} is a set of cards labeled with an Attitude and C_O is a set of cards labeled with an Other label. Further, the set C_G is the union of the finite disjoint sets $C_{G_{and-split}}$, $C_{G_{and-join}}$, $C_{G_{xor-split}}$ and $C_{G_{xor-join}}$.

The 4 relations are a directed is next to *relation $R_{next} \subseteq ((C_{AS} \cup C_G) \times (C_{AS} \cup C_G)) \cup (C_{AP} \times C_{AP}) \cup (C_{Re} \times C_{Re}) \cup (C_{Ro} \times C_{Ro}) \cup (C_{At} \times C_{At})$, a directed* is needed for *relation $R_{need} \subseteq ((C_{AP} \cup C_{Ro} \cup C_{At} \cup C_{Re}) \times (C_{AS} \cup C_{AP})) \cup ((C_{Ro} \cup C_{At}) \times (C_{Ro} \cup C_{At} \cup C_{Re})) \cup (C_{Re} \times C_{Re})$, a directed* is input for *relation $R_{in} \subseteq ((C_{AP} \cup C_{Ro} \cup C_{At} \cup C_{Re}) \times (C_{AP} \cup C_{Ro} \cup C_{At} \cup C_{Re}))$ and a symmetric* is linked to *relation $R_{link} \subseteq (((C \setminus C_G) \times (C \setminus C_G)) \setminus (C_{AS} \times C_{AS}))$.*

A 4-tuple $(C, R_{next}, R_{need}, R_{in}, R_{link})$ is called Visual Language Plan Structure.

This definition for the syntax of Visual Language Plans is incomplete. It allows modeling splits and joins without using cards labeled with Gates and it does not enforce that a Visual Language Plan is weakly connected. We extend this definition to get a unique behavior of the model, avoid error-prone plans and enable validity checking. Therefore, we need the preset and postset of cards.

Definition 5 (Pre- and Postset, Information Preset). *Given a Visual Language Plan Structure $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$. The preset of a card $c \in C$ is the set of cards $\bullet c := \{c' \in C \mid (c', c) \in (R_{next} \cup R_{need})\}$. The postset of a card $c \in C$ is the set of cards $c\bullet := \{c' \in C \mid (c, c') \in (R_{next} \cup R_{need})\}$*

The information preset *$\circ c := \{c' \in C \mid (c', c) \in R_{in}\}$ of a card $c \in C$ is the preset only considering the* input for *relation.*

We call a card labeled with an Activity Stage $c \in C_{AS}$ with no other card labeled with an Activity Stage in its preset an *initial card* and a card labeled with an Activity

Stage $c' \in C_{AS}$ with no other card labeled with an Activity Stage in its postset an *end card*. To ensure a unique meaning, we demand that a Visual Language Plan fulfills following properties:

Definition 6 (valid Visual Language Plan Structure). *Given a Visual Language Plan Structure* $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$. *We call* P valid *if it fulfills all the following properties:*

(I) A visual language plan has one initial card $c_i \in C_{AS}$, *i.e.* $\bullet c_i \cap C_{AS} = \emptyset$, *and one end card* $c_e \in C_{AS}$, *i.e.* $c_e \bullet \cap C_{AS} = \emptyset$. *All other cards* $c \in C_{AS}$ *have one incoming* next to *arc and one outgoing* next to *arc, i.e. for all other cards* $c \in C_{AS} \setminus \{c_i, c_e\}$ *exist two unique cards* $c', c'' \in C_{AS}$ *such that* $(c', c) \in R_{next}$ *and* $(c, c'') \in R_{next}$ *hold.*

(II) All cards $c \in C_{G_{and-split}} \cup C_{G_{xor-split}}$ *have one incoming* next to *arc and two outgoing* next to *arcs, i.e. for all* $c \in C_{G_{and-split}} \cup C_{G_{xor-split}}$ *exist 3 unique cards* $c_1, c_2, c_3 \in C_{AS} \cup C_G$, $c_2 \neq c_3$, *such that* $(c_1, c) \in R_{next}$ *and* $\{(c, c_2), (c, c_3)\} \subset R_{next}$ *hold.*

(III) All cards $c \in C_{G_{and-join}} \cup C_{G_{xor-join}}$ *have two incoming* next to *arcs and one outgoing* next to *arc, i.e. for all* $c \in C_{G_{and-join}} \cup C_{G_{xor-join}}$ *exist 3 unique cards* $c_1, c_2, c_3 \in C_{AS} \cup C_G$, $c_1 \neq c_2$, *such that* $\{(c_1, c), (c_2, c)\} \subset R_{next}$ *and* $(c, c_3) \in R_{next}$ *hold.*

(IV) The number of and *splits is equal to the number of* and *joins, i.e.* $|C_{G_{and-split}}| = |C_{G_{and-join}}|$. *The number of* xor *splits is equal to the number of* xor *joins, i.e.* $|C_{G_{xor-split}}| = |C_{G_{xor-join}}|$.

Property (I) ensures that a Visual Language Plan has a unique card labeled with an Activity Stage as start for the execution and all properties together define a very strict structure for the abstract learnflow model. They are implicit contained in the *Guideline for the Visual Language* [23]. This restrictive structure for the low detail cards of a Visual Language Plan supports the students while building their abstract model. The students refine their abstract model with high detail cards afterwards. Property (IV) ensures, together with the other properties, that every *and*-split is joined with an *and*-join and every *xor*-split is joined with an *xor*-join. This is also a requirement given in [23]. Because of the idea of refinement, we have to find for each card with high detail, i.e. each card not labeled with an Activity Stage or Gate, a card labeled with an Activity Stage. Therefore we need paths in a Visual Language Plan.

Definition 7 (Path in a Visual Language Plan). *Given a Visual Language Plan Structure* $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$ *and a set of arcs* $A \subseteq R_{next} \cup R_{need} \cup R_{in} \cup R_{link}$. *A* directed path *in* P *within* A *from a card* $c_1 \in C$ *to a card* $c_m \in C$ *is a sequence* $\sigma = c_1, \ldots, c_m$ *of cards* $c_i \in C$ *such that for* $1 \leq i \leq m-1 : (c_i, c_{i+1}) \in A$ *holds. A* undirected path *in* P *within* A *from a card* $c_1 \in C$ *to a node* $c_m \in C$ *is a sequence* $\sigma = c_1, \ldots, c_m$ *of cards* $c_i \in C$ *such that for* $1 \leq i \leq m-1 : (c_i, c_{i+1}) \in A$ *or* $(c_{i+1}, c_i) \in A$ *holds.*

In a Visual Language Plan, a card $c \in C_{AS}$ labeled with an Activity Stage is usually refined with the help of other cards. Those other cards are direct or indirect connected to

c with *is needed for* or *is linked to* arcs. We call a card which refines a card labeled with an Activity Stage a subordinated card. From the idea of refinement of cards follows that a card can only be subordinate to one card labeled with an Activity Stage.

Definition 8 (Subordination). *Given a valid Visual Language Plan Structure* $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$ *and a card labeled with an Activity Stage* $c \in C_{AS}$*. We call the set of nodes* $S_{1,c} := (\bullet c \setminus C_{AS}) \cup \{c^* \in C \mid (c^*, c) \in R_{link}\}$ *the* set of first order subordinated cards to *c. A card* $c' \in C$ *is subordinated to c if a card* $c'' \in S_{1,a}$ *and a undirected path* σ *within* $R_{next} \cup R_{need} \cup R_{link}$ *from* c' *to* c'' *exist such that* σ *does not contain the card c.* S_c *is the set of all subordinated nodes of c.*

While executing a plan, students often need achievements from earlier stages to solve later stages. They can use the *is input for* relation to propagate a resource to a later card. This requires that they finished the earlier task before they can start the later task. Further, we demand that the connected cards refine different cards labeled with an Activity Stage.

Definition 9 (Visual Language Plan). *Given a valid Visual Language Plan Structure* $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$*. We call* P *a* Visual Language Plan*, if it fulfills all the following properties:*

(V) P *is weakly connected, i.e. for each two cards* $c, c' \in C$ *exist an undirected path within* $R_{next} \cup R_{need} \cup R_{in} \cup R_{link}$ *from c to* c''*.*

(VI) Each card not labeled with an Activity Stage or a Gate is subordinated to precisely one card labeled with an Activity Stage, i.e. for all cards $c \in C \setminus (C_{AS} \cup C_G)$ *a unique card* $c' \in C_{AS}$ *exist such that* $c \in S_{c'}$ *holds.*

(VII) Each is input for *arc connects cards which are subordinated to different cards labeled with an Activity Stage, i.e. for all* $(c_1, c_2) \in R_{in}$*,* $c_1, c_2 \in C$*, unique different cards* $c', c'' \in C_{AS}$ *exist such that* $c_1 \in S_{c'}$ *and* $c_2 \in S_{c''}$ *holds.*

(VIII) The is next to*,* is needed for *and* is linked to *relations only connect cards not labeled with an Activity Stage which are subordinated to the same card labeled with an Activity Stage, i.e. for each pair of cards* $c_1, c_2 \in C \setminus (C_{AS} \cup G)$ *with* $(c_1, c_2) \in R_{next} \cup R_{need} \cup R_{link}$ *exist a card* $c \in C_{AS}$ *such that* $c_1 \in S_c$ *and* $c_2 \in S_c$ *holds.*

4 Semantics of the visual language

In this section we will define the semantics of a Visual Language Plan with the help of Petri nets. A card is in one of 3 states *idle*, *started* or *done*. A card shows its state with its coloring. The coloring of an idle card is grey, the coloring of a started card is yellow and the coloring of a finished card is green. Through this coloring, the students are able to see what they did and what is next. This is important because most of the Metafora challenges need more than 4 school lessons and include homework sessions. The set of the state of all cards is the state of the plan. This state function maps the set of cards C to $\{1, 2, 3\}$. With this notation 1 means idle, 2 means started and 3 means finished.

Definition 10 (State of a Visual Language Plan). *Given a Visual Language Plan* $P = (C,\ R_{next},\ R_{need},\ R_{in},\ R_{link})$*. The state of* P *is a function* $s : C \to \{1, 2, 3\}$*.*

We call a card $c \in C$ idle *iff* $s(c) = 1$, started *iff* $s(c) = 2$ *and* finished *iff* $s(c) = 3$. *We call a Visual Language Plan finished, if its end card is finished.*

In the *Planning Tool*, a card change its coloring if a student select this card as started or finished. If a student select a card labeled with a Resource as started the linked tool opens in a new Metafora tab. Still, Metafora does not clearly define the semantics of a Visual Language Plan. The *Planning Tool* allows students to mark cards as started or finished without checking any rules. At the moment, the semantics for Visual Language Plans is only given as informal description [23]. To develop a Metafora workflow engine, we need to analyze state changes of cards. Therefore, we require the formal semantics of Visual Language Plans. We extracted the following execution rules from the informal descriptions:

(a) Students must start a card before they can finish it.
(b) Students only can start a card if they have started all cards before that card.
(c) Students can finish a card if all they have finished all cards before that card.
(d) Students can only choose one path after a xor-gate.
(e) Students must solve both paths after an and-gate before they can finish the joining and-gate.
(g) Students must start a card labeled with an Activity Stage before they start their refining cards.
(h) Students must finish all refining cards of a card labeled with an Activity Stage before they can finish the card labeled with an Activity Stage.
(i) For cards connected with the *is input for* relation, students can only start the successor if they finished the predecessor before.

These rules overlap and interfere. For example there is a clash of rule (b) and (d) for the joining card of an xor-split. The meaning of *before* in rule (b) and (c) is different. Through the different detail level of cards it depends on their neighborhood if they can change their state. The occurrence rule for cards of a Visual Language Plan is a complicated logical formula which is expensive to test. To avoid this, we decided to define the semantics of a Visual Language Plan with the help of a Petri net. For our mapping, we consider the starting and finishing of cards as events. In the Petri net to a Visual Language Plan transitions represent this events. If we only consider the starting of cards, the Petri net roughly looks like the Visual Language Plan. If we only consider the finishing of cards, the Petri net roughly looks like the Visual Language Plan, too. Places and arcs which control the learnflow connect these parts. In the following we will give step by step a mapping for a Visual Language Plan $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$ to a Petri net $N = (S, T, F, m_0)$. The Petri net N defines the semantics of the plan P. The following steps lead to the corresponding Petri net N:

Step 1: Choose a fixed enumeration for all cards $c \in C$, i.e. a bijection $M : C \to \mathbb{N}_0$, and add for each card $c \in C$ two transitions $t_{M(c),s}$ and $t_{M(c),f}$ to the net N. For a card $c \in C$, the transition $t_{M(c),s}$ represents the starting of c and the transition $t_{M(c),f}$ represents the finishing of c. To make sure that the finishing event of a card can only occur after the starting event, add for each card $c \in C$ a place $p_{M(c)}$ between $t_{M(c),s}$ and $t_{M(c),f}$, i.e. add a place $p_{M(c)}$ and the two edges $(t_{M(c),s},\ p_{M(c)})$ and $(p_{M(c)},\ t_{M(c),f})$ to N. All places $p_{M(c)}$ are not marked, i.e. $\forall p_{M(c)} \in P\ :\ m_0(p_{M(c)}) = 0$.

Figure 4 shows our enumeration for the example and Figure 3 shows the mapping of a card $c \in C$ to their event transitions.

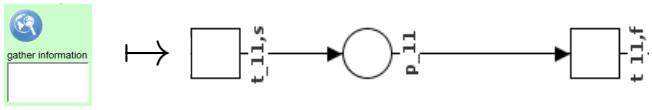


Fig. 3. Step 1: We map the card c with $M(c) = 11$ on two transitions $t_{11,s}$ and $t_{11,f}$ representing their start and finish events and add a places p_{11}.

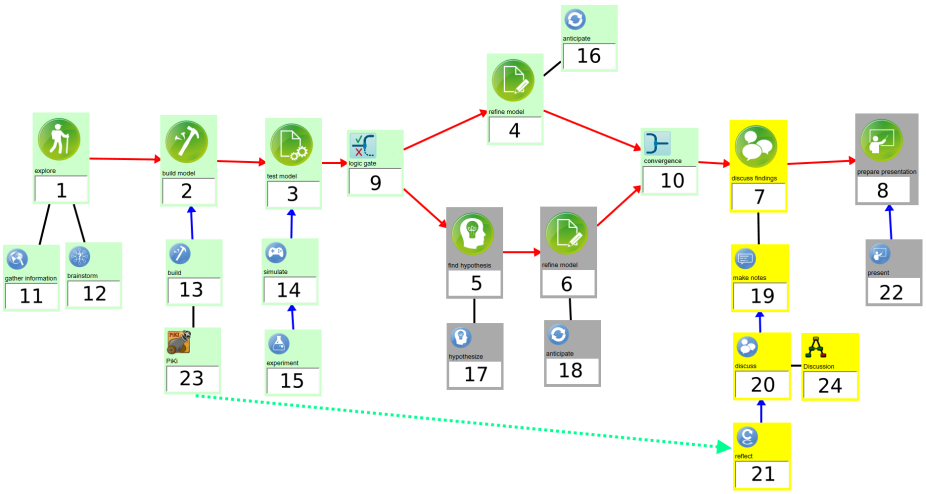


Fig. 4. This figure shows the bijection $M : C \to \mathbb{N}_0$ which we use for our example. For each card $c \in C$ we wrote the value $M(c)$ on that card.

Step 2: To transfer the *is next to* relation, add for each arc $(c, c') \in R_{next}$ a new place $p_{next,M(c),M(c'),s}$ between $t_{M(c),s}$ and $t_{M(c'),s}$, i.e. add $p_{next,M(c),M(c'),s}$ and the edges $(t_{M(c),s}, p_{next,M(c),M(c'),s})$ and $(p_{next,M(c),M(c'),s}, t_{M(c'),s})$ to N. This ensures that the students can only start the succeeding card c' if they started the preceding card c before. Further, add for each arc $(c, c') \in R_{next}$ a place $p_{next,M(c),M(c'),f}$ between $t_{M(c),f}$ and $t_{M(c'),f}$ to the net, i.e. add $p_{next,M(c),M(c'),f}$ and the edges $(t_{M(c),f}, p_{next,M(c),M(c'),f})$ and $(p_{next,M(c),M(c'),f}, t_{M(c'),f})$ to N. This ensures that the finish event of the succeeding card can only occur if the finish event of the preceding card already occurred.

Figure 5 shows the mapping of the *is next to* relation between cards labeled with an Activity Stage.

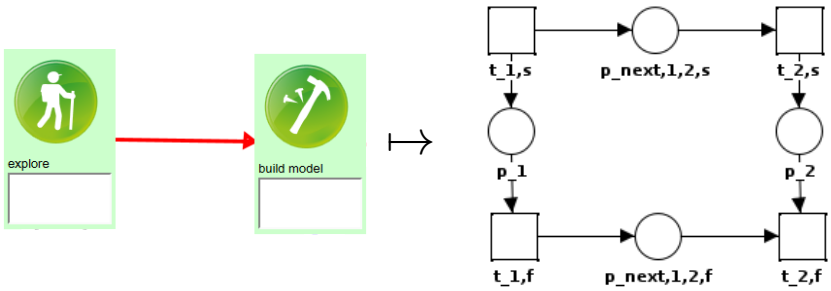


Fig. 5. Step 2: This figure shows how we map the *is next to* relation between the card c labeled with *exlpore*, $M(c) = 1$, and the card c' labeled with *build model*, $M(c') = 2$, to the Petri net. The place $p_{next,1,2,s}$ make sure that c starts before c'. The place $p_{next,1,2,f}$ make sure that c finish before c'.

Step 3: If students use the *is next to* relation with an *XOR* split, they can only start one successor card because of rule (d). To fulfill this rule we add for each card $c \in C_{G_{xor-join}}$ a place $p_{xor,M(c),s}$ between the $t_{M(c),s}$ and the starting event transitions for all cards labeled with an Activity Stage or a Gate in the postset of c, i.e. add $p_{xor,M(c),s}$, $(t_{M(c),s},\ p_{xor,M(c),s})$ and for each card $c' \in c\bullet \cap (C_{AS} \cup C_G)$ an edge $(p_{xor,M(c),s}, t_{M(c'),s})$ to N. Through $p_{xor,M(c),s}$ the places $\{p_{next,M(c),M(c'),s} \mid c' \in c\bullet \cap (C_{AS} \cup C_G)\}$ are superfluous and we removed it. If we would only want to fulfill rule (d) we could stop now but we want to keep the symmetry of the Petri net and avoid useless marked places. Add $p_{xor,M(c),f}$, $(t_{M(c),s},\ p_{xor,M(c),f})$ and for each card $c' \in c\bullet \cap (C_{AS} \cup C_G)$ an edge $(p_{xor,M(c),f},\ t_{M(c'),s})$ to N.

Figure 6 shows the result of this mapping of an xor-split.

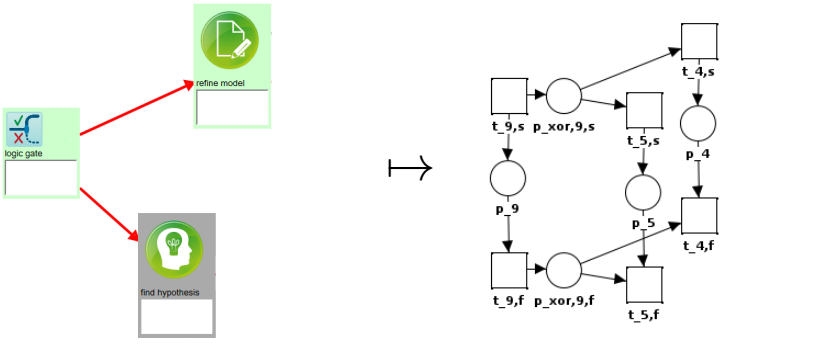


Fig. 6. Step 3: This figure shows the result of the mapping of an xor-split between the cards c labeled with *logic gate*, $M(c) = 9$, the card c' labeled with *refine model*, $M(c') = 4$, and the card c'' labeled with *find hypothesis*, $M(c'') = 5$, to the Petri net.

Step 4: If students use the *is next to* relation with a *XOR* join, they can choose only one path. Due to property (III) there are two preceding cards before this *XOR* join. Because of those two preceding cards, we added two places to N in Step 1. The students can only execute one path before this *XOR* join. This cause a deadlock. To solve this problem melt those two places into one place. For each card $c \in C_{G_{xor-join}}$ remove all places $\{p_{next,M(c'),M(c),s} \mid c' \in {\bullet}c \cap (C_{AS} \cup C_G)\}$ and add a place $p_{xor,M(c),s}$, an arc $(p_{xor,M(c),s}, t_{M(c),s})$ and arcs $\{(t_{M(c'),s},\ p_{xor,M(c),s} \mid c' \in {\bullet}c \cap (C_{AS} \cup C_G)\}$. We also do this for the finish event part of our net. For each card $c \in C_{G_{xor-join}}$ remove all places $\{p_{next,M(c'),M(c),f} \mid c' \in {\bullet}c \cap (C_{AS} \cup C_G)\}$ and add a place $p_{xor,M(c),f}$, an arc $(p_{xor,M(c),f}, t_{M(c),f})$ and arcs $\{(t_{M(c'),f},\ p_{xor,M(c),f} \mid c' \in {\bullet}c \cap (C_{AS} \cup C_G)\}$.

Figure 7 shows the result of this mapping of an xor-join.

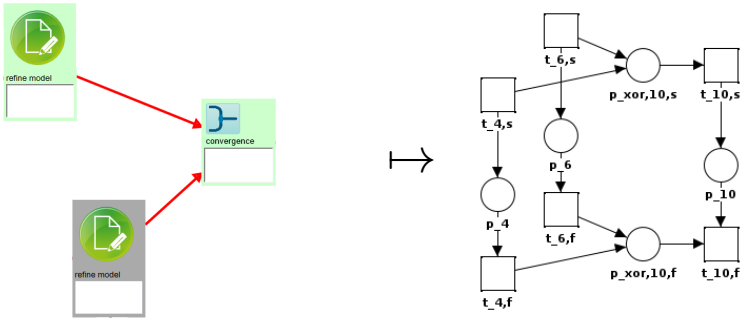


Fig. 7. Step 4: This figure shows the result of the mapping of an xor-join between the cards c labeled with *convergence*, $M(c) = 10$, and the cards c' labeled with *refine model*, $M(c') = 4$, and c'' labeled with *refine model*, $M(c'') = 6$.

Step 5: To transfer the *is needed for* relation, add for each arc $(c, c') \in R_{need}$ a new place $p_{need,M(c),M(c'),s}$ between $t_{M(c),s}$ and $t_{M(c'),s}$, i.e. add $p_{need,M(c),M(c'),s}$ and the edges $(t_{M(c),s}, p_{need,M(c),M(c'),s})$ and $(p_{need,M(c),M(c'),s}, t_{M(c'),s})$ to N. This ensures that students can only start the succeeding card c' after they started the preceding card c. Further, add for each arc $(c, c') \in R_{need}$ a place $p_{need,M(c),M(c'),f}$ between $t_{M(c),f}$ and $t_{M(c'),f}$ to the net, i.e. add $p_{need,M(c),M(c'),f}$ and the edges $(t_{M(c),f}$, $p_{need,M(c),M(c'),f})$ and $(p_{need,M(c),M(c'),f},\ t_{M(c'),f})$ to N. This ensures that the finish event of the succeeding card can only occur if the finish event of the preceding card already occurred.

This mapping is similar to the mapping of the *is next to* relation in Step 1.

Step 6: If students use the *is needed for* relation to model subordination, the places added with the last step enforce that the starting events of the subordinated cards occur before the starting event of the card labeled with an Activity Stage can occur. This is wrong and we remove those places, i.e. for each card $c \in C_{AS}$ remove all places $\{p_{need,M(c'),M(c),s} \mid c' \in S_c\}$. For each card $c \in C_{AS}$, all to c subordinated cards are only allowed to start after c. We enforce this by adding places between c and all to c subordinated cards, i.e. for each $c \in C_{AS}$ and each $c' \in S_c$ add a place $p_{sub,M(c),M(c'),s}$

and edges $(t_{M(c),s}, p_{sub,M(c),M(c'),s})$ and $(p_{sub,M(c),M(c'),s}, t_{M(c'),s})$. Rule (h) raise the requirement that for each card $c \in C_{AS}$ all subordinated cards finish before the finish event of c can occur. We make this sure by adding places between the to c subordinated cards and c, i.e. for each $c \in C_{AS}$ and each $c' \in S_c$ add a place $p_{sub,M(c),M(c'),f}$ and edges $(p_{sub,M(c),M(c'),f}, t_{M(c),f})$ and $(t_{M(c'),f}, p_{sub,M(c),M(c'),f})$.

Figure 8 shows the result of this step.

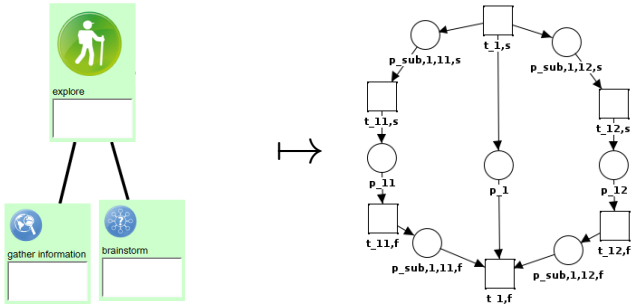


Fig. 8. Step 6: This figure shows the mapping of subordination of the cards c' labeled with *gather information*, $M(c) = 11$, and c'' labeled with *brainstorm*, $M(c'') = 12$, to the card c labeled with *explore*, $M(c) = 1$.

Step 7: To transfer the *is input for* relation, add for each arc $(c, c') \in R_{in}$ a new place $p_{in,M(c),M(c')}$ between $t_{M(c),f}$ and $t_{M(c'),s}$, i.e. add $p_{in,M(c),M(c')}$ and the edges $(t_{M(c),f}, p_{in,M(c),M(c')})$ and $(p_{in,M(c),M(c')}, t_{M(c'),s})$ to N. This ensures that the succeeding card c' can only start after finishing the preceding card c.

Figure 9 shows the mapping of the *is input for* relation.

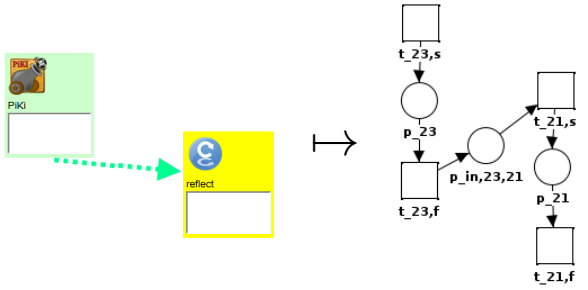


Fig. 9. Step 7: This figure shows how we map the *is input for* relation between the card c labeled with *PiKI*, $M(c) = 23$, and the card c' labeled with *reflect*, $M(c') = 21$, to the Petri net. The place $p_{in,23,21}$ make sure that the students finished c before they can start c'.

Step 8: Finally, two convenience places $p_{initial}$ and p_{end} are added. For the unique initial card c_i add a place $p_{initial}$ and an edge $(p_{initial}, t_{M(c_i),s})$ with $m_0(p_{initial}) = 1$.

The marking of the initial place $p_{initial}$ is 1 if the students did not start executing the Visual Language Plan P. For the unique end card c_e add a place p_{end} and an edge $(t_{M(c_e),f}, p_{end},)$ with $m_0(p_{initial}) = 0$. The marking of this place is 1 if the students finished the Visual Language Plan P.

The *is linked to* relation does not restrict starting or finishing of cards and we do not need to translate it. The language L of this Petri net N is the language of P, i.e. for each transition sequence $\sigma \in L$ starting and finishing of cards according to this sequence is valid for P with respect to the rules given above.

For our example Visual Language Plan we have chosen the mapping shown in Figure 4. Figure 10 shows the Petri net corresponding to this Visual Language Plan which results from the mapping described above. The upper part of the net consists of the transitions which control the starting sequence of the cards labeled with an Activity Stage and has a similar structure as the cards labeled with an Activity Stage in our example. The starting of a card labeled with an Activity Stage enables the start event transitions of their subordinated cards. The Petri net has more places than required and an algorithm for deletion of implicit places could remove $p_{sub,3,14,s}$. It is difficult to decide if a place is an implicit place and we need a fast mapping from the visual language plan to the Petri net so we keep those places. The middle part of the Petri net models the grey, yellow and green sequence. The place $p_{in,23,21}$ for the *is input for* relation from the *PiKI* card to the *reflect* card is also in the middle part. This is the only connection form the lower part of the Petri net to its upper part. The finishing event of the refined cards can only occur after the finishing event of all their refining cards.

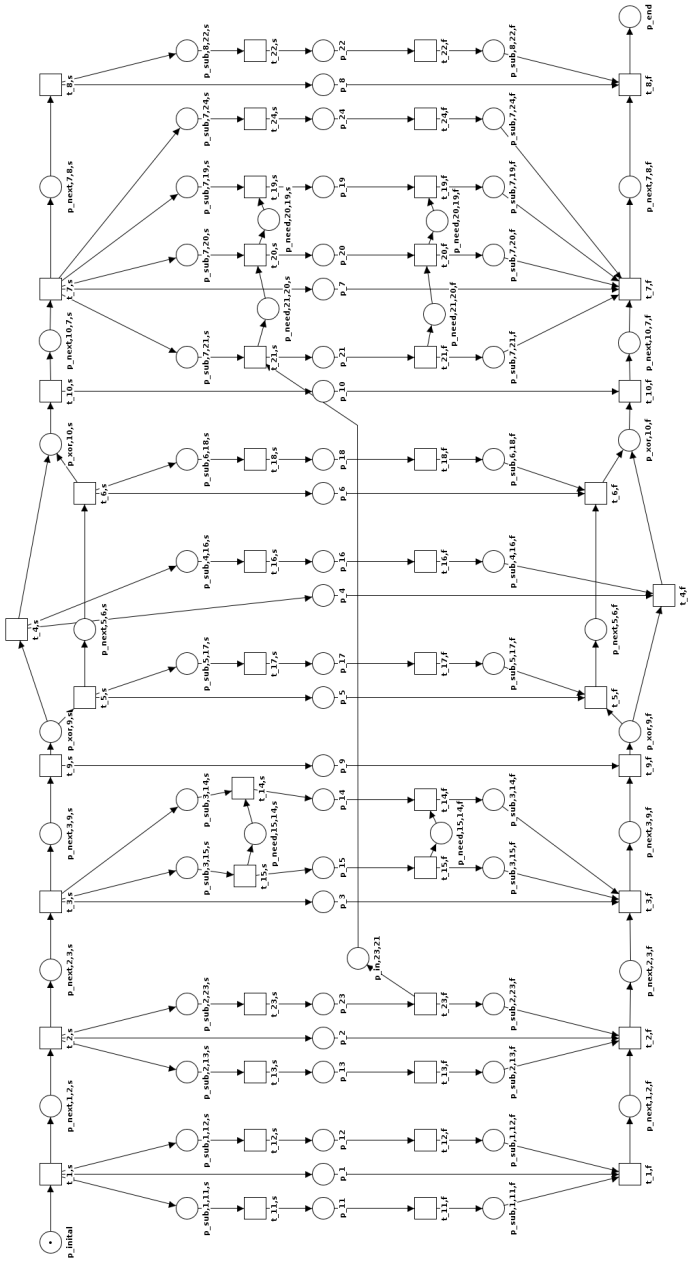


Fig. 10. Petri net for our example. The upper and lower section contain the transitions for the yellow and green occurrence of the activity stage cards. The middle section comprises the transitions for the refining cards.

5 Applications of the Visual Language

With the help of the formal syntax, shown in Section 3, we can automatically check if a Visual Language Plan is valid while students are modeling it. The Metafora learnflow engine we develop will listen to the logs of the *Planning Tool*, analyze the events done by the students and send feedback messages to the students. It will send affirmative feedback messages if the students fulfill desired properties, like refining cards labeled with an Activity Stage. If the students violate syntactic rules it will send corrective feedback messages. For example, this is the case if students connect cards labeled with an Activity Stage with an *is linked to* relation. The feedback messages help the students to create a meaningful and selfregulated learnflow model. Besides all this, we need the formal definition of a syntax for Visual Language Plans to define a formal semantics.

The Petri net mapping for a Visual Language Plan, shown in Section 4, defines a formal semantics for Visual Language Plans. The Metafora learnflow engine will create the Petri net for each Visual Language Plan and use it to analyze the starting and finishing events, done by students to generate helpful feedback messages. The feedback messages are affirmative, corrective or informative. The learnflow engine send an affirmative feedback message if students respect the execution rules, e.g. if a student finished the card labeled with the Activity Stage *explore*. If students violate the execution order of the Visual Language Plan it sends corrective feedback messages. For example, if students start the card labeled with the Activity Stage *test model* before they start the card labeled with the Activity Stage *build model* all students will get a corrective feedback message telling them to build the model first. We use informative feedback messages to tell students working on the same Visual Language Plan about meaningful actions. If Bob and Alice work on the same Visual Language Plan the learnflow engine will send Alice the informative feedback message *'Bob finished build model.'* when Bob changes the state of the card labeled with the Activity Stage *build model* to finished. With these feedback messages we intend to help the students planning and executing their learnflow. We want to shorten the training phase and help the students to concentrate on their learnflow instead of think about occurrence rules for cards. With the help of informative feedback messages we try to help the students keeping track of current state of their learnflow while they use microworlds.

In Metafora, our learnflow engine is not able to enforce the syntax or semantics of a Visual Language Plan. Furthermore, the pedagogical case studies of the Metafora project showed that the students often use their Visual Language Plan for reflecting about their actions and rearrange specific elements to document how the learning actually took place. Reflection is one of the L2L2 behaviors which we support to grant more flexibility on the students side and enable a tight engagement in the planning and execution phases. To do this, we have to change a learnflow during the execution and transfer a state from a Visual Language Plan to its Petri net. The changing of the learnflow or faulty starting and finishing of cards can cause an invalid state of the Visual Language Plan. In case of a faulty state of the Visual Language Plan, a direct mapping would cause a not reachable marking of the Petri net and result in unwanted behavior.

In the following we will describe an approach to transfer a state from a Visual Language Plan to a marking of the corresponding Petri net which can handle faulty states and calculates valuable information to generate useful feedback. In case of a faulty state

change of a card or a change of the learnflow model, we collect the state change events which caused the current state of the Visual Language Plan. This means for a Visual Language Plan $P = (C, R_{next}, R_{need}, R_{in}, R_{link})$ with its state $s : C \to \{1, 2, 3\}$ and the corresponding Petri net $N = (S, T, F, m_0)$ we calculate a set E of transitions which occurred to reach this state. We do this by checking the state of each card $c \in C$ and get the transition set $E = \{t_{M(c),s} \in T \mid c \in C \ \wedge \ s(c) \geq 1\} \cup \{t_{M(c),f} \in T \mid c \in C \ \wedge \ s(c) = 2\}$. Next, we calculate for the net N a valid sequence σ of the transitions contained in E by occurring all enabled transitions, adding them to σ and removing them from E. We do this iterative until E is empty or has no more enabled transitions. For Visual Language Plans, we can do this because we have the state information for joining and splitting cards no conflicts can happen. Now, we have a maximal valid sequence σ of transitions of E, a subset $E' \subseteq E$ of faulty transitions and can easily get a set $A \subseteq T$ of enabled transitions. With this information we can tell the students about the cards with faulty states by analyzing E'. We can recommend possible cards to the students by analyzing A. Moreover, we can calculate a minimal sequence σ', with prefix σ, which enable all transitions $t \in E'$ and recommend steps leading to a valid state of the Visual Language Plan with the help of σ'.

Figure 11 shows our example plan with annotations for the current state of the plan and Figure 12 shows the corresponding Petri net to this plan with a marking corresponding to the current state.

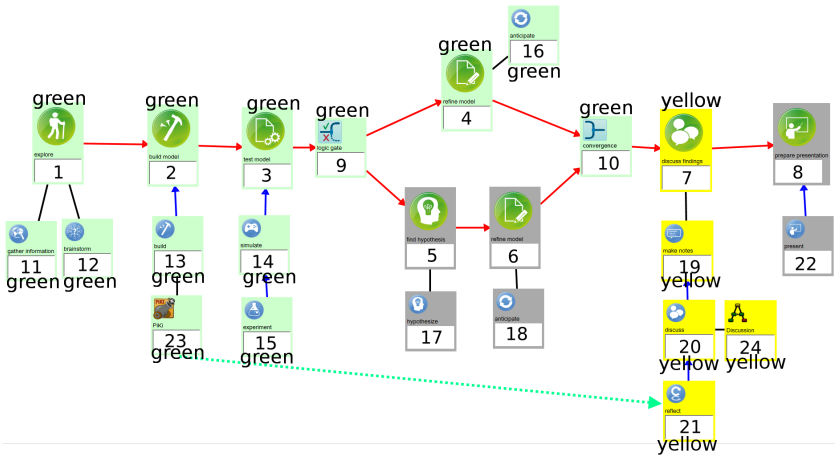


Fig. 11. This figure shows our example Visual Language Plan. For each card $c \in C$ we wrote value $M(c)$ on that card. The cards' states are visible trough the coloring of the cards and we annotated the card with their color for print versions.

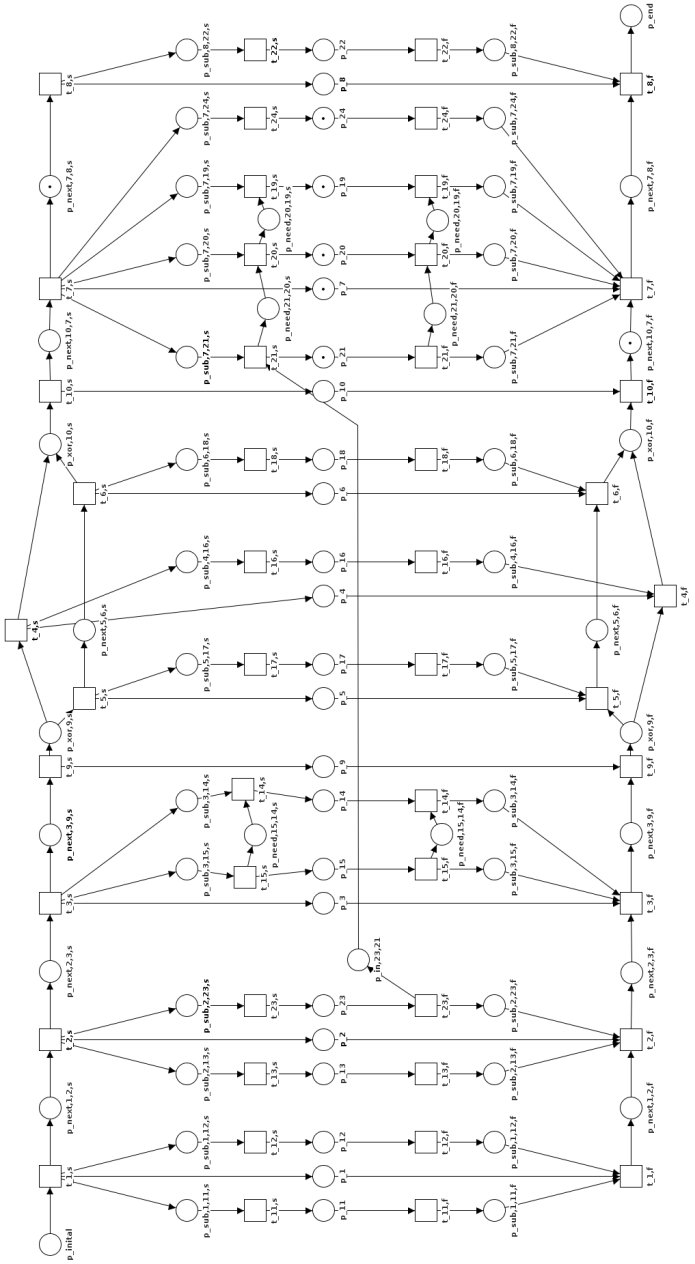


Fig. 12. This figure shows the Petri net for our example shown in Figure 11 with the marking for the state of the plan.

If a student starts the card c labeled with *present* ($M(c) = 22$) the Metafora learnflow engine evaluates this event as occurrence of the not enabled transition $t_{22,s}$. Because of this faulty state change, it calculates the sequence σ of transitions which cause the marking of Figure 12, the set $E = \{t_{22,s}\}$ and the set $A = \{t_{21,f}, t_{20,f}, t_{19,f}, t_{24,f}, t_{8,s}\}$. Now, it unfolds the Petri net with this marking and find the minimal sequence $\sigma' = \sigma, t_{8,s}$. σ' enable all transitions in $E = \{t_{22,s}\}$. Finally, the learnflow engine sends a feedback message recommending to start the card labeled with *prepare presentation*.

6 Conclusion

The Metafora project developed the Visual Language Plans for learnflow modelling. Visual Language Plans support the pedagogy of L2L2. Metaora is a web-based computer supported collaborative learning platform implementing this Visual Language Plans and using Web 2.0 features. The Metafora project did not develop a formal syntax or semantics for this Visual Language Plans.

We develop a Metafora learnflow engine for automatic support of students using the Metafora system. In this paper we give an overview of Visual Language Plans for modeling learnflows. Further, we extracted consistent rules for the syntax and semantics of Visual Language Plans from the available publications and developed a formal syntax and semantics for this plans. To define the semantics of Visual Language Plan we presented a mapping to a Petri net.

The Metafora learnflow engine will analyze events done by students and support them while planning and executing Visual Language Plans. With the syntax for Visual Language Plans the Metafora learnflow engine can generate feedback messages supporting the students in modeling their Visual Language Plan and with the semantics it can generate feedback messages supporting the students while editing and executing their plan. This feedback messages are affirmative, corrective or informative [12]. Furthermore, the corresponding Petri net for a Visual Language Plan enables the Metafora learnflow engine to recommend steps to the students for reaching a valid state.

References

1. Metafora: Project website. `http://www.metafora-project.org/`
2. Metafora: Demo System. `https://www.metafora-project.de/`
3. Metafora Glossary: Visual Language. `http://static.metafora-project.de/VisualLanguage.html`
4. Metafora Glossary: Learning To Learn Together. `http://static.metafora-project.de/L2L2.html`
5. Yang, Y., Wegerif, R., Dragon, T., Mavrikis, M., McLaren, B.M.: Learning how to learn together (L2L2): Developing tools to support an essential complex competence for the internet age. In Rummel, N., Kapur, M., Nathan, M., Puntambekar, S., eds.: CSCL 2013 Conference Proceedings. Volume 2., Madison, International Society of the Learning Sciences (2013) 193–196
6. Dragon, T., Mavrikis, M., McLaren, B.M., Harrer, A., Kynigos, C., Wegerif, R., Yang, Y.: Metafora: A web-based platform for learning to learn together in science and mathematics. Learning Technologies, IEEE Transactions on **6**(3) (2013) 197–207

7. Mavrikis, M., Dragon, T., Yiannoutsou, N., McLaren, B.M.: Towards Supporting 'Learning To Learn Together' in the Metafora platform. Presented at the Intelligent Support for Learning in Groups workshop at the 16th International Conference on Artificial Intelligence in Education (AIED 2013) (2013)
8. Abdu, R., Schwarz, B.: "Metafora" and the fostering of collaborative mathematical problem solving. http://www.academia.edu/1784715/_Metafora_and_the_fostering_of_collaborative_mathematical_problem_solving
9. Metafora: Challenge: The Bouncing Cannon Ball. http://static.metafora-project.de/BouncingCannonBall.html
10. Humboldt Universität Berlin: LASAD. http://cses.informatik.hu-berlin.de/research/details/lasad/
11. Metafora: Report D 2.1 - Visual Language for Learning Processes. http://data.metafora-project.de/reportD2_1.pdf
12. Harrer, A., Pfahler, K., Lingnau, A.: Planning for Life-Educate Students to Plan: Syntactic and Semantic Support of Planning Activities with a Visual Language. In Chen, N.S., Huang, R., Kinshuk, Li, Y., Sampson, D.G., eds.: Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on, Washington, IEEE, CPS (2013) 309–313
13. Bergenthum, R., Desel, J., Harrer, A., Mauser, S.: Learnflow mining. In Seehusen, S., Lucke, U., Fischer, S., eds.: DeLFI 2008. Volume P-132 of LNI., Bonn, Geselschaft für Informatik (2008) 269–280
14. Petri, C.A.: Kommunikation mit Automaten. Dissertation, Technische Hochschule Darmstadt (1962)
15. Smyrnaiou, Z., Moustaki, F., Yiannoutsou, N., Kynigos, C.: Interweaving meaning generation in science with learning to learn together processes using Web 2.0 tools. Themes in Science and Technology Education **5**(1-2) (2013) 27–44
16. Kynigos, C., Moustaki, F.: Designing tools to support group work skills for constructionist mathematical meaning generation. http://data.metafora-project.de/P12_kynigos_moustaki.pdf (2013)
17. Daskolia, M., Yiannoutsou, N., Xenos, M., Kynigos, C.: Exploring Learning-to-learn-together Processes within the Context of an Environmental Education Activity. In: Proceedings of The Ireland International Conference on Education - IICE-2012. (2012)
18. Abdu, R., DeGroot, R., Drachman, R.: Teacher's Role in Computer Supported Collaborative Learning. In: CHAIS conference. (2012) 1–6
19. Smyrnaiou, Z., Varypari, E., Tsouma, E.: Dialogical interactions concerning the scientific content through face to face and distance communication using web 2 tools. In Pintó, R., López, V., Simarro, C., eds.: Computer Based Learning in Science Conference Proceedings 2012, Barcelona, CRECIM (2012) 117–125
20. Pifarré, M., Wegerif, R., Guiral, A., del Barrio, M.: Developing Technological and Pedagogical Affordances to Support Collaborative Inquiry Science Processes. In Demetrios, S.G., Spector, M.J., Ifenthaler, D., Isaias, P., eds.: IADIS International Conference on Cognition and Exploratory Learning in Digital Age, Lisbon, IADIS Press (2012) 139–147
21. Moustaki, F., Kynigos, C.: Meanings for 3d mathematics shaped by online group discussion. In Kynigos, C., Clayson, J.E., Yiannoutsou, N., eds.: Constructionism 2012 Conference - Theory, Practice and Impact, Athens, The Educational Technology Lab (2012) 174–183
22. Yiannoutsou, N., Kynigos, C.: Boundary Objects in Educational Design Research: designing an intervention for learning how to learn in collectives with technologies that support collaboration and exploratory learning. In: Educational design research - Part B: Illustrative cases. SLO, Enschede (2013) 357–381
23. Pfahler, K.: Guideline for the Visual Language. http://data.metafora-project.de/VisualLanguageGuideline.pdf

Slicing High-level Petri Nets

Yasir Imtiaz Khan and Nicolas Guelfi

University of Luxembourg, Laboratory of Advanced Software Systems
6, rue R. Coudenhove-Kalergi, Luxembourg
{yasir.khan,nicolas.guelfi}@uni.lu

Abstract. High-level Petri nets (evolutions of low-level Petri nets) are well suitable formalisms to represent complex data, which influence the behavior of distributed, concurrent systems. However, usual verification techniques such as model checking and testing remain an open challenge for both (i.e., low-level and high-level Petri nets) because of the state space explosion problem and test case selection. The contribution of this paper is to propose a technique to improve the model checking and testing of systems modeled using Algebraic Petri nets (a variant of high-level petri nets). To achieve the objective, we propose different slicing algorithms for Algebraic Petri nets. We argue that our slicing algorithms significantly improve the state of the art related to slicing APNs and can also be applied to low-level Petri nets with slight modifications. We exemplify our proposed algorithms through a case study of a car crash management system.

Key words: High-level Petri nets, Model checking, Testing, Slicing

1 Introduction

Petri nets are well known low-level formalism for modeling and verifying distributed, concurrent systems. The major drawback of low-level Petri nets formalism is their inability to represent complex data, which influences the behavior of a system. Various evolutions of low-level Petri nets (PNs) have been created to raise the level of abstraction of PNs. Among others, high-level Petri nets (HLPNs) raise the level of abstraction of PNs by using complex structured data [17]. However, HLPN can be unfolded into a behaviourally equivalent PNs.

For the analysis of concurrent and distributed systems (including which are modeled using PNs or HLPNs) model checking is a common approach, consisting in verifying a property against all possible states of a system. However, model checking remains an open challenge for both (PNs & HLPNs) because of the state space explosion problem. As systems get moderately complex, completely enumerating their states demands a growing amount of resources which, in some cases, makes model checking impractical both in terms of time and memory consumption [2, 4, 11, 20]. This is particularly true for HLPN models, as the use of complex data (with possibly large associated data domains) makes the number of states grow very quickly.

An intense field of research is targeting to find ways to optimize model checking, either by reducing the state space or by improving the performance of model checkers. In recent years major advances have been made by either modularizing the system or by reducing the states to consider (e.g., partial orders, symmetries). The symbolic model checking partially overcomes this problem by encoding the state space in a condensed way by using *Decision Diagrams* and has been successfully applied to PNs [1, 2]. Among others, Petri net slicing (*PN slicing*) has been successfully used to optimize model checking and testing [3,7,10,12–16,21]. *PN slicing* is a syntactic technique used to reduce a Petri net model based on the given *criteria*. The given *criteria* refer to the point of interest for which the Petri net model is analyzed. The sliced part constitutes only that part of the Petri net model that may affect the anaylsis based on the criteria..

One limitation of the proposed slicing algorithms that are designed to improve the model checking in the literature so far is that most of them are only applicable to low-level Petri nets. Recently, an algorithm for slicing APNs has been proposed [10]. We extend their proposal and introduced a new slicing algorithm. By evaluating and comparing both algorithms, we showed that our slicing algorithm significantly improves the model checking of APNs. Another limitation of the proposed slicing algorithms that are designed to improve the testing is that they are limited to low-level Petri nets. We define a slicing algorithm for the first time in the context of testing for APNs. The objective is to reduce the effort of generating large test input data by generating a smaller net. We highlight the significant differences of different slicing constructions (designed for improving model checking or testing) and their evaluations and applications contexts. Our slicing algorithms can also be applied to low-level Petri nets with some slight modifications.

The remaining part of the paper is structured as follows: in section 2, we give formal definitions necessary for the understanding of proposed slicing algorithms. In section 3, different slicing algorithms are presented together with their informal and formal descriptions. In section 4, we discuss related work and we give a comparison with existing approaches. A small case study from the domain of crisis management system (a car crash management system) is taken to exemplify the proposed slicing algorithms in section 5. An experimental evaluation of the proposed algorithms is performed in section 6. In section 7, we draw conclusions and discuss future work.

2 Basic Definitions

Algebraic Petri nets are an evolution of low-level Petri nets. APNs have two aspects, i.e., the control aspect, which is handled by a Petri net and the data aspect, which is handled by one or many algebraic abstract data types (AADTs) [5, 15,17,18] (Note: we refer the interested reader to [9] for the details on algebraic specifications used in the formal definition of APNs for our work.) .

Definition 1. *A marked Algebraic Petri Net* $APN =< SPEC, P, T, f, asg, cond, \lambda, m_0 >$ *consist of*

∘ *an algebraic specification SPEC = (Σ,E), where signature Σ consists of sorts S and operation symbols OP and E is a set of Σequations defining the meaning of operations,*

∘ *P and T are finite and disjoint sets, called places and transitions, resp.,*

∘ $f \subseteq (P \times T) \cup (T \times P)$, *the elements of which are called arcs,*

∘ *a sort assignment* $asg : P \to S$,

∘ *a function,* $cond : T \to \mathcal{P}_{fin}(\Sigma - equation)$, *assigning to each transition a finite set of equational conditions.*

∘ *an arc inscription function λ assigning to every (p,t) or (t,p) in f a finite multiset over* $T_{OP,asg(p)}$, *where* $T_{OP,asg(p)}$ *are algebraic terms (if used "closed" (resp.free) terms to indicate if they are build with sorted variables closed or not),*

∘ *an initial marking* m_0 *assigning a finite multiset over* $T_{OP,asg(p)}$ *to every place p.*

Definition 2. *The preset of* $p \in P$ *is* ${}^\bullet p = \{t \in T | (t,p) \in f\}$ *and the postset of p is* $p^\bullet = \{t \in T | (p,t) \in f\}$. *The pre and post sets of* $t \in T$ *defined as:* ${}^\bullet t = \{p \in P | (p,t) \in f\}$ *and* $t^\bullet = \{p \in P | (t,p) \in f\}$.

Definition 3. *Let m and m′ two markings of APN and t a transition in T then* $< m, t, m' >$ *is a valid firing triplet (denoted by* $m[t\rangle m'$*) iff*

1) $\forall p \in^\bullet t \mid m(p) \geq \lambda(p,t)$ *(i.e., t is enabled by m).*

2)$\forall p \in P \mid m'(p) = m(p) - \lambda(p,t) + \lambda(t,p)$.

3 Slicing Algorithms

PN slicing is a technique used to syntactically reduce a PN model in such a way that at best the reduced PN model contains only those parts that may influence the property the PN model is analyzed for. Considering a property over a Petri net, we are interested to define a syntactically smaller net that could be equivalent with respect to the satisfaction of the property of interest. To do so the slicing technique starts by identifying the places directly concerned by the property. Those places constitute the *slicing criterion*. The algorithm then keeps all the transitions that create or consume tokens from the criterion places, plus all the places that are pre-condition for those transitions. This step is iteratively repeated for the latter places, until reaching a fixed point. Roughly, we can divide PN slicing algorithms into two major classes, which are *Static Slicing algorithms* and *Dynamic Slicing algorithms*. An algorithm is said to be static if the initial markings of places are not considered for building the slice. Only a set of places is considered as a *slicing criterion*. The *Static Slicing algorithms* starts from the given criterion place and includes all the pre and post set of transitions together with their incoming places. There may exist sequence of transitions in the sliced net that are not fireable because their incoming places are initially empty and do not get markings from any other way. An algorithm is said to be *dynamic slicing algorithm*, if the initial markings of places are considered for building the slice. The *slicing criterion* will utilize the available information of initial markings and produce a smaller sliced net. For a given *slicing criterion* that consists of

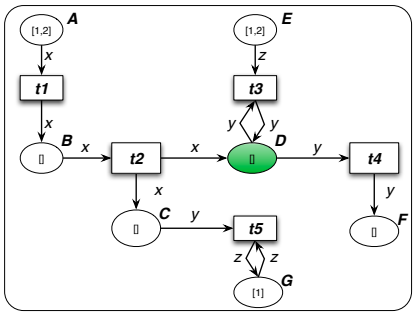


Fig. 1. An example APN model (*APNexample*)

initial markings and a set of places for a PN model, we are interested to extract a subnet with those places and transitions of PN model that can contribute to change the marking of criterion place in any execution starting from the initial marking. The sliced net will exclude sequence of transitions in the resultant slice that are not fireable because their incoming places are not initially marked and do not get markings from any other way.

One characteristic of APNs that makes them complex to slice is the use of multiset of algebraic terms over the arcs. In principle, algebraic terms may contain the variables. Even though, we want to reach a syntactically reduced net, its reduction by slicing, needs to determine the possible ground substitutions of these algebraic terms.

We follow [10] to partially unfold the APN first and then perform the slicing on the unfolded APN. In general, unfolding generates all possible firing sequences from the initial marking of the APN. The AlPiNA tool (a symbolic model checker for Algebraic Petri nets) allows user to define partial algebraic unfolding and presumed bounds for the infinite domains [1], using some aggressive strategies for reducing the size of large data domains. The complete description of the partial unfolding for APNs is out of the scope, for further details and description about the partial unfolding used in our approach, we refer the interested reader to follow [1, 10]. The Fig. 1 shows an APN model, all the places and variables over the arcs are of sort *naturals* (defined in the algebraic specification of the model, and representing the $\mathbb{N}$ set). Since the $\mathbb{N}$ domain is infinite (or anyway extremely large even in its finite computer implementations), it is clear that it is impractical to unfold this net by considering all possible bindings of the variables to all possible values in $\mathbb{N}$. However, given the initial marking of an APN and its structure it is easy to see that none of the terms on the arcs (and none of the tokens in the places) will ever assume any natural value above 3. For this reason, following [1], we can set a *presumed bound* of 3 for the *naturals* data type, greatly reducing the size of the data domain. By assuming this bound, the unfolding technique in [1] proceeds in three steps. First, the data domains of the variables are unfolded up to the presumed bound. Second, variable bindings are computed, and only those are kept that satisfy the guards. Third, the computed

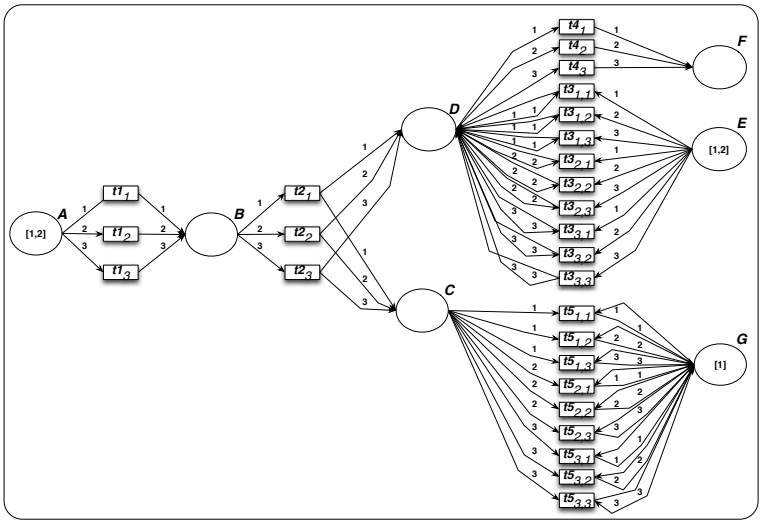


Fig. 2. The unfolded example APN model (*UnfoldedAPN*)

bindings are used to instantiate a binding-specific version of the transition. The resulting unfolded APN model of Fig.1 is shown in the Fig. 2. The transitions arcs are indexed with the incoming and outgoing values of tokens.

3.1 Abstract Slicing on Unfolded APNs

Abstract slicing has been defined as a *static slicing algorithm*. The objective is to improve the model checking of APNs. In the previous static algorithm proposed for APNs, the notions of *reading and non-reading transitions* are applied to generate a smaller sliced net. The basic idea of *reading and no-reading transitions* was coined by Astrid Rakow in the context of PNs [16], and later adapted in the context of APNs in [10]. Informally, the *reading transitions* are transitions that are not subject to change the marking of a place. On the other hand the *non-reading transitions* change the markings of a place (see Fig.3). To identify a transition to be a reading or non-reading in a low-level or high-level Petri nets, we compare the arcs inscriptions attached over the incoming and outgoing arcs. Excluding *reading transitions* and including only *non-reading transitions* reduces the slice size.

Definition 4. ***(Reading(resp.Non-reading) transitions of APN)*** *Let $t \in T$ be a transition in an unfolded APN. We call t a reading-transition iff its firing does not change the marking of any place $p \in ({}^\bullet t \cup t^\bullet)$, i.e., iff $\forall p \in ({}^\bullet t \cup t^\bullet), \lambda(p,t) = \lambda(t,p)$. Conversely, we call t a non-reading transition iff $\lambda(p,t) \neq \lambda(t,p)$.*

We extend the existing slicing operators by introducing the notion of *neutral transitions* and using them with the *reading transitions*. Informally, a *neutral*

transition consumes and produces the same token from its incoming place to an outgoing place. The cardinality of incoming (resp.) outgoing arcs of a neutral traniston is strictly equal to one and the cardinality of outgoing arcs from an incoming place of a neutral transition is equal to one as well.

Definition 5. ***(Neutral transitions of APN)*** *Let $t \in T$ be a transition in an unfolded APN. We call t a neutral-transition iff it consumes token from a place $p \in^\bullet t$ and produce the same token to $p' \in t^\bullet$, i.e., $t \in T \wedge \exists p \exists p' / p \in^\bullet t \wedge p' \in t^\bullet \wedge |p^\bullet| = 1 \wedge |^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge \lambda(t,p) = \lambda(t,p')$.*

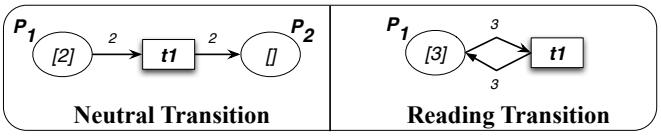


Fig. 3. Neutral and Reading transitions of Unfolded APN

Abstract Slicing Algorithm: The abstract slicing algorithm starts with an unfolded APN and a slicing criterion $Q \subseteq P$ containing criterion place(s). We build a slice for an unfolded APN based on Q by applying the following algorithm:

Algorithm 1: Abstract slicing algorithm

AbsSlicing($\langle SPEC, P, T, F, asg, cond, \lambda, m_0 \rangle, Q$){
$T' \leftarrow \{t \in T / \exists p \in Q \wedge t \in (^\bullet p \cup p^\bullet) \wedge \lambda(p,t) \neq \lambda(t,p)\}$;
$P' \leftarrow Q \cup \{^\bullet T'\}$;
$P_{done} \leftarrow \emptyset$;
while $((\exists p \in (P' \setminus P_{done}))$ **do**
  while $(\exists t \in ((^\bullet p \cup p^\bullet) \setminus T') \wedge \lambda(p,t) \neq \lambda(t,p))$ **do**
    $P' \leftarrow P' \cup \{^\bullet t\}$;
    $T' \leftarrow T' \cup \{t\}$;
  end
  $P_{done} \leftarrow P_{done} \cup \{p\}$;
end
while $(\exists t \exists p \exists p' / t \in T' \wedge p \in^\bullet t \wedge p' \in t^\bullet \wedge |^\bullet t| = 1 \wedge |t^\bullet| = 1 \wedge |p^\bullet| = 1$
$\wedge p \notin Q \wedge p' \notin Q \wedge \lambda(p,t) = \lambda(t,p'))$ **do**
  $m(p') \leftarrow m(p') \cup m(p)$;
  while $(\exists t' \in^\bullet p / t' \in T'$ $)$ **do**
    $\lambda(p'^\bullet, p) \leftarrow \lambda(p'^\bullet, p') \cup \lambda(t', p)$;
  end
  $T' \leftarrow T' \setminus \{t \in T' / t \in p^\bullet \wedge t \in^\bullet p'\}$;
  $P' \leftarrow P' \setminus \{p\}$;
end
return $\langle SPEC, P', T', F|_{P',T'}, asg|_{P'}, cond|_{T'}, \lambda|_{P',T'}, m_{0|_{P'}} \rangle$;
}

In the Abstract slicing algorithm, initially T' (representing transitions set of the slice) contains a set of all the *pre and post* transitions of the given criterion places. Only the *non-reading transitions* are added to T'. P'(representing the places set of the slice) contains all the *preset* places of the transitions in T'. The algorithm then iteratively adds other *preset* transitions together with their *preset* places in the T' and P'. Then the *neutral transitions* are identified and their *pre and post* places are merged to one place together with their markings.

Considering the APN-Model shown in fig. 1, let us now apply our proposed algorithm on two example properties (i.e., one from the class of *safety* properties and one from *liveness* properties). Informally, we can define the properties:

φ_1 : *"The values of tokens inside place D are always smaller than 5".*
φ_2 : *"Eventually place D is not empty".*

Formally, we can specify both properties in the *CTL* as:
$\varphi_1 = \mathbf{AG}(\forall token \in m(D)/token < 5)$.
$\varphi_2 = \mathbf{AF}(|m(D)| \neq \emptyset)$.

For both properties, the slicing criterion $Q = \{D\}$, as D is the only place concerned by the properties. The resultant sliced net can be observed in fig.4, which is smaller than the original unfolded net (shown in fig.2).

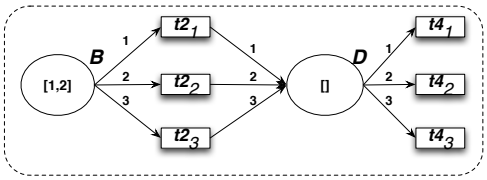


Fig. 4. The sliced unfolded APNs (by applying *abstract slicing*)

Table 1. Comparison of number of states required to verify the property with and without abstract slicing

Properties	*No of states required without slicing*	*No of states required with slicing*
φ_1	*148*	9
φ_2	*148*	9

Let us compare the number of states required to verify the given property without slicing and after applying abstract slicing. In the first column of Table.1, number of states are given that are required to verify the property without slicing and in the second column number of states are given to verify the property by slicing.

The *abstract slicing* can be applied to low-level Petri nets with slight modifications. The *criteria* to build *abstract slice* for both formalisms (i.e., Algebraic Petri nets and low-level Petri nets) remain the same. In case of low-level Petri nets, we do not unfold the net and the slice is built directly. The idea of including *non-reading transitions* together with merging of places by identifying *neutral transitions* remains the same for both formalisms. (Note: we refer the interested reader to [9] for the proof of preservation of properties by applying the *Abstract slicing algorithm.*)

Abstract Slicing on APN without unfolding : *Abstract slicing* extends the previous proposal of APNs slicing by unfolding the APN and then slicing the unfolded APN. One major criticism on *abstract slicing* and previous slicing construction is the complexity of unfolding APNs. As discussed in the previous section, APNs are unfolded to identify the *reading transitions(resp. neutral transitions)* such that a smaller sliced net can be obtained. We can avoid the complexity of unfolding APNs and can perform slicing directly on APNs with a slight trade-off. It is important to note that by applying *abstract slicing* directly on APNs, the sliced net may end up with some *reading transitions (resp.neutral transitions)* included. This is due to he fact that the arc inscriptions are syntactically compared to identify *reading transitions(resp. neutral transitions)* in slicing algorithm. In Fig.5, two *reading transitions(resp. neutral transitions)* can be observed, *abstract slicing* will not consider the transition (shown in the right side of the figure 5) as a *reading transition(resp. neutral transitions)*. This is a slight trade off to avoid the complexity of unfolding. It is a rare situation to have syntactically *non-reading transitions(resp. non-neutral transitions)* which are semantically *reading transitions(resp. neutral transitions)*. The *Abstract slicing algorithm* can be directly appliled to APNs without any change in the syntax.

3.2 Concerned Slicing

Concerned slicing algorithm has been defined as a *dynamic slicing algorithm.* The objective is to extract a subnet with those places and transitions of the APN model that can contribute to change the markings of a given *criterion* place in any execution starting from the initial markings. Concerned slicing can be useful in debugging. Consider for instance that the user is analyzing a particular trace of the marked APN model (using a simulation tool) so that erroneous state is reached.

The *slicing criterion* to build the concerned slice is different as compared to the *abstract slicing* algorithm. In the *concerned slicing* algorithm, available

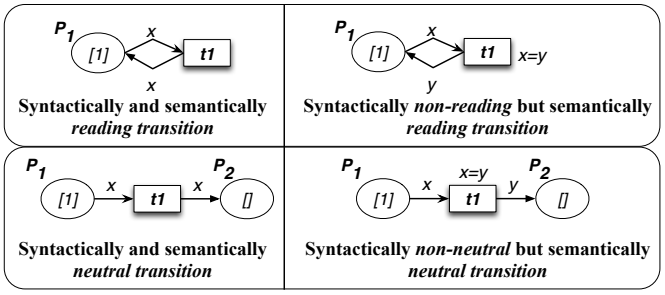


Fig. 5. Syntactically reading (resp. neutral) and non-reading (resp. non-neutral) transitions of APNs

information about the initial markings is utilized and it is directly applied to APNs instead of their unfoldings.

Algorithm 2: Concerned slicing algorithm

ConcernedSlicing($\langle SPEC, P, T, F, asg, cond, \lambda, m_0\rangle, Q$){
$T' \leftarrow \emptyset$;
$P' \leftarrow Q$;
while $({}^\bullet P \neq T')$ **do**
  $P' \leftarrow P' \cup^\bullet T'$;
  $T' \leftarrow T' \cup^\bullet P'$;
end
$T'' \leftarrow \{t \in T'/m_0[t\rangle\}$;
$P'' \leftarrow \{p \in P'/m_0(p) > 0\} \cup T''^\bullet$;
$T_{do} \leftarrow \{t \in T' \setminus T''/{}^\bullet t \subseteq P''\}$;
while $T_{do} \neq \emptyset$) **do**
  $P'' \leftarrow P'' \cup T_{do}^\bullet$;
  $T'' \leftarrow T'' \cup T_{do}$;
  $T_{do} \leftarrow \{t \in T' \setminus T''/{}^\bullet t \subseteq P''\}$;
end
return $\langle SPEC, P'', T'', F_{|_{P'',T''}}, asg_{|_{P''}}, cond_{|_{T''}}, \lambda_{|_{P'',T''}}, m_{0_{|_{P''}}}\rangle$;
}

Starting from the *criterion* place the algorithm iteratively include all the incoming transitions together with their input places until reaching a fix point. Then starting from the set of initially marked places set the algorithm proceeds further by checking the enabled transitions. Then the post set of places are included in the slice. The algorithm computes the paths that may be followed by the tokens of the initial marking.

Considering the APN-Model shown in fig. 1, let us now take the place D as criterion and apply our proposed algorithm on it. The resultant sliced APN-Model is shown in the fig. 6. The test input data can be generated for the sliced APN-model to observe which tokens are coming to the criterion place.

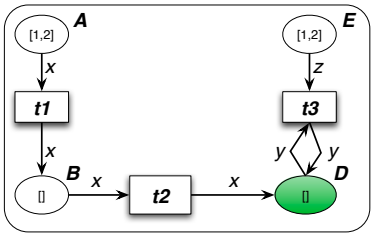


Fig. 6. The sliced APN by applying *concerned slicing*

4 Related Work

The term slicing was coined by M.Weiser for the first time in the context of program debugging [22]. According to Wieser proposal a program slice *(ps)* is a reduced, executable program that can be obtained from a program p based on the variables of interest and line number by removing statements such that ps replicates part of the behavior of a program.

To explain the basic idea of *program slicing* according to Wieser [22], let us consider an example program shown in the Fig.7. The Fig.7(a) shows a program which requests a positive integer number n and computes the sum and the product of the first n positive integer numbers. We take as *slicing criterion* a line number and a set of variables, $C = (line10, \{product\})$.

The Fig.7(b) shows the sliced program that is obtained by tracing backwards possible influences on the variables: In the line 7, *product* is multiplied by i, and in the line 8, i is incremented too, so we need to keep all the instructions that impact the value of i. As a result all the computations that do not contribute to the final value of *product* have been sliced away (The interested reader can find more details about the *program slicing* from [19, 23]).

```
(1)  read(n) ;
(2)  i := 1 ;
(3)  sum := 0 ;
(4)  product := 1 ;
(5)  while i <= n do
        begin
(6)        sum := sum + i ;
(7)        product := product * i ;
(8)        i := i + 1 ;
        end ;
(9)  write (sum) ;
(10) write (product) ;
```

(a) Example program.

```
read(n) ;
i := 1 ;

product := 1 ;
while i <= n do
   begin

      product := product * i ;
      i := i + 1 ;
   end ;

write (product) ;
```

(b) Program slice w.r.t. (10,product).

Fig. 7. An example program and sliced program w.r.t. given *criterion*

The first algorithm about Petri net slicing was presented by Chang et al [3]. They proposed an algorithm on Petri nets testing that slices out all sets of paths, called concurrency sets, such that all paths within the same set should be executed concurrently. Lee et al. proposed a Petri nets slicing approach to partition huge place/transition net models into manageable modules such that the partitioned model can be analyzed by compositional reachability analysis technique [12]. Llorens et al. introduced two different techniques for dynamic slicing of Petri nets [13]. In the first technique, the Petri net and an initial marking is taken into account, but produces a slice w.r.t. any possibly firing sequence. The second approach further reduces the computed slice by fixing a particular firing sequence. Wangyang et al presented a backward dynamic slicing algorithm [21]. The basic idea of the proposed algorithm is similar to the algorithm proposed by Lloren et al, [13]. At first for both algorithms, a static backward slice is computed for a given *criterion* place(s). Secondly, in the case of Llorens et al a forward slice is computed for the complete Petri net model whereas in the case of Wangyang et al, a forward slice is computed for the resultant Petri net model obtained from the static backward slice.

Astrid Rakow developed two algorithms for slicing Petri nets i.e., CTL^*_{-X} slicing and *Safety slicing* in [16]. The key idea behind the construction is to distinguish between *reading and non-reading* transitions. A reading transition $t \in T$ can not change the token count of a place $p \in P$ while other transitions are called *non-reading transitions* as they change the token acount. For the CTL^*_{-X} slicing, a subnet is built iteratively by taking all non-reading transitions of a place P together with their input places, starting with the given criterion place. For the *Safety slicing* a subnet is built by taking only transitions that increase token count on the places in P and their input places. The CTL^*_{-X} slicing algorithm is fairly conservative. By assuming a very weak fairness assumption on Petri net it approximates the temporal behavior quite accurately by preserving all the CTL^*_{-X} properties and for the safety slicing focus is on the preservation of stutter-invariant linear safety properties only.

Khan et al presented a slicing technique for algebraic Petri nets [10]. They argued that all the slicing constructions are limited to low-level Petri nets and cannot be applied as it is to the high-level Petri nets. In order to be applied to high-level Petri nets they need to be adapted to take into account the data types. In algebraic Petri nets (APNs), terms may contain the variables over the arcs from place to transitions (or transitions to places) or guard conditions. Authors proposed to unfold the APN to know the ground substitutions of the variables. They used a particular unfolding approach developed by the SMV group i.e., a partial unfolding [1]. Perhaps, the proposed approach is independent of any unfolding approach. The algorithm proposed for slicing APNs starts by taking an unfolded APN and the criterion places. We use the same strategy for defining static slicing for algebraic Petri nets as proposed by khan et al in [10]. The major difference between their and our slicing construction is that we use the *neutral transition* together with *reading transition* to reduce the slice size (as discussed

in the section 3). We also introduce a notion of dynamic slicing for the first time in the context of APNs.

5 Case Study

We took a small case study from the domain of crisis management systems (car crash management system) for the experimental investigation of the proposed slicing algorithms. In a car crash management system (CCMS); reports on a car crash are received and validated, and a `Superobserver` (i.e., an emergency response team) is assigned to manage each crash.

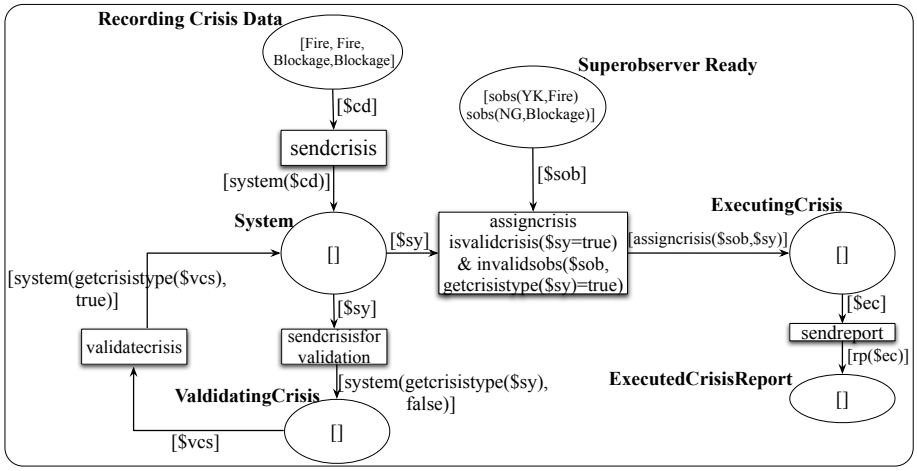


Fig. 8. Car crash APN model

The APN Model can be observed in Fig. 8, it represents the semantics of the operation of a car crash management system. This behavioral model contains labeled places and transitions. There are two tokens in the place `Recording Crisis Data` that are `Fire` and `Blockage`. These tokens are used to mention which type of data has been recorded. The input arc of transition `sendcrisis` takes the `cd` variable as an input from the place `Recording Crisis Data` and the output arc contains term `system(cd)` of sort `sys` (It is important to note that for better readability, we omit $ symbol from the terms over the arcs). The `sendcrisis` transition passes a recorded crisis to system for further operations. All the recorded crises are sent for validation through `sendcrisisforvalidation` transitions. Initially, every recoded crisis is set to false. The output arc of `validatecrisis` contains the `system(getcrisistype(vcs),true)` term which sends validated crisis to system. The transition `assigncrisis` has two guards, the first one is `isvalid(sy)=true` that enables to block invalid crisis reporting to be executed for the mission and the second one is `isvalid(sob,getcrisestype(sy))=`

`true` which is used to block invalid `Superobserver` (a skilled person for handling crisis situation) to execute the crisis mission. The `Superobserver YK` will be assigned to handle `Fire` situation only. The transition `assigncrisis` contains two input arcs with `sob` and `sy` variables and the output arc contains term `assigncrisis(sob,sy)` of sort `crisis`. The output arc of transition `sendreport` contains term `rp(ec)`. This enables to send a report about the executed crisis mission. We refer the interested reader to [6] for the algebraic specification of a car crash management system.

An important safety threat, which we will take into an account in this case study is that the invalid crisis reporting can be hazardous. The invalid crisis reporting is the situation that results from a wrongly reported crisis. The execution of a crisis mission based on the wrong reporting can waste both human and physical resources. In principle, it is essential to validate a crisis that it is reported correctly. Another, important threat could be to see the number of superobservers should not exceed from a certain limit. Informally, we can define the properties:

Formally we can specify the properties as, let `Crises` be a set representing recorded crisis in car crash management system. Let $isvalid : Crises \rightarrow BOOL$, is a function used to validate the recorded crisis.

$\varphi_1 = \mathbf{AF}(\forall crisis \in System | isvalid(crisis) = true)$.

$\varphi_2 = \mathbf{AG}(|SuperobserverReady| \leq 2)$.

In contrast to generate the full state space for the verification of the properties φ_1 *and* φ_2, we alleviate the state space by applying our proposed algorithm i.e., *abstract slicing algorithm*. For $\varphi_1 and \varphi_2$, the criterion places are `System and Superobserver Ready`. The unfolded car crash APN model is shown in the Fig. 9. The abstract slicing algorithm takes *an unfolded car crash APN model* and *System (an input criterion place)* as an input and iteratively builds the sliced net for φ_1. Respectively for φ_2, the algorithm starts from *Superobserver Ready(as input criterion place)* and builds the slice. The sliced unfolded car crash APN models are shown in the Fig. 10, for the both prperties i.e., φ_1 and φ_2.

Let us compare the number of states required to verify the given property without slicing and after applying abstract slicing. In the first column of Table.2, the number of states are given that are required to verify the property without slicing and in the second column the number of states are given to verify the property by slicing.

Let us take a criterion place (i.e, *System*) from the car crash APN model and apply our proposed *concerned slicing algorithm* to find which transitions and places can contribute tokens to that place. It is important to note that, we perform concerned slicing directly on the car crash APN model instead of the unfolded car crash APN model (as discussed in the section 3). The sliced car crash APN-model can be observed in the Fig.11.

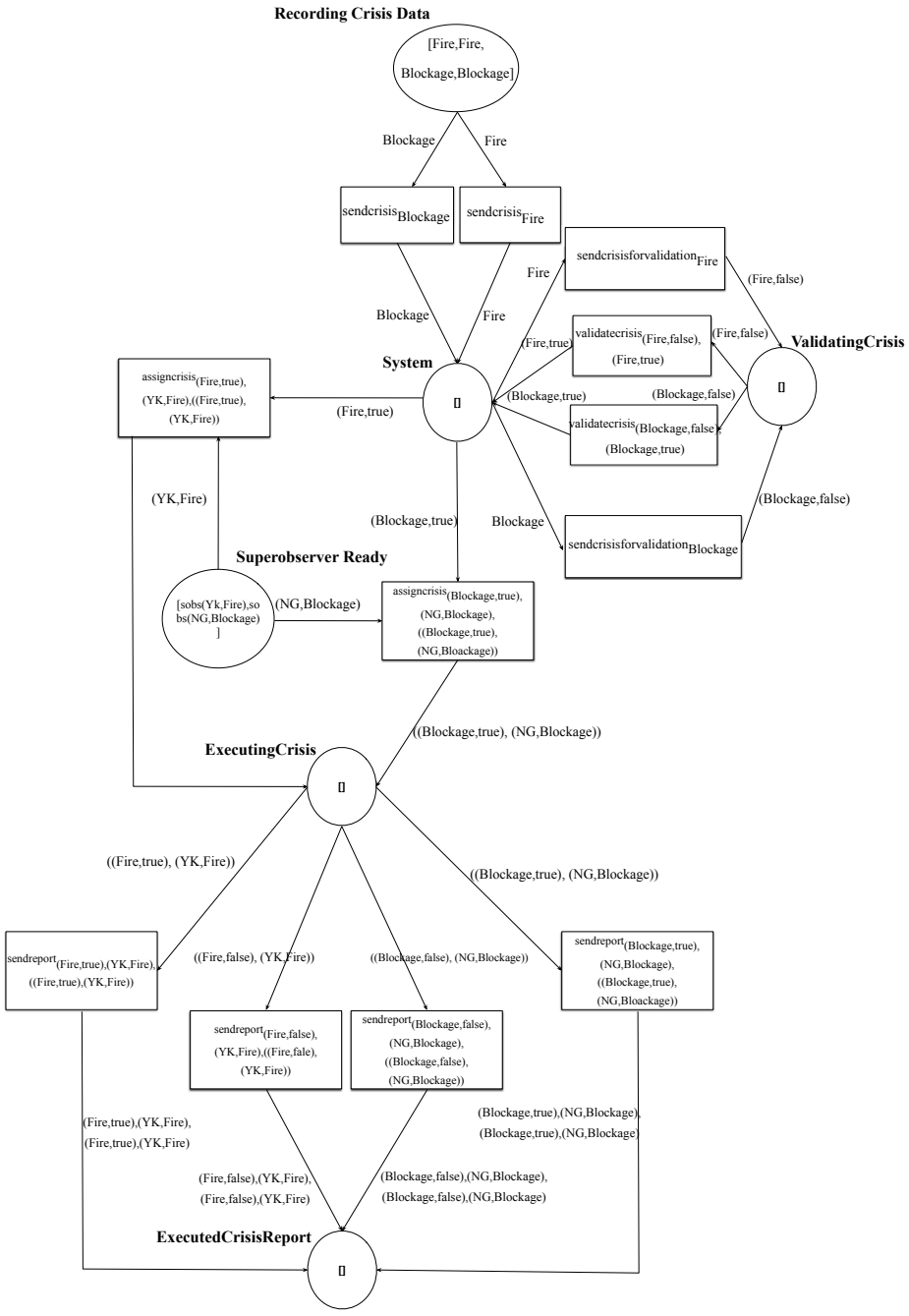


Fig. 9. The unfolded car crash APN model

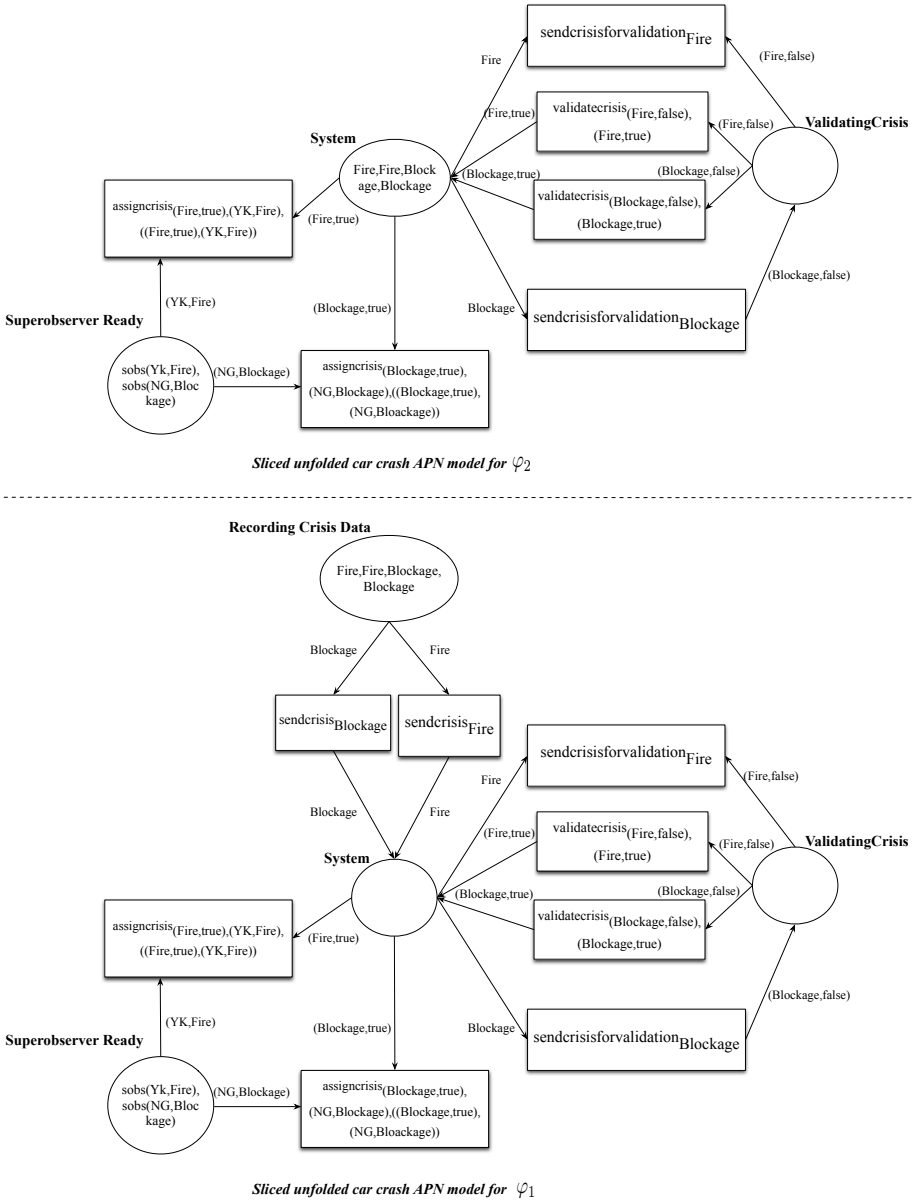


Fig. 10. Sliced unfolded car crash APN model (by applying *Abstract slicing*)

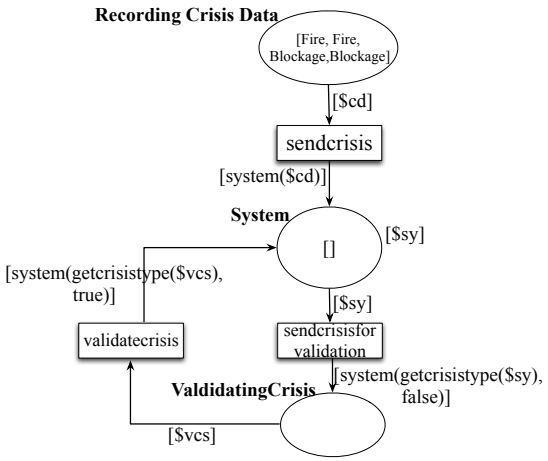


Fig. 11. Sliced car crash APN model (by applying *concerned slicing*)

6 Evaluation

In this section, we evaluate our abstract slicing algorithm and compare with existing slicing construction for APNs (Note: We do not include concerned slicing algorithm in the evaluation. As discussed in section 3, concerned slicing algorithm is designed to improve the testing of APN for the first time. We only include slicing algorithm that are designed to improve the model checking). We measure the effect of slicing in terms of savings of the reachable state space, as the size of the state space usually has a strong impact on time and space needed for model checking.

To show that state space could be reduced for practically relevant properties. We took some specific examples of temporal properties from the different case studies. Instead of presenting properties for which our method the best one, it is interesting to see where it gives an average or worst case results. Let us specify the temporal properties that we are interested to verify on the given APN model. (Note: we refer the interested reader to [8] for APN models of case studies used in the evaluation).

For the *Daily Routine of two Employees and Boss APN model*, for example, we are interested to verify that: *"Boss has always meeting"*. Formally, we can specify the property:

$\varphi_1 = \mathbf{AG}(NM \neq \emptyset)$, *where* "NM" represents a place not meeting.

For *Simple Protocol*, for example, we are interested to verify that: *"All the packets are transmitted eventually"*. Formally, we can specify the property:

$\varphi_2 = \mathbf{AF}(|PackTorec| = |PackTosend|)$, *where* "PackTosend and PackTorec" represents places.

And for a *Complaint Handling APN model*, we are interested to verify: *"All the registered complaints are collected eventually"*. Formally, we can specify the property:

$\varphi_3 = \mathbf{AG}(RecComp \Rightarrow \mathbf{AF}CompReg)$, *where* "RecComp" (resp. CompReg) means "place RecComp (resp. CompReg) is not empty".

For an *Insurance claim APN model* an interesting property could be to verify that: *"Every accepted claim is settled"*. Formally, we can specify the property:

$\varphi_4 = \mathbf{AG}(AC \Rightarrow \mathbf{AF}CS)$, *where* "AC" (resp. CS) means "place AC (resp. CS) is not empty".

For a *Customer support production system* an interesting property could be to verify that: *"Number of requests are always less than 10 "*. Formally, we can specify the property:

$\varphi_5 = \mathbf{AG}(|Requests| < 10)$.

For a *Producer Consumer APN model* an interesting property could be to verify that: *"Buffer place is never empty"*. Formally, we can specify the property:

$\varphi_6 = \mathbf{AG}(|Buffer| > 0)$.

Table 2. Results with different properties concerning to APN models

System	*Property*	*Tot.States*	*APNslicing*	*AbstractSlicing*	*Reduction*
Daily Routine of 2 Employees & Boss	φ_1	*80*	*5*	*3*	*96.25%*
Simple Protocol	φ_2	*21*	*21*	*9*	*57.143%*
Complaint Handling	φ_3	*2200*	*679*	*112*	*94.91%*
A Customer support Production system	φ_4	*471*	*171*	*91*	*80.68%*
Insurance Claim	φ_5	*889*	*121*	*49*	*94.48%*
Producer Consumer	φ_6	*372*	*372*	*372*	*0.0%*

Let us study the results summarized in the table shown in Table. 2, the first column represents the system under observation whereas the second column refers to the property that we are interested to verify. In the third column, total number of states is given based on the initial markings of places. In the fourth column, number of states are given that are required to verify the given property by applying *APNslicing*. In the fourth column, number of states that are required to verify the given property by applying *abstract slicing*. The last column represents the number of states that are reduced (in percentage) after applying Abstract slicing algorithm.

We can draw the following conclusions from the evaluation results:

- *Abstract slicing* often reduces the slice size as compared to *APNslicing* slice size. This is due to the inclusion of *neutral transition* together with *reading transitions*. As a result number of states are reduced to verify the given property, which is an improvement towards model checking. We can observe Table. 2, a part for property φ_2, there is always an improvement in the reduction of states. It is important to note that at worst the slice size obtained after applying *abstract slicing* is equal to the slice size obtained by applying *APNslicing*.

- Reduction can vary with respect to the net structure and markings of the places (this is true for both *abstract slicing* and *APNslicing*). The slicing refers to the part of a net that concerns to the property, remaining part may have more places and transitions that increase the overall number of states. If slicing removes parts of the net that expose highly concurrent behavior, the savings may be huge and if the slicing removes dead parts of the net, in which transitions are never enabled then there is no effect on the state space.

- It has been empirically proved that in general slicing produces best results for work-flow nets in [10, 16]. Our experiments also prove that for work-flow nets abstract slicing produces better results.

- *Abstract slicing algorithm* is a linear time complex.

7 Conclusion and Future Work

In this work, we have presented two slicing algorithms (i.e., *Abstract slicing* and *Concerned slicing*) to improve the verification of systems modeled in the Algebraic Petri nets. The *Abstract slicing* algorithm has been designed to improve the model checking whereas the *Concerned slicing* has been designed to improve the testing of APNs. Both the algorithms are linear time complex and significantly improves the model checking and testing of APNs.

As a future work, we are targeting to define more refined slicing constructions in the context of APNs and to implement a tool named SLAPn (i.e., slicing algebraic Petri nets). The objective of SLAPn is to show the practical usability of slicing by implementing the proposed slicing algorithms. The initial strategy to implement SLAPn is to extend the AlPiNA (Algebraic Petri net analyzer) a symbolic model checker. As discussed in the section 3, we are using the same unfolding approach as AlPiNA. Certainly, this will help to reduce the implementation effort.

8 Acknowledgement

This work has been supported by the National Research Fund, Luxembourg, Project RESIsTANT, ref.PHD-MARP-10.

References

1. D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. Alpina: A symbolic model checker. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 2010.
2. J. R. Burch, E. Clarke, K. L. McMillan, D. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 428–439, 1990.
3. J. Chang and D. J. Richardson. Static and dynamic specification slicing. In *In Proceedings of the Fourth Irvine Software Symposium*, 1994.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.
5. K. Jensen. Coloured petri nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer Berlin Heidelberg, 1987.
6. Y. I. Khan. A formal approach for engineering resilient car crash management system. Technical Report TR-LASSY-12-05, University of Luxembourg, 2012.
7. Y. I. Khan. Optimizing verification of structurally evolving algebraic petri nets. In V. K. A. Gorbenko, A. Romanovsky, editor, *Software Engineering for Resilient Systems*, volume 8166 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.
8. Y. I. Khan. Optmizing algebraic petri net model checking by slicing. Technical Report TR-LASSY-13-02, University of Luxembourg, 2013.
9. Y. I. Khan. Slicing high-level petri nets. Technical Report TR-LASSY-14-03, University of Luxembourg, 2014.
10. Y. I. Khan and M. Risoldi. Optimizing algebraic petri net model checking by slicing. *International Workshop on Modeling and Business Environments (ModBE'13, associated with Petri Nets'13)*, 2013.
11. L. Lamport. What good is temporal logic. *Information processing*, 83:657–668, 1983.
12. W. J. Lee, H. N. Kim, S. D. Cha, and Y. R. Kwon. A slicing-based approach to enhance petri net reachability analysis. *Journal of Research Practices and Information Technology*, 32:131–143, 2000.
13. M. Llorens, J. Oliver, J. Silva, S. Tamarit, and G. Vidal. Dynamic slicing techniques for petri nets. *Electron. Notes Theor. Comput. Sci.*, 223:153–165, Dec. 2008.
14. A. Rakow. Slicing petri nets with an application to workflow verification. In *Proceedings of the 34th conference on Current trends in theory and practice of computer science*, SOFSEM'08, pages 436–447, Berlin, Heidelberg, 2008. Springer-Verlag.
15. A. Rakow. *Slicing and Reduction Techniques for Model Checking Petri Nets*. PhD thesis, University of Oldenburg, 2011.
16. A. Rakow. Safety slicing petri nets. In S. Haddad and L. Pomello, editors, *Application and Theory of Petri Nets*, volume 7347 of *Lecture Notes in Computer Science*, pages 268–287. Springer Berlin Heidelberg, 2012.
17. W. Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.
18. K. Schmidt. T–invariants of algebraic petri nets. *Informatik– Bericht*, 1994.

19. F. Tip. A survey of program slicing techniques. *JOURNAL OF PROGRAMMING LANGUAGES*, 3:121–189, 1995.
20. A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.
21. Y. Wangyang, Y. Chungang, D. Zhijun, and F. Xianwen. Extended and improved slicing technologies for petri nets. *High Technology Letters*, 19(1), 2013.
22. M. Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
23. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

Performance Analysis of M/G/1 Retrial Queue with Finite Source Population Using Markov Regenerative Stochastic Petri Nets

Lyes Ikhlef[1], Ouiza Lekadir[2] and Djamil Aïssani[3]

Research Unit LaMOS (Laboratories of Modelization and Optimization of Systems)
Bejaia University.
[1]ikhlefilyes@gmail.com
[2]ouizalekadir@gmail.com
[3]lamos_bejaia@hotmail.com

Abstract. This paper aims to present an approach for modeling and analyzing an $M/G/1//2$ retrial queue, using the $MRSPN$ (Markov Regenerative Stochastic Petri Nets) tool. The consideration of the retrials and finite source population introduce analytical difficulties. The expressive power of the $MRSPN$ formalism provides us with a detailed modeling of retrial systems. In addition to this modeling, this formalism gives us a qualitative and a quantitative analysis which allow us to obtain the steady state performance indices. Indeed, some illustrative numerical results will be given by using the software package Time Net.

Keywords: Markov Regenerative Process, Markov Regenerative Stochastic Petri Nets, Retrial Systems, Steady State, Modeling, Performance Evaluation.

1 Introduction

Retrial queueing systems have been extensively studied by several authors including Kosten 1947, Wilkinson 1956, Cohen 1957. A survey work on the topic has been written by Falin and Templeton [9]. An exhaustive bibliography is given in Artalejo [5]. Recently, several papers were published for retrial systems [16,4]. These queueing models arise in many practical applications such as: computer systems, communication systems, telephone systems, etc.
The main characteristic of retrial systems is that, an incoming customer having found the server busy does not exit the system but it joins the orbit to repeat its demand after a random period (see Fig. 1).
Generally, the analytical treatment of retrial systems is difficult to obtain. Taking into account the flow of the repeated calls complicate the structure of the stochastic process corresponds to the retrial systems.
In order to evaluate the performances of these systems, a large number of different approximating algorithms and approaches were proposed [11,18,19].

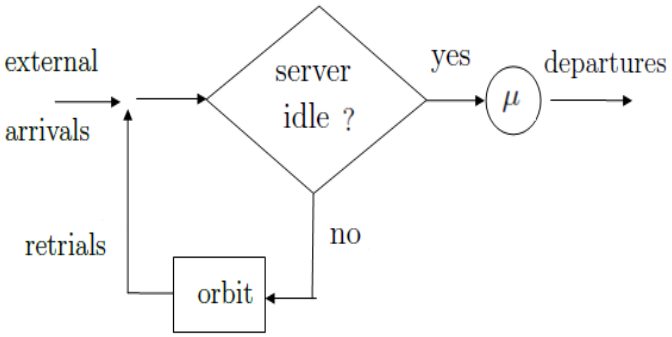


Fig. 1. Schematic diagram of retrial queue

Stochastic Petri Nets (SPN) are Petri nets in which each transition is associated with an exponentially distributed random variable that expresses the delay from the enabling condition to the firing of the transition. They are defined by Molloy [12] then extended by A. Marson et al [8] to a class of generalized stochastic Petri nets ($GSPN$) by allowing immediate transition. The underlying stochastic process of SPN or $GSPN$ is a continuous time Markov chain ($CTMC$). H. Choi [6] introduced a new class called Markov regenerative stochastic Petri nets ($MRSPN$), where a timed transition can fire according to an exponential or any other general distribution function. The underlying stochastic process of $MRSPN$ is the Markov regenerative process ($MRGP$). With the restriction at most one generally distributed timed transition is enabled in each marking. The process subordinated in two regeneration time points is a continuous time Markov chain.
The main advantages of an $MRSPN$ are:

- Modeling and evaluating the performance of complex systems comprising concurrency, synchronization, etc
- Providing automated generation and solution to discrete time Markov chains.
- Offering a qualitative and a quantitative analysis of systems.
- Existence of software tools developed within the $MRSPN$ (Time Net, SHARP, WebSPN, ...)

Most studies in the literature deal with infinite customers source retrials queues. However, in many practical situations, it is important to consider that the rate of generation of new primary calls decreases as the number of customers in the system increases. This can be done with the finite-source or quasi-random input models. The Markovian $GSPN$ is used by N. Gharbi [14,4] for analyzing an retrial queue and Oliver [17] for studying an $M/M/1//N$ queue with vacation. In 1993 H. Choi [6,7] carries out the transient and steady state analysis of $MRSPN$ (non-Markovian $GSPN$), as example $M/G/1/2/2$ is analyzed. Recently, the performance analysis of queueing systems $M/G/1//N$ with different

vacation schemes is given by K.Ramanath and P.Lakshmi[10]. The structure of the transition probability matrix P of the embedded Markov chain EMC related to $M/G/1//N$ with retrial is not an $M/G/1$-*type* [13]. Unfortunately, for such an EMC there is not a general solution and the matrix analytic method (MAM) can not be applied for analyzing these processes. Our goal in this work, is to exploit the features of $MRSPN$ for modeling and performance analysis of retrial queue $M/G/1$ with finite source population.
The remainder of this paper is organized as follows. In section 2, we introduce the analysis technique proposed for $MRSPN$. In section 3, we describe the $MRSPN$ associated to the system $M/G/1//N$ with retrial. In section 4 and 5 some performance measures are provided. Finally, the section 6 concludes the paper.

2 Steady State Analysis of MRSPN

Different approaches and numerical techniques have been explored in the literature for dealing with non-Markovian $GSPN$, we quote:

- The approach of approximating the general distribution by phase type expansion [1]
- The approach based on Markov regenerative theory [6]
- The approach based on supplementary variable [3]

The analysis of MRSPN is based on the observation that the underlying stochastic process $\{M(t), t \geq 0\}$ enjoys the absence of memory at certain instants of time $(t_0, t_1, t_2, ...)$. This instants referred as regeneration points. An embedded Markov chain (EMC) $\{Y_n, n \geq 0\}$ can be defined at the regeneration points. An analytical procedure for the derivation of expression for the steady state probability is proved in [6]. The conditionals probability necessary for the analysis of a $MRSPN$ are:

- The matrix $K(t)$ is called global kernel given by $K_{ij}(t) = P\{Y_1 = j, t_1 \leq t / Y_0 = i, i, j \in \Omega\}$. It describes the process behavior immediately after the next Markov regenerative point. (Ω is the set of state of tangible markings).
- The matrix $E(t)$ is called the local kernel given by $E_{ij}(t) = P\{M_t = j, t_1 > t / Y_0 = i\}$. It is for the behavior between two Markov regeneration points.

When the EMC is finite and irreducible its steady state probability vector v is obtained by the solution of the linear system equation: $vP = v$ and $v1 = 1$. Where the one-step transition probability matrix P of the EMC is derived from the global kernel ($P = \lim_{t \to +\infty} K(t)$). The steady state distribution $\pi = (\pi_1, \pi_2, ...)$ of the $MRGP$ can be obtained by: $\pi = \dfrac{\sum_{k \in \Omega} v_k \alpha_{kj}}{\sum_{k \in \Omega} v_k \sum_{l \in \Omega} \alpha_{kl}}$ where $\alpha_{ij} = \int_0^\infty E_{ij}(t)dt$.

3 M/G/1//N with Retrials

We consider a single server retrial queue with finite population of size N. A customer arrives from the source according to a poisson process with parameter "λ". When the server is idle the customer immediately occupies the service. The service time distribution follows a general law with probability distribution function $F^g(x)$. If the server is busy, the customer joins the orbit to repeat its demand for service after an exponential time with parameter θ until it finds a free server. Fig. 2 shows the $MRSPN$ model describing the $M/G/1//N$ queueing system with retrial. In Fig. 2 thick black bar represents GEN transition, thick white bars represent EXP transitions, thin bars represent immediate transitions.

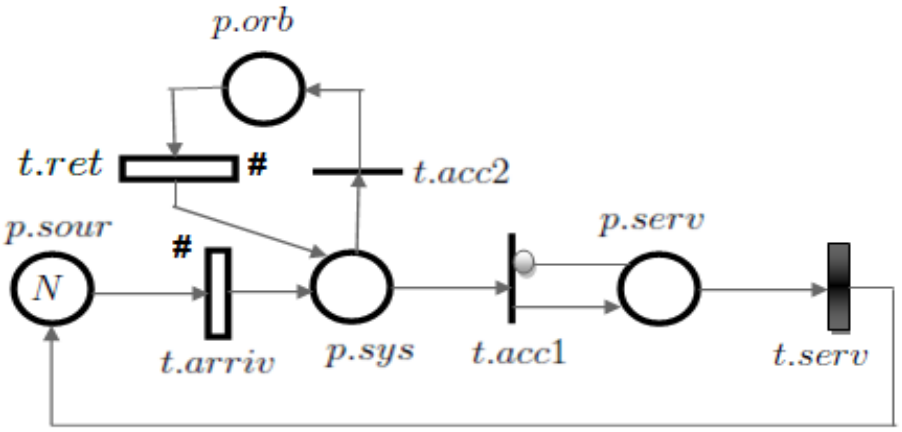


Fig. 2. MRSPN for the M/G/1//N retrial queueing system.

The initial marking of the $MRSPN$ is :

$$M_1(M(p.sour), M(p.sys), M(p.serv), M(p.orb)) = M_1(N, 0, 0, 0)$$

- The firing of timed transition $t.arriv$ indicates the arrival of a customer in source thus the place $p.sys$ receives a token. The firing of $t.arriv$ is marking dependent, its firing rate is $\#(p.sour)\lambda$.
- The immediate transition $t.acc1$ is enabled when the place $p.sys$ contains at least one token and $p.serv$ contains no token (the server is free). The firing of $t.acc1$ consists to destroy a token in place $p.sys$ and builds a token in place $p.serv$ (this represents the fact that the customer has started its service and the server is moved from the free state to the busy state).

- The firing of the timed transition $t.serv$ consists to destroy a token in the place $p.serv$ and constructs a token in the place $p.sour$ (the costumer has completed its service). The server is moved from the busy state to the free state. The firing policies of $t.serv$ is the race with enabling memory.
- The immediate transition $t.acc2$ is enabled when the place $p.sys$ and $p.serv$ contain a token (the server is busy). The firing of the transition $t.acc2$ consists to destroy a token in $p.sys$ and constructs a token in place $p.orb$ (the customer joins the orbit). The immediate transition $t.acc1$ has higher priority than the immediate transition $t.acc2$.
- The firing of the timed transition $t.ret$ consists to remove a token from place $p.orb$ and constructs a token in place $p.sys$. The firing of $t.ret$ is marking dependent, thus its firing rate is $\#(p.orb)\lambda$.

4 Case of the $M/G/1//2$ retrial queueing system

In this section we consider the $M/G/1//2$ retrial queue. We obtain the reachability tree which describes all possible states of our $MRSPN$ starting from the initial marking M_1 (see Fig. 3).
From this reachability tree, by marging the vanishing markings into their successor tangible markings, we have obtained the state transition diagram of the $MRSPN$ depicted in Fig.2
In Fig.4 solid arcs indicate state transition by EXP transitions, dotted arcs indicate state transitions by GEN transitions.
The infinitesimal generator matrix of the subordinated $CTMC$ with respect to transition $t.serv$ is given by:

$$Q = \begin{pmatrix} -2\lambda & 2\lambda & 0 & 0 \\ 0 & -\lambda & 0 & \lambda \\ 0 & \theta & -(\theta+\lambda) & \lambda \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Local kernel $E(t)$:

$$E(t) = \begin{pmatrix} e^{-2\lambda t} & 0 & 0 & 0 \\ 0 & e^{-\lambda t}[1-F^g(t)] & 0 & (1-e^{-\lambda t})[1-F^g(t)] \\ 0 & 0 & e^{-(\theta+\lambda)t} & 0 \\ 0 & 0 & 1-F^g(t) & 0 \end{pmatrix}$$

Global kernel $K(t)$:

$$K(t) = \begin{pmatrix} 0 & 1-e^{-2\lambda t} & 0 & 0 \\ \int_0^t e^{-\lambda x} dF^g(x) & 0 & \int_0^t [1-e^{-\lambda x}] dF^g(x) & 0 \\ 0 & \frac{\theta}{\theta+\lambda}[1-e^{-(\theta+\lambda)t}] & 0 & \frac{\lambda}{\theta+\lambda}[1-e^{-(\theta+\lambda)t}] \\ 0 & 0 & \int_0^t dF^g(x) & 0 \end{pmatrix}$$

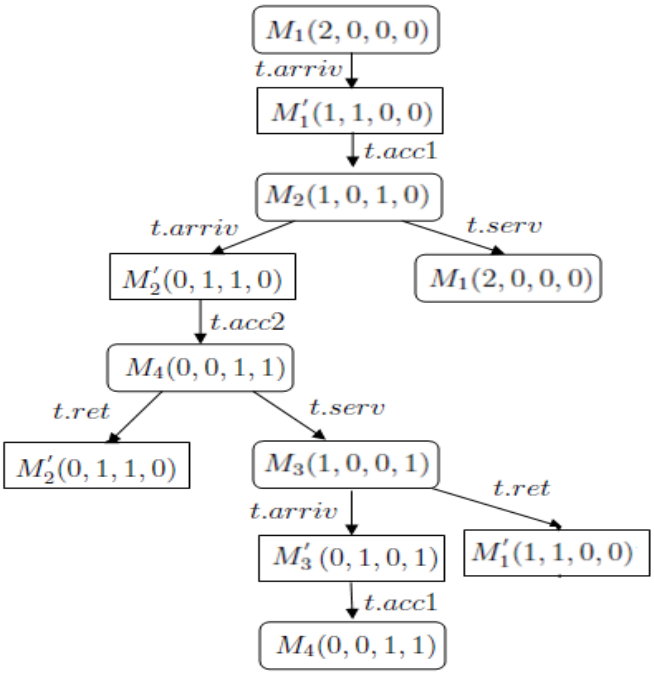


Fig. 3. Reachability tree for the $MRSPN$ of FIG. 2 $(N = 2)$.

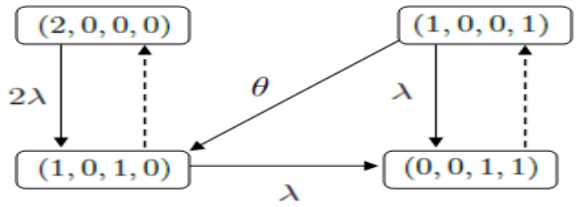


Fig. 4. Subordinated CTMC for the $MRSPN$ of FIG.2 $(N = 2)$.

Where the density function of the firing time of $t.serv$ is given by hyperexponential distribution "$H_2(\frac{1}{3}, \frac{\mu}{2}, \mu)$": $f^g(x) = \frac{1}{6}\mu e^{\frac{-1}{2}\mu x} + \frac{2}{3}\mu e^{-\mu x}$. The one-step

transition probability matrix P given by:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ \frac{1}{3}\frac{\mu(5\lambda+3\mu)}{(2\lambda+\mu)(\lambda+\mu)} & 0 & \frac{2}{3}\frac{\lambda(3\lambda+2\mu)}{(2\lambda+\mu)(\lambda+\mu)} & 0 \\ 0 & \frac{\theta}{\theta+\lambda} & 0 & \frac{\lambda}{\theta+\lambda} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The $MRSPN$ depicted in FIG. 2 $(N = 2)$, is bounded and admits M_1 like home state so it is ergodic.

We calculate the steady state probabilities by solving: $vP = v$ and $v1 = 1$:

$$v_1 = \frac{1}{2}\ \frac{\theta\mu(5\lambda+3\mu)}{6\theta\lambda^2+9\theta\lambda\mu+3\theta\mu^2+6\lambda^3+4\lambda^2\mu}\ ,$$

$$v_2 = \frac{3}{2}\ \frac{\theta(2\lambda+\mu)(\lambda+\mu)}{6\theta\lambda^2+9\theta\lambda\mu+3\theta\mu^2+6\lambda^3+4\lambda^2\mu}$$

$$v_3 = \frac{\lambda(\theta+\lambda)(3\lambda+2\mu)}{6\theta\lambda^2+9\theta\lambda\mu+3\theta\mu^2+6\lambda^3+4\lambda^2\mu}\ ,$$

$$v_4 = \frac{\lambda^2(3\lambda+2\mu)}{6\theta\lambda^2+9\theta\lambda\mu+3\theta\mu^2+6\lambda^3+4\lambda^2\mu}$$

$\alpha_{11} = \frac{1}{2\lambda}$, $\alpha_{22} = \frac{2}{3}\ \frac{3\lambda+2\mu}{(2\lambda+\mu)(\lambda+\mu)}$, $\alpha_{24} = \frac{2}{3}\ \frac{4\lambda^2+3\lambda\mu}{\mu(2\lambda+\mu)(\lambda+\mu)}$, $\alpha_{33} = \frac{1}{\alpha+\lambda}$, $\alpha_{44} = \frac{4}{3\mu}$
The steady state probabilities: $\pi = (\pi_{(2,0,0,0)}, \pi_{(1,0,1,0)}, \pi_{(1,0,0,1)}, \pi_{(0,0,1,1)})$ are given by:

$$\pi_{(2,0,0,0)} = \frac{3\theta\mu^2(5\lambda+3\mu)}{39\theta\mu^2\lambda+9\theta\mu^3+48\theta\lambda^3+72\theta\lambda^2\mu+68\lambda^3\mu+24\lambda^2\mu^2+48\lambda^4}$$

$$\pi_{(1,0,1,0)} = \frac{12\theta\lambda\mu(3\lambda+2\mu)}{39\theta\mu^2\lambda+9\theta\mu^3+48\theta\lambda^3+72\theta\lambda^2\mu+68\lambda^3\mu+24\lambda^2\mu^2+48\lambda^4}$$

$$\pi_{(1,0,0,1)} = \frac{12\lambda^2\mu(3\lambda+2\mu)}{39\theta\mu^2\lambda+9\theta\mu^3+48\theta\lambda^3+72\theta\lambda^2\mu+68\lambda^3\mu+24\lambda^2\mu^2+48\lambda^4}$$

$$\pi_{(0,0,1,1)} = \frac{4\lambda^2(12\lambda^2+8\lambda\mu+12\theta\lambda+9\theta\mu)}{39\theta\mu^2\lambda+9\theta\mu^3+48\theta\lambda^3+72\theta\lambda^2\mu+68\lambda^3\mu+24\lambda^2\mu^2+48\lambda^4}$$

Having the steady state probabilities $\pi = (\pi_{(2,0,0,0)}, \pi_{(1,0,1,0)}, \pi_{(1,0,0,1)}, \pi_{(0,0,1,1)})$ several performance characteristics of $M/G/1//N$ with retrial can be derived:

- The effective arrival rate λ_e: $\lambda_e = \lambda[1 + \pi_{(2,0,0,0)} - \pi_{(0,0,1,1)}]$
- The mean number of customers in the orbit n_{orb}: $n_{orb} = \pi_{(1,0,0,1)} + \pi_{(0,0,1,1)}$
- The mean number of customers in the system n_s: $n_s = 1 - \pi_{(2,0,0,0)} + \pi_{(0,0,1,1)}$
- The mean response time τ, from Little's law: $\tau = \frac{n_s}{\lambda_e} = \frac{1-\pi_{(2,0,0,0)}+\pi_{(0,0,1,1)}}{\lambda[1+\pi_{(2,0,0,0)}-\pi_{(0,0,1,1)}]}$

Table 1. Performance measures for the $MRSPN$ of FIG.2 ($N = 2$, $\lambda = 0,8$, $\theta = 0,2$, $\mu = 1,0$).

Steady state probabilities		Performance indices	
$\pi_{(2,0,0,0)}$	$0,0456482045$	λ_e	$0,4403095383$
$\pi_{(1,0,1,0)}$	$0,0918181028$	n_{orb}	$0,8625336928$
$\pi_{(1,0,0,1)}$	$0,3672724111$	n_s	$1,449613077$
$\pi_{(0,0,1,1)}$	$0,4952612817$	τ	$3,292259083$

5 Numerical Results

In this section we present some numerical results using the Time Net [10] (Timed Net Evaluation Tool) software package which supports a class of non-markovian $GSPN$. We illustrate the effect of the parameters on the main performance characteristics. The model proposed was validated by the exact analytical results of $M/G/1//N$ without retrial, see Table 2.

Table 2. Validation of results.

Performance indices	$M/G/1//2$ without retrial ($\lambda = 0,5$, Service $U_{[0,5;1,0]}$)	$MRSPN$ associated to $M/G/1//2$ with retrial ($\lambda = 0,5$, Service $U_{[0,5;1,0]}$, $\theta \simeq \infty$)
λ_e	$0,69488$	$0,69390$
n_s	$0,61022$	$0,61218$
τ	$0,87816$	$0,88223$

From the Table 2, when the retrial rate is very large, the performance indices corresponding the $MRSPN$ associated to $M/G/1//2$ queue with retrial are very close to those obtained by $M/G/1//2$ queue without retrial.
For $N = 25$, $\lambda = 0.1$, $\theta = 0.25$, we obtain the performance indices of our $MRSPN$. Where λ_e, θ_e: respectively represents the effective customers arrival rate and retrial rate. n_{orb}, n_s: respectively represents the average number of customers in orbit and in system. $\overline{W}$, $\overline{T}$: respectively represents the mean response time in system and mean waiting time in the orbit, which are summarized in the Table 3.
In Figure 5, 6 and 7 we give some graphical results in order to illustrate the way in which the model is affected from the variation in the retrial rate and the size of the source.
In FIG.5, we observe that the mean number of customers in the orbit decreases as the retrial rate increases.

Table 3. Some performance measures for the $MRSPN$ of FIG.2 ($N = 25,\ \lambda = 0.1,\ \theta = 0.25$).

Performance indices	Service Det$(0, 8)$	Service $U_{[0,5;1,0]}$
λ_e	0,9865245	1,0340604
θ_e	3,5863839	3,4709629
n_{orb}	14,3455359	13,8838515
n_s	15,1347555	14,6593961
$\overline{W}$	15,3414897	14,1765376
$\overline{T}$	14,5414897	13,4265382

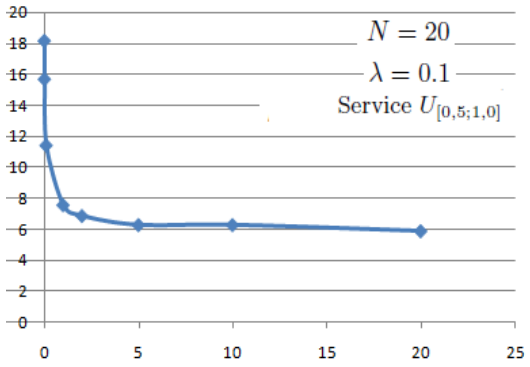


Fig. 5. Effect of retrial rate on mean number of customers in the orbit.

In FIG.6, we observe that mean response time of the system decreases as the retrial rate increases.
In FIG.7, we observe that mean response time of the system increases as the size of the source increases.

6 Conclusion

In this work a single server retrial queue $M/G/1$ with finite source population is considered. We focused on how to exploit the features of $MRSPN$ to cope with the complexity of such system. The $MRSPN$ approach allowed us to compute efficiently exact performance measures. We have illustrated the functionality of this approach with the example $M/G/1//2$ with retrial. Some performance

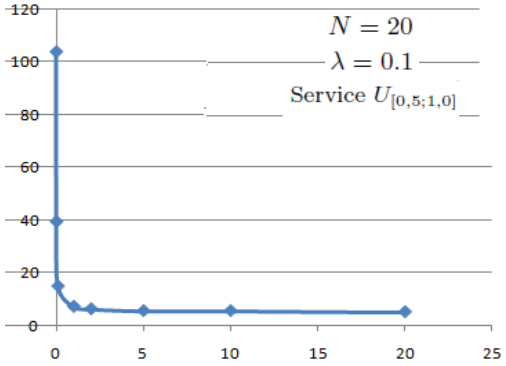


Fig. 6. Effect of retrial rate on mean response time in the system

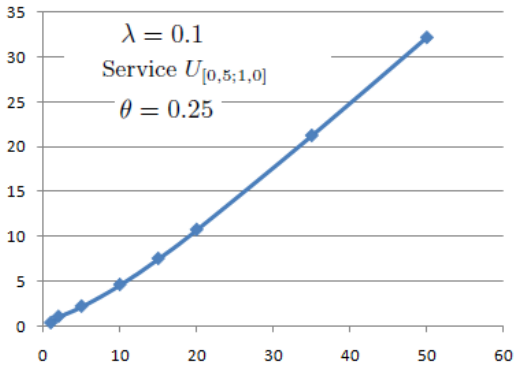


Fig. 7. Effect of size of source on mean response time in the system

measures are carried out by the help of the software package Time Net. Our future work aims at make generalization for any N (size of population) in order to propose an algorithm for computing the transition matrix and performance measures without generating the reachability graph. Also it may be interesting to provide a more detailed study by including to the same model: vacation, breakdown, etc.

References

1. A. Cumani. Esp: A package for the evaluation of stochastic Petri nets with phase-type distributed transition times. In proceedings International Workshop Timed Petri nets, pages 144-151, Torino (Italy), (1985). IEEE Computer Society Press no. 674.
2. C.Ciardo, R.German, and C.Lindeman: A characterization of the stochastic process underlying a stochastic petri nets. IEEE, Trans, 20,506-515(1994).
3. D.R. Cox: The analysis of non-markovian stochastic processes by the inclusion of supplementary variables. Proceedings of the Cambridge Philosophical Society, 51: 433-440, (1955).
4. F. Zhang and Jinting Wang: Performance analysis of the retrial queues with finite number of sources and service interruptions. Journal of the Korean Statistical Society 42 (2013) 117-131.
5. J. R. Artalejo: Accessible bibliography on retrial queues. Mathematical and computer Modelling, 30: 1-6,(1999).
6. H. Choi, V.G.Kulkarni and K. Trivedi: Markov regenerative stochastic Petri nets. Performance Evaluation, 20: 337-357, (1994).
7. H.Choi, V.G.Kulkarni and K.S.Trivedi: Markov Regenerative Stochastic Petri Nets. IEEE trans. comput., 31(9): 913-917,(1982).
8. G. Chiola, M.A. Marsan, G.Balbo, and G.Conte: Generalized stochastic Petri nets, a definition at the net level and its implications. IEEE trans. on software Eng.,19(2): 89-107,(1993).
9. G. I.Falin and J.G.C. Templeton: Retrial queues. Chapman and Hall, London, 1997.
10. K. Ramanath and P. Lakshmi: Modelling $M/G/1$ queueing systems with server vacations using stochastic Petri nets, 22(2), pp.131-154(2006).
11. L. Berjdoudj and D. Aissani: Strong stability in retrial queues. Theor. Probability and Math. Statis.,68,11-17,(2003)
12. M.K.Molloy: Performance analysis using stochastic Petri nets. IEEE trans. comput., 31(9):913-917,(1982)
13. M. Neuts: Structured Stochastic Matrices of M/G/1 Type and Their Applications, Marcel Dekker, Inc., New York and Basel, 1989.
14. N.Gharbi and M.Ioualalen: Performance analysis of retrial queueing systems using generalized stochastic petri nets. USTHB-Alger, Algérie.
15. N. Gharbi and C. Dutheillet: An algorithmic approach for analysis of finite-source retrial systems with unreliable servers. Computers and Mathematics with Applications 62 (2011) 2535-2546.
16. O.Dudina , C. Kim and S.Dudin: Retrial queuing system with Markovian arrival flow and phase-type service time distribution, Computers Industrial Engineering 66 (2013) 360-373.
17. Oliver C. and Kishor S: Stochastic Petri net analysis of finite population. J.C. Baltzer A.G. Scientific Publishing Company, USA (1990).
18. S.N. Stepanov: Numerical methods of calculation for systems with repeated calls, Nauka Moscow(1983).
19. T.Yang, M.J.M.Poser, J.G.C.Templeton and H.Li: An approximation for the M/G/1 retrial queue with general retrials times, European Journal of Operational Research, 76, 552-562,(1994).
20. TimeNET 4.0: A Software Tool for the Performability Evaluation with Stochastic and Colored Petri Nets. User Manual. Armin Zimmermann and Michael Knoke Technische Universität Berlin Real Time Systems and Robotics Group Faculty of EE and CS Technical Report 2007/13 ISSN: 1436/9915, August (2007).

Petri Nets Based Approach for Modular Verification of SysML Requirements on Activity Diagrams

Messaoud Rahim[1], Malika Boukala-Ioualalen[2], and Ahmed Hammad[1]

[1] FEMTO-ST Institute, UMR CNRS 6174, Besançon, France.
{Lastname.firstname}@femto-st.fr
[2] MOVEP, Computer Science department, USTHB, Algiers, Algeria.
mioualalen@usthb.dz

Abstract. The validation of SysML specifications needs a complete process for extracting, formalizing and verifying SysML requirements. Within an overall approach which considers an automatic verification of SysML designs by translating both requirement and behavioral diagrams, this paper proposes a modular verification of SysML functional requirements on activity diagrams. The contribution of this paper is the proposition of a methodology guided by the relationships between requirements and SysML activities for verifying complex systems with many components. We propose a model-to-model transformation to automatically derive from SysML activities a modular Petri net, then SysML requirements are formalized and verified using the derived Petri net modules. A case study is presented to demonstrate the effectiveness of the proposed approach.

Keywords: SysML, Activity Diagram, SysML Requirements, Requirements Formalization, Modular Verification, Petri nets

1 Introduction

Model-based systems engineering is becoming a promising solution to design complex systems. SysML (System Modeling Language) [1] is a standard modeling language which has been proposed to specify systems that include heterogeneous components. It covers four perspectives on system modeling : structure, behavior, requirement, and parametric diagrams. Particularly, the SysML requirement diagram is used for better organizing requirements at different levels of abstraction, allowing their representation as model elements, and showing explicitly the various kinds of relationships between requirements and design elements [2]. However, one of the main challenge in system design is to ensure that a model meets its requirements. To provide a validation of SysML specifications, existing approaches [3–5] propose to translate SysML behavioral models into formal specification languages, then they verify temporal properties by using model-checking techniques. These approaches ignore systems composition and do not relate system requirements to design elements. The activity diagram is one of

SysML models used to specify the system behavior and where requirements can be verified. Based on using the call behavior action concept, a modular design of complex systems can be obtained by structuring its behaviour in many activities. This provides a compositional specification and enables modular analysis of the specified systems [6]. Requirements can be expressed as properties to verify by an activity diagram during its execution. Unfortunately, the need for formal specifications of properties expressed using logics or automata is a major obstacle for the adoption of formal verification techniques by SysML practitioners [7]. The contribution of this paper is the proposition of a methodology which provides a modular verification of functional SysML requirements captured by activity diagrams. It consists on: *(1)* performing a compositional translation from SysML activity diagrams into modular Petri nets where modular PNML [8] is used as target language. *(2)* proposing a new language (AcTRL : Activity Temporal Requirement Language) to express functional requirements related to activities and showing how AcTRL expressions can be automatically translated into properties expressed as temporal logic formulas. Finally, *(3)* presenting a modular verification algorithm. The compositional translation enables the modular verification by considering the decomposition of activity diagrams into sub-activities and the use of AcTRL avoids the specification of SysML requirements directly as properties of the formal semantic model (Petri nets in our case).
This paper is organized as follows. Section 2 surveys related works. Section 3 presents related concepts. Section 4, introduces our overall methodology. In Section 5, we present a compositional translation from SysML activities to modular Petri nets. In Section 6, we define AcTRL and its grammar. An algorithm for modular verification of requirements will be presented in section 7. In Section 8, we illustrate our approach by a case study. Finally, in Section 9, we conclude and we outline future works.

2 Related Work

Ensuring the correctness of complex and critical systems needs automated approaches for verifying and validating their designs. In [3], authors propose to derive for each SysML behavioral diagram a formal semantic model reflecting its characteristics. In this work, requirements was expressed as temporal properties on the formal semantic model which makes the verification process difficult for SysML practitioners. Linhares et al [9] designed a process for verifying SysML specification by considering block, activity and requirement diagrams and where requirements must be expressed using Linear Temporal Logic(LTL). In [4], authors propose TEPE, a graphical temporal property expression language based on SysML parametric diagram to express system requirements. This work is restricted to state machine diagrams. Regarding activity diagram, a symbolic model checking was proposed in [10], where activity diagram is translated into SMV and the NuSMV model checker was used to verify LTL properties. Data flows were not considered in this work. In [11], the authors present a technique to map the SysML activities to time Petri net for validating the requirements of

real-time systems with energy constraints. This work considers non functional requirements. The work presented in [12] proposes a model-driven engineering approach for simulating SysML activity diagram using Petri net and VHDL-AMS. This work focuses on defining rules to translate SysML diagram elements to Petri net specification but it does not consider compositional structure of activity diagrams. To our knowledge, the present work is the first that considers a modular verification of SysML requirements by taking into account their relations to activities.

3 Preliminaries

In this section, we present SysML requirement and activity diagrams as described in the OMG standard [1]. In addition, we describe modular PNML language [8] for representing modular Petri nets.

3.1 SysML requirement diagram

Requirements in SysML are defined in an informal way with an identifier and a text. Requirement diagrams are used for specifying requirements and to depict their hierarchy and the exiting relationships between them and other SysML models. As depicted in Figure 1, the <<Verify>> relationship is a dependency between a requirement and a test case that can determine whether a system fulfills the requirement [1]. The <<deriveReqt>> relationship is a dependency between two requirements. It is used to derive a requirement from another. Other relationships exists, we refer to [1] for a detailed description.

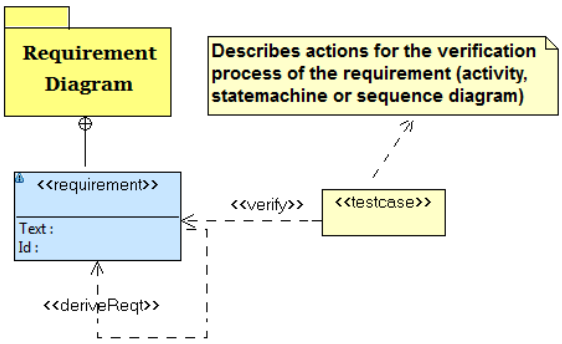


Fig. 1: <<Verify>> and <<deriveReqt>> relationships in requirement diagram

In this paper, we exploit the <<Verify>> relationship, to determine the activities which are used to verify requirements. We derive from functional requirements, more formal requirements described as properties about activity diagram

elements. For tractability purpose, the <<deriveReqt>> relationship will be exploited to relate between natural text and the more formal requirements.

3.2 SysML activity diagram

In this section, we introduce only a brief description of the SysML activity diagram and its elements, more details can be found in [1]. In SysML, an activity is a formalism for describing behaviour that specifies the transformation of inputs to outputs through a controlled sequence of actions. The basic constructs of an activity are actions and control nodes as illustrated in Figure 2. Actions are the building blocks of activities, each action can accept inputs and produces outputs, called tokens. These tokens can correspond to anything that flows such as information or physical item (e.g., water, signal). Control nodes include fork,

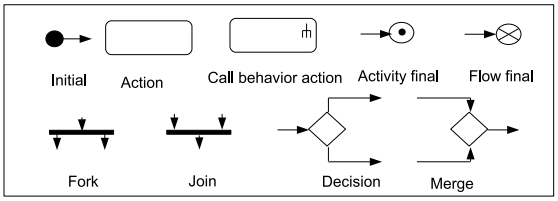


Fig. 2: Activity diagram basic constructs

join, decision, merge, initial, activity final, and flow final.
A specific type of action is the call behavior action. A call behavior action permits to invoke an activity when it starts, and passes the tokens from its input pins to the input parameter nodes of the invoked activity. A call behavior action terminates when its invoked activity reaches an activity final, or when the action receives a control disable. The tokens on the output parameter nodes of the activity are placed on the output pins of the action and a control token is placed on each of the control outputs of the action.

3.3 PNML for Modular Peri nets

The Petri Net Markup Language (PNML) [13] is an interchange format for all kinds of Petri nets. It is currently standardised by ISO/IEC JTC1/SC7 WG 19 as Part 2 of ISO/IEC 15909 [13]. The main features of PNML are its readability which is guaranteed by its XML syntax, its universality to support different Petri net type and its mutuality guaranteed by the use of common principals and common notations [8]. To address real world systems which are too large to be drawn on single page, the PNML provide a net type independent mechanism for structuring large Petri nets. Two mechanisms are proposed, pages and modules. The concept of pages is used with the concept of references to structure the nets

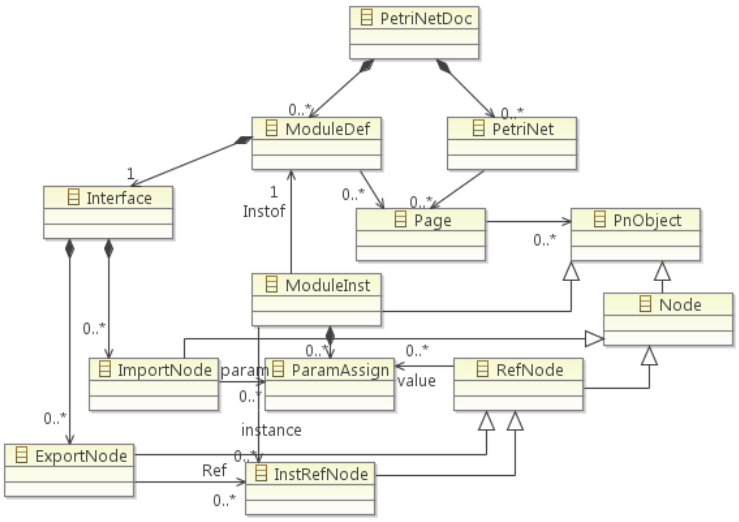


Fig. 3: Extract of the defined meta-model for the modular PNML

on several pages. It is used only for more convivial visual structure of the nets. The concept of modules is supported by modular PNML. Modular PNML as presented in [8] is an extension of the PNML to describe modular Petri nets. It is proposed for defining Petri net modules and for constructing nets from different instances of such modules. A Petri net module is defined as a Petri net with an interface composed by imported and exported nodes. For the transformation to perform in this work, we have extended the PNML (P/T Type) meta-model to support a modular structure of Petri nets. The Figure 3 presents an extract of Modular PNML meta-model which we have inspired from [8]. The presented Modular PNML meta-model extends the core PNML meta-model to allow the definition of modules (ModuleDef) and their instantiation (ModuleInst). Each module includes an interface which contains import and export nodes. A module instance assigns import nodes to reference nodes (ParmAssign) and can contains a reference nodes from other instances(InstRefNode). More explanations can be found in [8].

4 Overall methodology

In this section, we describe our methodology for verifying SysML requirements on activity diagrams. First, the SysML designer creates requirement and activity diagrams to specify the system. Then, he drives from functional requirements, which are related to the activity diagram by a <<Verify>> relationship, temporal requirements described as properties about activity diagram elements. After that, an automatic translation process is used to transform this SysML specification into a formal specification. The SysML activity diagram is translated into

modular Petri net and temporal requirements into formal properties described as temporal logic formulas. A Petri Net tool will be used to check if these requirements are verified in the derived Petri net modules. The verification will be guided by the existing <<Verify>> relationships between requirements and activities. Finally, a feed back is given to the SysML designer to correct his specification. As our approach is modular, in the case of the non satisfaction of a requirement, the generated feed back can give a more accurate indication about the sub activity and the actions which are related to the design error. The Figure 4 summarizes the steps of our methodology.

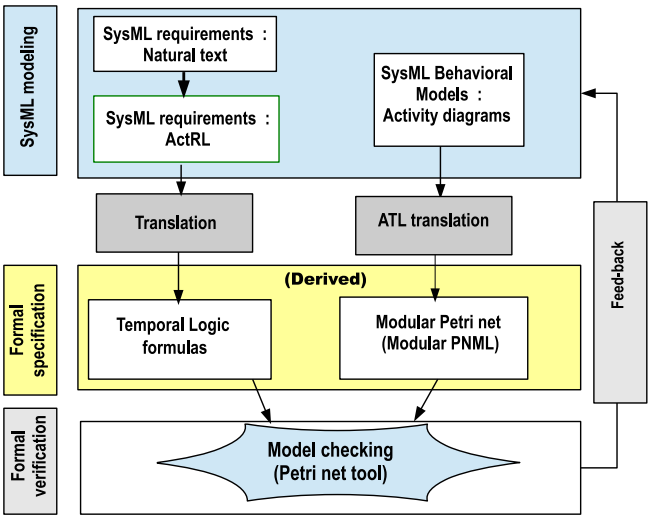


Fig. 4: Overall methodology

5 From activity diagrams to modular Petri nets

In this section, we describe our translation of SysML activity diagrams into modular PNML. We propose to use the activity diagram meta-model defined in the TOPCASED tool [14] as source meta-model and the modular PNML meta-model presented in Section 3.3 as target meta-model. Based on EMF(Eclipse Modeling Framework) with Ecore meta meta-model and ATL language [15], the transformations are defined as semantic and structural mappings based on the respective meta-models. The target model of the transformation is a modular Petri net described in modular PNML which preserve the structure and the semantic of the source SysML activity diagram model.

5.1 Mapping the structure

The transformation must preserve the composite structure of the SysML activity diagram in the target Petri net model. When the SysML activity diagram includes a call behavior actions to define composite activities, the derived Petri net will be composed of modules. Each sub-activity is translated into Petri net module.

The Figure 5 illustrates the derivation of the Petri net modular structure according to the activity decomposition.

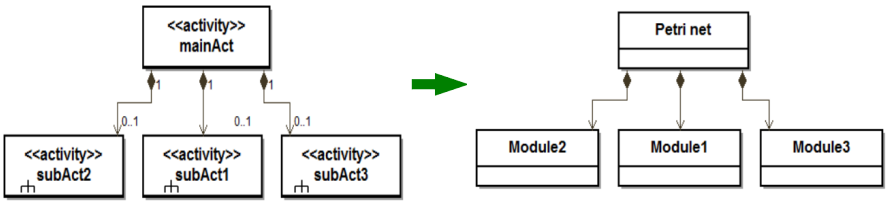


Fig. 5: The structure of the derived Petri net

We create a PNML document and Petri net only for the main activity diagram. For a sub-activity, we create a Petri net module. The ATL rule used to select the main activity is given below:

```
rule mainAct2mpnml {
from
d: ADUML!Activity(not (d.owner.oclIsTypeOf(ADUML!Activity)))
to
 p : MPNML!PetriNetDoc (nets <- f, modulesDef <-
 PNML!ModuleDef.allInstances()),
 s: MPNML!Name(text <- d.name),
 f: MPNML!PetriNet(name <- s)
 .....
}
```

5.2 Translating SysML activity constructs

The translation of basic activity constructs is inspired from the work presented in [5,6]. So, as we are interested to preserve the composite structure of the SysML activity diagram, we have adapted this translations mainly for input and output pins. The Figure 6 presents the used translation rules.

Translating call behavior actions Three principal steps are considered when translating call behavior actions :

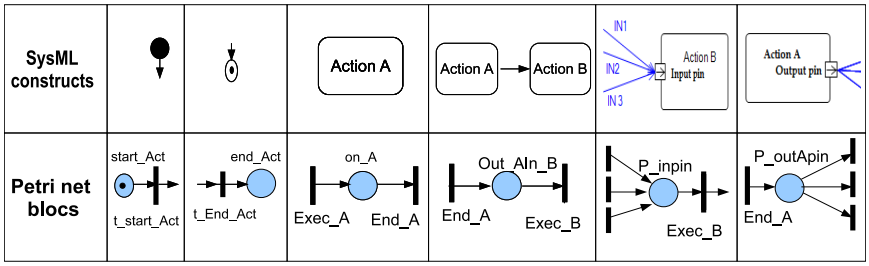


Fig. 6: Translation of basic SysML AD constructs

– Step 1: pass input flows from call behavior action to the called activity.
– Step 2: execute the called activity.
– Step 3: pass output flows from the called activity to the call behaviour action.

The mapping of a call behavior action A that invokes an activity Act with one input and one output control flow, n input pins and m output pins is as presented in Figure 7. The PNML code related to this translation includes definitions of

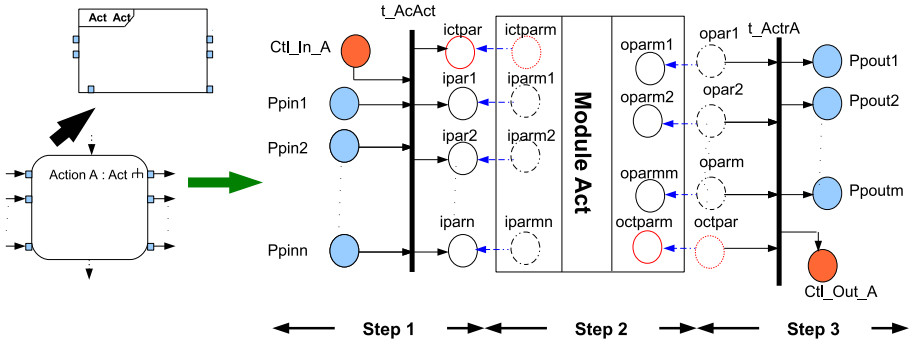


Fig. 7: Translation of call behavior action

transitions, places and arcs related to step 1 and step 3. It must also include an instance of the Petri net module defined for the activity Act (see the next section). This instance is defined like in the following listing :

```
<instance id="A_Act" ref=URI#Act>
<Paramassign parameter="ictparm" ref="ictpar"/>
<Paramassign parameter="iparm1" ref="ipar1"/>
......
<Paramassign parameter="iparmn" ref="iparn"/>
</instance>
```

We signal that the nodes $octpar, opar1, opar2, ...oparn$ (step 3) are instance reference places. They are defined in modular PNML like :

```
<InstRefPlace id="octpar" instance="A_Act" ref="octparm"/>
<InstRefPlace id="opar1"  instance="A_Act" ref="oparm1"/>
.....
<InstRefPlace id="oparm" instance="A_Act" ref="oparmm"/>
```

5.3 Mapping Sub-Activities

As described in Section 5.1, sub-activities are translated into PNML modules. Activity parameters are for accepting inputs to an activity and providing outputs from it. An activity with input and output parameters is translated into PNML module as illustrated in Figure 8. Input activity parameters are translated into

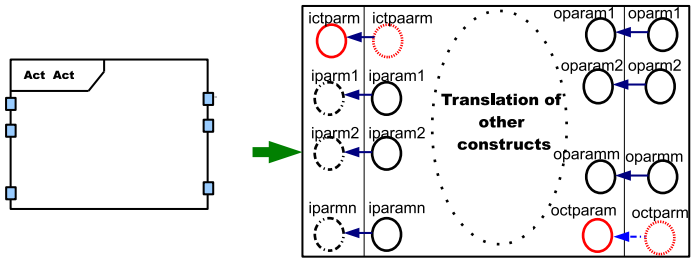


Fig. 8: Translation of Sub-activities

reference places. Output activity parameters are translated into places. The interface of the PNML module is composed of import places and export places. Import places are refereed by the reference places representing input activity parameters. Export places refer to places representing output activity parameters.

6 AcTRL: Activity Temporal Requirement language

As presented in Section 3.1, SysML requirements are described using an Id and natural text. To address this limitation, we propose AcTRL(Activity temporal requirement language) which can be used by SysML designers to express requirements to verify on activity diagrams. First, we define a high level representation of the activity diagram operational semantic as states/transitions system. Then, we define a set of predicate expressions which can be formulated about the states of activity diagram elements. To express temporal requirements related to the execution of an activity diagram, predicate expressions about activity elements are temporally quantified using the property specification pattern system proposed in [16].

6.1 Operational semantic of SysML activity diagram

During execution, at each instant, an activity has a specific state. This state is defined among others by : the activity status (not started, started, finished),

the states of all its actions, the states of its input and output parameters and the value of all the local variables used to express guards defined to control the tokens flows. The state of an action is defined by : its status (active, not active), the states of its incoming and outgoing control flows and the states of its input and output pins. According to this description, the operational semantic of an activity diagram can be represented by a high level states/transitions system as depicted in Figure 9.

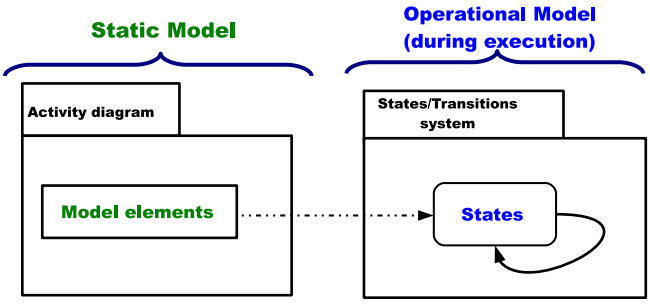


Fig. 9: Operational semantic of activity diagram

6.2 Predicate expressions about activity diagram elements

In this section, we present a sub-set of predicate expressions which can characterize the elements of an activity diagram during its execution. Let $ActivityName$ an activity, examples of such predicate expressions are :

1. If $actionName$ is action from activity $ActivityName$, then $[ActivityName].[actionName].isActive()$ is a valid predicate expression. Its value is True on a given state if $actionName$ is on execution.
2. If $actionName$ is action from activity $ActivityName$ and $ctlfName$ an incoming control flow of $actionName$, then $[actionName].incoming[ctlfName].isNotEmpty()$ is a valid predicate expression. The same expression can be defined for an output control flow.
3. If $actionName$ is action from activity $ActivityName$ and $PinName$ is its input pin, then $[actionName].input[PinName].isNotEmpty()$ is a valid predicate expression. The same expression can be defined for an output pin.
4. All the boolean OCL expressions about the objects manipulated in the activity $ActivityName$ are valid predicate expressions.
5. All the boolean expressions about the local variables used in $ActivityName$ are valid predicate expressions.
6. If $actExp$ is a valid predicate expression, then $not\ actExp$ is valid predicate expression.

7. If $actExp1$, $actExp2$ are valid expressions, then $actExp1$ *and* $actExp2$ and $actExp1$ *or* $actExp2$ are valid expressions.

6.3 Temporal expressions

The idea of property specification pattern system [16] is to allow users to construct complex properties from basic, assuredly correct building blocks by providing generic specification patterns (left side of Figure 10) encoding certain elementary properties: existence, absence, universality, bounded existence, precedence (chains), and response (chains), each specialized for a set of different scopes (right side of Figure 10) : globally, before R, after Q, between Q and R, after Q until R.

Given an activity diagram, requirements about its execution are interpreted as

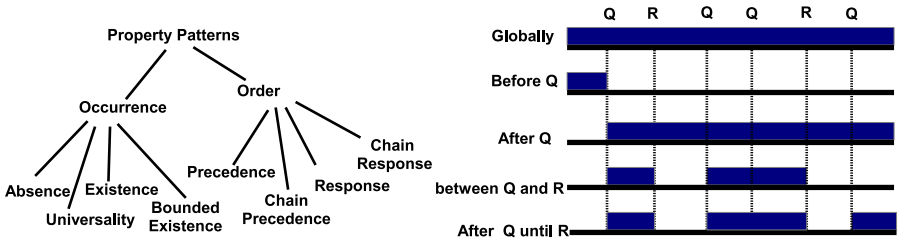


Fig. 10: Temporal pattern and scopes

properties over the states of its elements. These states can be characterized by the predicate expressions defined above. The SysML designer derives from functional requirements the elementary predicate expressions, then, he formalizes requirements by quantifying the predicate expressions by the necessary patterns and scopes to get temporal requirements about the activity execution. In the following, is given the grammar of the AcTRL (<Pred-exp> is predicate expression as defined above):

```
<AcTRL>::= <pattern> <scope>
<pattern>::= always <Pred-exp>
           | never  <Pred-exp>
           | eventually <Pred-exp>
           | <Pred-exp> precededing <Pred-exp>
           | <Pred-exp> following <Pred-exp>
 <scopes> ::= globally
           | before <Pred-exp>
           | after <Pred-exp>
           | between <Pred-exp> and <Pred-exp>
           | after <Pred-exp> until <Pred-exp>
```

6.4 Translation into CTL/LTL formulas

Functional requirements described using AcTLR can automatically translated into temporal logic formulas. A complete library is provided in [17], which propose translations into many formalisms (LTL, CTL, QREs, ...). So, we have to give a semantic for the defined predicate expressions according to the translation of activity diagram into Petri nets. As example, we consider the following predicate expression: $[ActivityName].[actionName].isActive()$, according to the translation of an action (Figure 6) will be translated into $(Marking(on\_A) =< 1 >)$, a proposition which means : the place on_A contains the mark $< 1 >$. As second example, the predicate expression $[actionName].output[PinName].isNotEmpty()$, according to the translation of output pins (Figure 6), will be translated into $(Marking(P\_OutApin)! =<>)$, which means : the place $P\_OutApin$ is marked.

7 Modular verification of requirements

Considering the compositional structure of activity diagrams can reduce the spatial and temporal complexity of their verification by model-checking. In this section, we propose an algorithm that guides the modular verification of SysML requirements on activities. We consider the verification of functional SysML requirements on a composite activity diagram where a set of call behavior actions is used to call set of activities. The proposed modular verification concerns the sub-activities and the composite activities according to their relations with SysML requirements. We assume that all functional requirements related to activities are expressed in AcTRL and depicted in a requirement diagram. By exploiting the $<< verify >>$ relationships, we generate a set of associations Set-Req-act containing all couples (ReqId, ActName), where ReqId is a requirement related to the activity ActName by a $<< verify >>$ relationship.
When performing the translation of an activity diagram into a modular Petri net, we generate a set of associations Set-ACT-PNM containing all couples (ActName, PNMname), where PNMname is a Petri net module translated from the activity ActName. When performing the translation of functional requirements expressed in AcTRL into temporal logic formulas, we generate a set of association Set-Req-LFor, which contains all couples (ReqId, lform), where Lform is a temporal logic formula derived from the requirement ReqId.
The verification of requirements on activities is achieving according to the algorithm 1. Changing the design of a sub-activity may influence the verification of a requirement related to its main activity. For this reason, the requirements related to a composite activity are verified after the validation of all the requirements related to their sub-activities. The algorithm begins by verifying the requirements related to sub-activities, then it process the verification of requirements related to composite activities until it reaches the main activity. The algorithm has complexity depending on the complexity of the model-checking (the function check()). It process N check; where N is the number of requirements.

Algorithm 1 Verify(ActName)

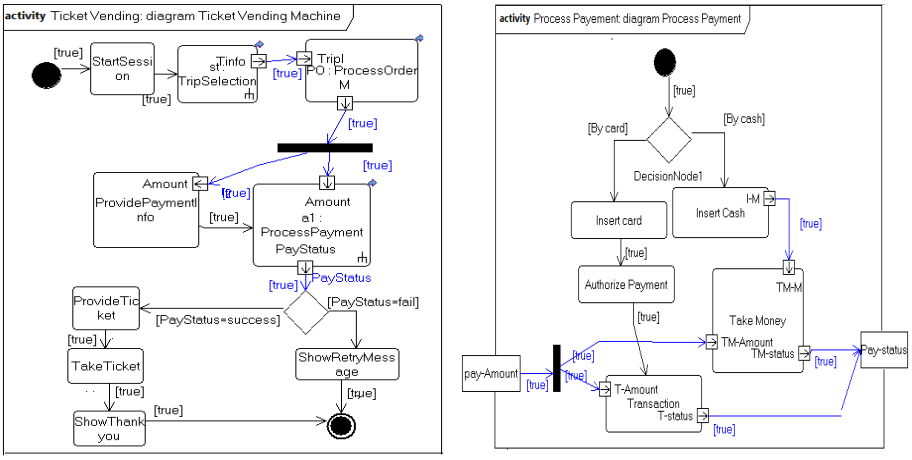
```
if (ActName does not contain call behavior actions) then
  Foreach (couple (ReqId, ActName)∈ Set-Req-act
  if ( (ActName, PNMname)∈ Set-ACT-PNM and (ReqId, lform) ∈ Set-Req-LFor)
  then
    check (lform, PNMname);
  end if
  EndForeach
else
  Foreach (sActName ∈ Sub-activities (ActName))
  Verify(sActName);
  EndForeach
  Foreach (couple (ReqId, ActName)∈ Set-Req-act
  if ( ActName, PNMname)∈ Set-ACT-PNM and (ReqId, lform) ∈ Set-Req-LFor)
  then
    check (lform, PNMname);
  end if
  EndForeach
end if
```

8 Application : A Ticket Vending Machine case study

In this section, we consider a Ticket Vending Machine(TVM) case study to illustrate our methodology. A TVM can be used to dispense tickets to passengers at a railway station. The behavior of the machine is triggered by passengers who need to buy a ticket. When passenger starts a session, TVM will request trip information from commuter. Passengers use the front panel to specify their boarding and destination place, details of passengers (number of adults and children) and date of travel. Based on the provided trip info, TVM will calculate payment due and display the fare for the requested ticket. Then, it requests payment options. Those options include payment by cash, or by credit or debit card. After that, the passenger chooses a payment option and processes to payment. After a success payment, the TVM prints and provides a ticket to passenger.
We specify the function of TVM by the activity diagram shown in Figure 11a. The activity diagram describes a composite activity which calls another activities. As example, we present the "process payment" sub-activity in the Figure 11b.

We specify the requirements to verify by activities using requirement diagrams. As example, two requirements are presented in Figure 12. They are expressed in AcTRL and related by a $<< verify >>$ relationship to activities. In this diagram extract, Set-Req-act = {(DREQ1, TicketVending), (DREQ2, Process Payment)}. From AcTRL expressions, we derive two logic formulas F1 and F2. As consequence, Set-Req-LFor = {(DREQ1, F1), (DREQ2, F2)}.

The activity diagram is translated into Petri net modules described in PNML. The running of the implemented ATL rules produces a XMI serialisation of the modular PNML document. The Figure 13 shows the structure of the derived



(a) A main activity diagram for TVM (b) A sub-activity representing the payment process

Fig. 11: Activities for TVM

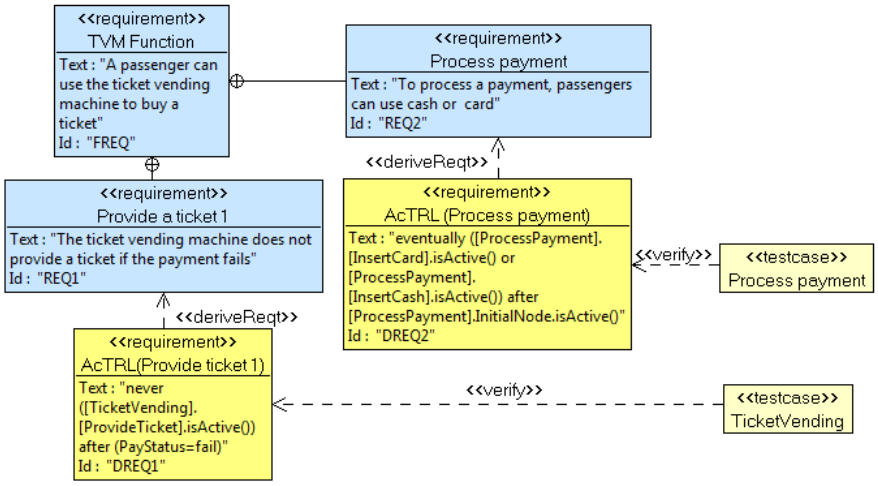


Fig. 12: A requirement diagram for TVM that uses AcTRL

PNML document. As the SysML activity diagram contains three sub-activities, the derived PNML document will be composed of a main Petri net and a Petri net module for each sub-activity. The set Set-ACT-PNM contains (TicketVending, Petri net TicketVending) and (Process Payment, Module ProcessPayment). By applying the algorithm 1, F2 will be checked in "Module ProcessPayment" then F2 will be checked in "Petri net TicketVending".

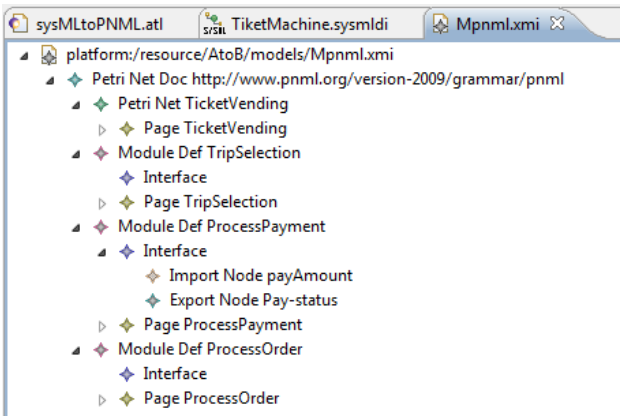


Fig. 13: Structure of the derived modular PNML

9 Conclusion and Future work

In this paper, we presented a methodology that proposes a modular verification of SysML specifications. The proposed methodology considers both requirements and activity diagrams. It consists on translating a composite activity diagram into modular Petri net. Then, it proposes a formalization of requirements related to activities by the proposition of AcTRL, which can be used by SysML designers and their expressions are translatable into temporal logics. Finally, an algorithm is proposed to guide the modular verification of SysML requirements. The translation from SysML activities to modular Petri net was fully automated using model to model transformation with ATL language. To illustrate the effectiveness of the proposed methodology, a practical case study was given.
As future work, we plan to automatize the translation of AcTRL expressions according to the translation of the activity diagram. Also, we plan to implement our methodology into complete framework. By completing these tasks, a SysML specification with requirements and activity diagram can be automatically verified using a Petri net tool. The next step will be the feedback of analysis results and their interpretation on SysML models.

References

1. OMG: OMG Systems Modeling Language (OMG SysML^{TM}) Version 1.2. (2010) downloadable from http://www.omg.org.
2. Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., Coq, T.: A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. Information and Software Technology **54** (2012) 569 – 590
3. Debbabi, M., Hassaine, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering: Assessing UML/SysML Design Models. 1st edn. Springer-Verlag New York, Inc., New York, NY, USA (2010)

4. Knorreck, D., Apvrille, L., de Saqui-Sannes, P.: TEPE: a SysML language for time-constrained property modeling and formal verification. ACM SIGSOFT Software Engineering Notes **36** (2011) 1–8
5. Foures, D., Vincent, A., Pascal, J.: ACTIVITYDIAGRAM2PETRINET : Transformation-Based Model In Accordance With The Omg SySML Specifications. In: Proceedings of the Eurosis, The 2011 European Simulation and Modelling Conference. (2011) 429–434
6. Rahim, M., Hammad, A., Ioulalen, M.: Modular and Distributed Verification of SysML Activity Diagrams. In: MODELSWARD 2013, 1st Int. Conf. on Model-Driven Engineering and Software Development, Barcelona, Spain. (2013) 202–205
7. Klein, F., Giese, H.: Joint structural and temporal property specification using timed story scenario diagrams. In: Fundamental Approaches to Software Engineering. Springer (2007) 185–199
8. Michael, W., Ekkart, K.: The Petri net markup language. In: Petri Net Technology for Communication-Based Systems. Springer (2003) 124–144
9. Linhares, Marcos Vinicius and de Oliveira, Rômulo Silva and Farines, J-M and Vernadat, François: Introducing the modeling and verification process in SysML. In: Emerging Technologies and Factory Automation (ETFA) IEEE Conference, IEEE (2007) 344–351
10. Eshuis, R.: Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology (TOSEM) **15** (2006) 1–38
11. Andrade, E., Macie, P., Callou, G., Nogueira, B.: A Methodology for Mapping SysML Activity Diagram to Time Petri Net for Requirement Validation of Embedded Real-Time Systems with Energy Constraints. In: Third International Conference on Digital Society, ICDS'09. (2009) 266–271
12. Foures, D., Albert, V., Pascal, J.C., Nketsa, A.: Automation of SysML activity diagram simulation with model-driven engineering approach. In: Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium. TMS/DEVS '12, San Diego, CA, USA, Society for Computer Simulation International (2012) 11:1–11:6
13. PNML.org: (Reference site for the implementation of Petri Net Markup Language (PNML)) url : http://www.pnml.org.
14. Farail, P., Goutillet, P., Canals, A., Le Camus, C., Sciamma, D., Michel, P., Crégut, X., Pantel, M.: The TOPCASED project: a toolkit in open source for critical aeronautic systems design. Ingenieurs de l'Automobile (2006) 54–59
15. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL-eclipse support for model transformation. In: Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference, Nantes, France. Volume 66., Citeseer (2006)
16. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the International Conference on Software Engineering, IEEE (1999) 411–420
17. Alavi, H., Avrunin, G., Corbett, J., Dillon, L., Dwyer, M., Pasareanu, C.: (Specification Patterns) url: http://patterns.projects.cis.ksu.edu.

Compatibility Analysis of Time Open Workflow Nets

Zohra Sbaï[1], Kamel Barkaoui[2], and Hanifa Boucheneb[3]

[1] Université de Tunis El Manar,
Ecole Nationale d'Ingénieurs de Tunis,
BP. 37 Le Belvédère, 1002 Tunis, Tunisia
zohra.sbai@enit.rnu.tn
[2] Conservatoire National des Arts et Métiers
292 rue Saint Martin, Paris Cedex 03, France
kamel.barkaoui@cnam.fr
[3] Ecole Polytechnique de Montréal,
P.O. Box 6079, Station Centre-ville, Montréal, Québec, Canada
hanifa.boucheneb@polymtl.ca

Abstract. Because of their expressive power, Petri nets are widely used in the context of concurrent and distributed systems. We study in this paper a sub class of time Petri nets, named time open workflow nets (ToWF-nets), used to interconnect time constrained business processes. To interact correctly with each other, these processes have to be compatible. This include not only composability of the involved processes but also the correct execution of the overall composite system. In this context, we suggest in this paper to study the compatibility of ToWF-nets in different aspects and to provide a formal approach to characterize and verify this property. This approach is based on qualitative and quantitative analysis ensured by TCTL model checking.

Keywords: Time open workflow nets, Reachability analysis, Compatibility, TCTL

1 Introduction

The workflow technology has shown a great interest to be adopted within organizations or even inter organizations. A workflow is the result of the automation of a business process, in whole or in part [35]. A business process consists of a number of tasks and ensure all the conditions that determine their order. A business process which involves different partner companies is said to be inter organizational. Indeed, it is a specific representation for which coordination mechanisms between activities, applications or participants can be managed by a workflow management system (WfMS). The success of workflow technology explains the fact that the number of emerging WfMS is growing fast. And therefore the need for effective mechanisms and tools for modeling and analysis of workflow processes is crucial.

Open workflow nets (oWF-nets) form a sub class of Petri nets which is successfully used to model workflow processes which communicate with other partners via interfaces. This class is promisingly used in the context of Web services orchestration and choreography. In fact each service in a composition is modeled by a workflow net augmented by interface places used to communicate with other services. In this way, one can guarantee the conversation between the processes interacting with each other. The conversation considered here is involved through the two well known behaviors: operational and control. An operational behavior is a behavior specific to each partner according to its business logic. A control behavior describes the general behavior of any process related to composite Web services. While we focused in a previous work [33] on the verification of oWF-nets, we propose in this paper to extend oWF-nets by modeling timing constraints and to study their analysis.

Several time Petri nets extensions were proposed in the literature which differ in their semantics and their analysis techniques. We propose to adopt in this work the time Petri nets, proposed by Merlin [31], in which transitions are labeled by intervals specifying the minimum and maximum delays of their firings. We extend, therefore oWF-nets by associating with each transition a minimum and maximum amount of time needed to its execution. The obtained model is said to be time open workflow net (ToWF-net). We define formally this model and present its semantics as well as the computation of its state space. The efficient construction of the state space leads to efficient techniques of ToWF-nets reachability analysis.

Dealing with time in inter organizational processes, we propose to study the compatibility of the processes communicating together. This property is not only related to the ability of processes to communicate (i.e. composability) but also to the correct and deadlock-free execution of the composite process. In this context, we propose to define compatibility classes of ToWF-nets and to emphasize a method of their verification.

To verify ToWF-nets compatibility, we propose to use formal methods due to their solid theoretical basis. More precisely, we present an analysis method based on model checking of the studied properties. In fact, Model checking is an automated verification technique for proving that a model satisfies a set of properties specified in temporal logic. Given a concurrent system Σ and a temporal logic formula φ, the model checking problem is to decide whether Σ satisfies φ. Hence, we have to formulate in temporal logic the properties to be verified. This kind of verification is situated at the design phase, allowing thus to find design bugs as early as possible and therefore to reduce the cost of failures. This, especially, permits us to check as early as possible if two or more processes are compatible before their composition. We express the proposed compatibility properties in Timed Computation Tree Logic (TCTL).

The rest of this paper is organized as follows. We propose in section 2 the ToWF-nets to model inter organizational workflow processes with timing delays. The same section presents the semantics of ToWF-nets in terms of states and their evolution and exposes a case study. Section 3 is dedicated to present some

results of the reachability analysis of ToWF-nets. We focus in section 4 on the verification of ToWF-nets compatibility. We begin with expressing the properties in TCTL and then we present some experiments in Romeo model checker. Section 5 exhibits related work and finally section 6 concludes the paper and announces future work.

2 Time open WorkFlow nets

In this section, we propose a new sub class of time Petri nets modeling workflow processes with interface places used to communicate with other partners. To begin with, we present a Petri nets modeling of communicating processes and then we propose a time extension.

2.1 Petri nets modeling of communicating workflows

Nowadays, many organizations are implementing their business functionality and outsource their services on the internet. Thus, the selection as well as inter organizational and heterogeneous integration with efficiency and effectiveness of Web services during the execution has become an important step in Web services applications. In particular, if no service can meet the needs of the user, there should be a possibility to combine existing services to meet the demands required by the user. This trend has led to the notion of the composition of Web services.

In fact, the composition or aggregation of Web services is a process that involves building new services or aggregates called composite services by assembling existing services. The composite service is a value added service that can be the distribution of basic services or composite ones.

This composition can be modeled by means of a Petri net class named open workflow nets [25,33,29,30]. We model each involved process by an open workflow net possessing interface places used to communicate with other processes. Thus the conversation and interaction between the involved processes are guaranteed. The communication considered here is entangled through operational and control behaviors. The operational behavior is a behavior specific to each partner according to its business logic while the control behavior describes the general behavior of any process related to composite Web services.

As mentioned above, open workflow nets are mainly an extension of workflow nets (WF-nets) to model workflow processes which interact with other workflow processes via interface places. Simple WF-nets [7]is a result of Petri nets' application to workflow management. The choice of Petri nets is based on their formal semantics, expressiveness, graphical nature and the availability of Petri nets based analysis techniques and tools.

A Petri net is a 4-tuple $N = (P, T, F, W)$ where P and T are two finite non-empty sets of places and transitions respectively, $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ is the weight function of N satisfying $W(x, y) = 0 \Leftrightarrow (x, y) \notin F$. If $W(u) = 1 \ \forall u \in F$ then N is said to be ordinary net and it is denoted by $N = (P, T, F)$. For every node $x \in P \cup T$, the

set of input nodes of x is defined by ${}^\bullet x = \{y|(y,x) \in F\}$ and the set of output nodes is denoted by $x^\bullet = \{y|(x,y) \in F\}$. We refer the reader to [8], for more Petri nets notations used in this paper.

A Petri net which models a workflow process is said to be a WF-net [3]. An ordinary Petri net $N = (P, T, F)$ is a WF-net iff N has one source place i named initial place (containing initially one token) and a sink place f named final place. In addition to this characteristic, in a WF-net, every node $n \in P \cup T$ is on a path from i to f.

For a composition, we propose to model each Web service by a WF-net specifying the set of tasks to be performed and their routing. The conversation between the different Web services is ensured by communication places used for messages sending. We are thus using open WF-nets (oWF-nets) which generalizes the classical WF-nets by introducing interface places for asynchronous communications with partners. Hence, we model a composition by a set of oWF-nets communicating via interface places. These places connect only transitions of different processes.

2.2 Time extension

When incorporating time constraints, different extensions of Petri nets were proposed. In general, when the time constraints are specified by constants(durations), the associated extension is said Timed Petri nets. This consider constant durations attached to places (P-Timed Petri nets) or transitions (T-Timed Petri nets). When these constraints are specified by intervals (delays) specifying the minimum and the maximum amounts of time needed for transitions' firing, the associated extension is called Time Petri nets. These intervals are attached to places (P-Time Petri nets), transitions (T-Time Petri nets) or arcs (A-Time Petri nets) leading thus to different extensions with variant semantics.

Petri nets form a powerful formalism for the expression of control flow in business processes [2,19,18,16]. In addition, several studies [1,6,27,22] have shown the importance of temporal reasoning in the specification of workflow systems.

In [27], the authors extend the simple WF-net presented by van der Aalst [3] by associating with each transition an amount of time representing the duration of the task it models. They propose a temporal extension of the WF-net, called Time WF-net and scored TWF-net. Timing discipline adopted in the proposed model announced that each enabled transition must start running immediately, otherwise it will be disabled, and once started, this transition can not be delayed, i.e. its duration should be respected. While this approach is strict in the fixed duration required, time Workflow nets (TWF-nets) incorporate time constraints of activities by associating to each transition an interval specifying its firing time [12,15,27].

Since this time consideration is flexible and given that we are interested by modeling the composition of workflow processes with time constraints, we propose the time open workflow net model (ToWF-net). This model associates a static time interval to each transition of an open workflow net to express the

execution time or delay of corresponding activity. The formal definition of a ToWF-net model net is the following:

Definition 1 ***(ToWF-net)***
A Time Open Workflow Net N is a tuple (P, T, F, FI, I, O) with:

- *P is a set of places,*
- *T is a set of transitions,*
- *I is a set of places representing input interfaces which are responsible for receiving messages from other services: ${}^\bullet I = \emptyset$.*
- *O is a set of places representing output interfaces that are responsible for sending messages to other services: $O^\bullet = \emptyset$.*
- *I, O and P are disjoint. I and O connect transitions of different partners.*
- *$F \subseteq ((P \cup I) \times T) \cup (T \times (P \cup O))$ is the flow relation,*
- *$FI : T \to \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$ is the function that associates with each transition $t \in T$ a static firing interval, i.e. $FI(t) = [minFI(t), maxFI(t)]$ where $minFI(t)$ and $maxFI(t)$ are rational numbers representing respectively the minimal and maximal firing time,*

The marking of N is a vector of $\mathbb{N}^P$ such that for each place $p \in P$, $M(p)$ is the number of tokens in p. The initial marking of N is M_i knowing that M_p is used to denote a marking for which $M(p) = 1$ and $M(q) = 0\ \forall q \in (P \cup I \cup O) \backslash \{p\}$.

A transition t is said to be enabled in a marking M if the required tokens are present in the input places of t. We denote by $En(M)$ the set of all the transitions enabled in the marking M. A transition t is said disabled by the firing of t' in M if it is enabled in M but it isn't in $M -^\bullet t'$. When focusing of newly enabled transitions after firing a transition t from M and leading to M', we denote by $NEn(M, t)$ the set of transitions enabled after this firing.

$NEn(M, t) = \{t' \in En(M') | t' = t \vee \neg M \geq^\bullet t +^\bullet t'\}$.

When a transition t becomes enabled, its firing interval is set to its static firing interval $FI(t)$. The lower and upper bounds of $FI(t)$ decrease synchronously with time, until t is fired or disabled by another firing. t can fire, if the lower bound of its firing interval reaches 0, but when upper bound of its firing interval reaches 0, t must be fired without any additional delay (strong semantic). The firing takes no time but may lead to another marking.

Let us define first the state of a ToWF-net and then the transition relation.

Definition 2 *A state in a ToWF-net represents the state of the process modeled with ToWF-net after the occurrence of an event. Formally, a state in a ToWF-net is a pair (M, Int) where:*

- *M is a marking,*
- *Int is a firing interval function, $Int : En(M) \to \mathbb{Q}^+ \times \mathbb{Q}^+ \cup \{\infty\}$. We denote $Int(t) = [minInt(t), maxInt(t)]$.*

The initial state of a ToWF-net is (M_0, Int_0) where $M_0 = M_i$ (since in a ToWF-net, only i contains initially one token) and $Int_0(t) = FI(t)\ \forall t \in En(M_0)$

Starting from the initial state (M_0, Int_0), the net evolves following the occurrence of events. An event corresponds to either a transition firing or a time progression. Hence, the transition relation between a state $s_1 = (M_1, Int_1)$ and $s_2 = (M_2, Int_2)$ is defined by $\xrightarrow{t}$ in case of a firing and by $\xrightarrow{d}$ in case of time progression. The conditions and the computation of the resulting state after an event occurrence are defined as follows:

1. $s_1 \xrightarrow{t} s_2$ if and only if s_2 is immediately reachable from s_1 by firing the transition t, i.e.
   $t \in En(M_1)$ and $minInt_1(t) = 0$,
   $M_2 = M_1 - {}^\bullet t + t^\bullet$, and
   $\forall t' \in En(M_2),\ Int_2(t') = \begin{cases} FI(t') \ if t' \in NEn(M_1, t) \\ Int_1(t') \qquad otherwise \end{cases}$
2. $s_1 \xrightarrow{d} s_2\ \forall d \in \mathbb{R}$ if and only if the state s_2 is reachable from s_1 by time progression with d time units, i.e.
   $minInt_1(t) + d \leq maxFI(t)$,
   $M_2 = M_1$, and
   $\forall t \in En(M_1),\ Int_2(t) = [Max(0, minInt_1(t) - d), maxInt_1(t) - d]$

Therefore, the semantics of a ToWF-net N is defined by a transition system $(S, s_0, \rightarrow)$ where S is the set of all the states reachable from the initial state s_0 by the transition relation $\rightarrow$ defined above.

2.3 A case study

In order to illustrate the proposed ToWF-net, we propose to study the process of awarding of pensions to handicapped persons. This process requires the collaboration of three organizations:

- The prefecture which manages scholarships and grant of license to the disabled.
- A medical entity that is responsible for negotiating the date of appointments with patients and collecting the medical informations.
- The Town Hall which establishes certificates, births extracts, etc.

The allocation of pension process is seen as a collaboration between the services offered by these organizations.

In fact, citizens with disabilities ask a government scholarship. To start the process, citizens request the form corresponding to the prefecture. Once the citizen receives the form, he fills it and sends it to the prefecture. The latter seeks medical entity to consider disability that the citizen presents. Medical entity subsequently contacts the citizen to negotiate with him about a date of appointment. Once an appointment is fixed, and after reviewing the citizen, the entity establishes a medical examination report and forwards it to the prefecture. Meanwhile, the prefecture asked the town hall to establish a certificate of residence of the citizen. Once the certificate of residence and the medical report is received, the prefecture makes the final decision.

The figure 1 shows a screenshot of this composition involving the four processes relevant to the Applicant, the Prefecture, the Town Hall and the Medical Unit.

These processes are interconnected with available interfaces that facilitate communication and exchange of messages between them. These interfaces correspond to places denoted by I_{sn} (for input interfaces) and O_{sn} (for output ones), where s is the service number and n is the interface number in each category. Note that each input interface place of a service has an equivalent output interface of another service and this will guarantee the services communication. For sake of clarity, in this example, the interfaces are given names which explain the sequence of exchanged messages between partners.

The various services are forced to respect the different temporal properties of each service, in what follows , we mention a few of them:

- Once the medical entity proposes dates for appointment to the citizen, it must receive the confirmation within 24 hours.
- Once the application for the grant is received, the prefecture sends its final decision to the citizen, after at least 49 hours and not more than 180 hours.
- The medical report may be sent to the prefecture after at least 24 hours and up to 48 hours of sending the medical examination.
- The receipt of the result of the request is within 210 hours after sending the request of the purse.
- Two hours is the maximum time to review a citizen in medical entity.
- The time of receipt of the certificate of residence and review of citizen ratio is up to three hours.
- Negotiation of the appointment date between the citizen and the medical entity runs for up to one hour.

We present in the following section the analysis of reachability of the Web services composition modeled by ToWF-nets and we expose the case study reachability analysis in the tool Romeo [21].

3 Reachability analysis of ToWF-nets

After the formal definition of ToWF-nets, we focus now on their reachability analysis. This analysis is based on the efficient construction of the state space.

By analogy with the marking graph defined in the context of an ordinary Petri net, we define a state space by a graph containing all accessible states of a ToWF-net from the initial state. Therefore, to calculate the state space of a ToWF-net, we must be able to calculate the reachable states by activating the enabled transitions.

Definition 3 *The state space of a ToWF-net has the following structure:* $SS = (S, \rightarrow, s_0)$*; where* S *is the set of nodes in form* (M, Int) *representing the reachable states from the initial one* $s_0 = (M_i, Int_0)$ *;* $\rightarrow$ *represents the transition relation which defines the evolution from one state to another.*

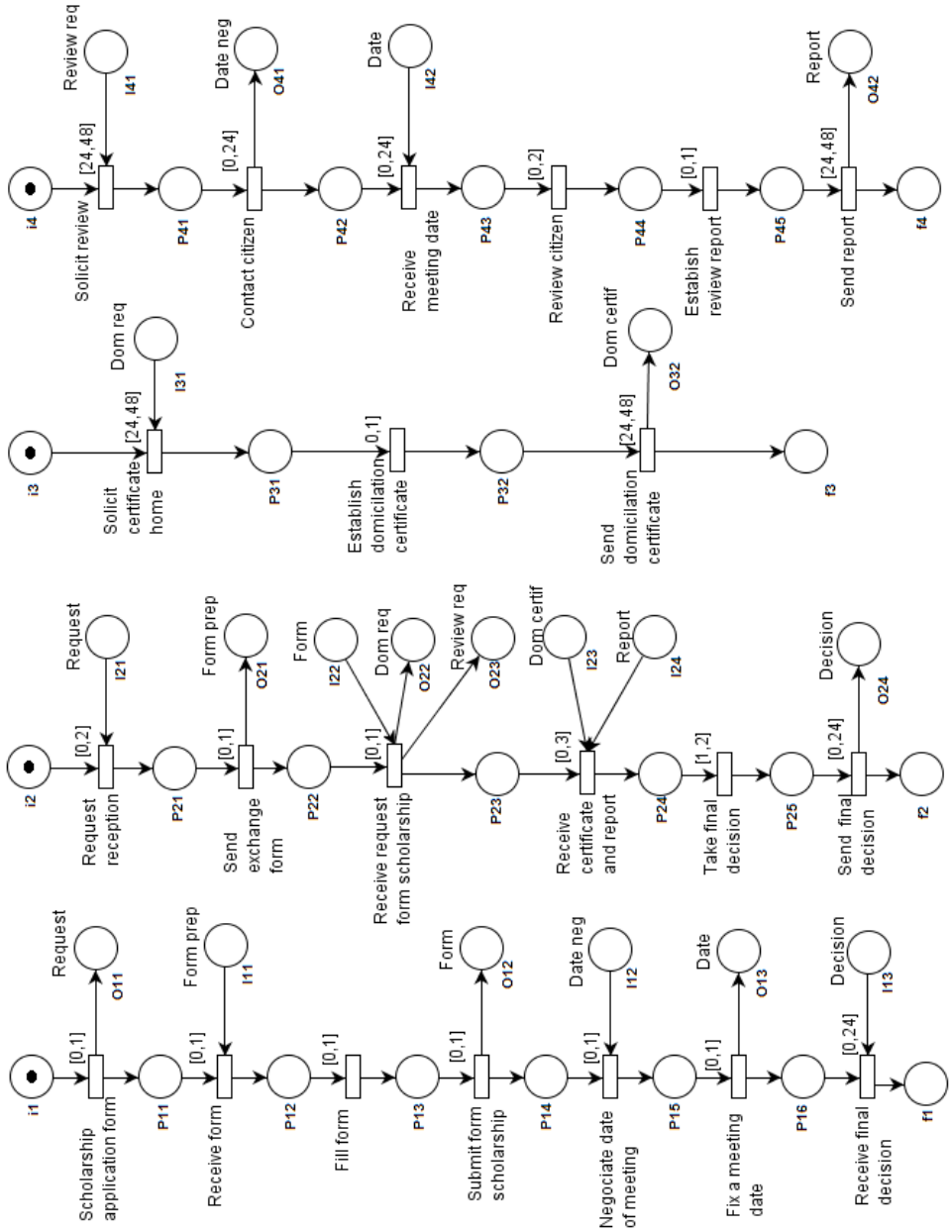


Fig. 1. Modeling of the case study with ToWF-nets

$S = \{s | s_0 \xrightarrow{*} s\}$ is the set of reachable states of the model, and $\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$.

The reachability analysis [12] in timed models (such as time extensions of Petri nets as well as timed automata) is based in general on abstraction, which preserves reachability properties. Such an abstraction for timed models, consists in considering only one node for all states reachable from the same firing sequence while abstracting from their firing times. The grouped states, known as state classes, are then considered modulo some equivalence relation preserving properties of interest.

In return, the state class method is intended to provide a finite representation of the infinite state space of any bounded time Petri net.

Technical classes produce for a large class of time nets a finite representation of their behavior states, which allows a reachability analysis similar to that permitted for Petri nets by the technique of marking graph.

The state classes can be represented by a marking and a firing domain. Formally, a state class is a couple (M, D) where M is a marking and D is characterized by a set of atomic constraints over the firing delays of enabled transitions: $minFI(t) \leq t \leq maxFI(t) \quad \forall t \in En(M)$.

Note that the initial class coincides with the initial state of the network. This initial class is (M_0, D_0) where $M_0 = M_i$ and D_0 corresponds to the firing domains of transitions enabled in M_0.

All states within the same node share the same untimed information and the union of their time domains is represented by a set of atomic constraints handled efficiently by means of a Difference Bound Matrix (DBM) [32]. A DBM form a system of linear inequalities which constrain single variables $(v_1 ... v_n)$ and their differences within limits identified by coefficients b_{ij}. This is formally expressed as:

$$\begin{cases} v_i - v_j \leq b_{ij} \quad i, j \in [0..n],\ b_{ij} \in \mathbb{Q} \\ v_0 = 0 \end{cases}$$

In terms of behavior, this state classes group preserves highly the states traces, and thus the safety properties.

The computation of the state class graph is necessary at this point to perform the various reachability analysis. Among the abstractions proposed in the literature [9], [10,36], we consider here the state class graph method [9] for its advantage, over the others, which is the finiteness property for all bounded time Petri nets (while using some approximations).

Romeo [21] is a software studio dedicated to time Petri nets analysis. It is developed at IRCCyN by the Real-Time Systems Team. It performs analysis on T-Time Petri nets and on one of their extensions to scheduling. We chose Romeo because it performs, among other features, the computation of the State Class Graph (SCG) and a graphical simulation of a T-Time Petri net. It is also a model checker for a subclass of TCTL formulas.

Therefore, we used Romeo to generate the SCG of our case study. We begun with composing the different services by superposing the interface places which correspond to the same interface communication. We then simulated the overall

obtained net in Romeo. Figure 2 presents a snapshot of the case study analysis conducted in Romeo and especially the beginning of the file generated by Romeo and which contains the SCG.

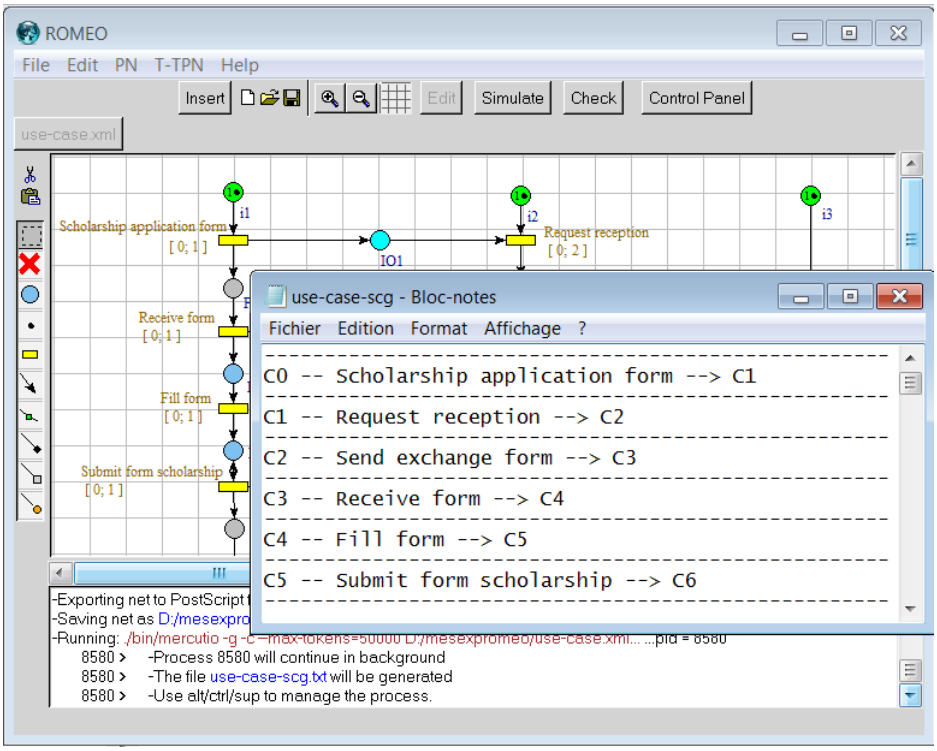


Fig. 2. Analysis of the case study in Romeo

4 Compatibility analysis of ToWF-nets

The analysis of the state space is very significant to the extent that it can reveal important characteristics of the modeled system, about its structure and dynamic behavior. However, for a more accurate verification, we should not be limited to this type of checking rather another specific properties. Indeed, we focus in this section on the formal verification of compatibility properties of ToWF-nets. We propose to use model checking method to verify these properties since this method permits an exhaustive verification over all the possible executions. Given a concurrent system Σ and a temporal logic formula φ, the model checking problem is to decide whether Σ satisfies φ. Hence, we have to formulate in temporal logic the properties to be verified.

4.1 Model checking TPN-TCTL

Real systems often have behaviors that depend on time. The ability to manipulate and model the temporal dimension of the events that take place in the real world is fundamental in many applications. These applications may involve banking, medical, or multimedia applications. The variety of applications motivate many recent studies that aim to integrate all the features necessary to take into account the time during verification.

TCTL (Timed Computational Tree Logic) is a timed extension of the temporal logic CTL. TCTL added to CTL a quantitative information on the delays between actions. It is built from atomic propositions, logical connectors and temporal operators (U, F, G, X, etc.). The TCTL temporal logic can be used to check the properties of a time Petri net.

The syntax of TCTL formulas is inductively defined by:

$\varphi ::= false \mid \neg\varphi \mid \varphi \wedge \varphi \mid A(\varphi\ U_I\ \varphi) \mid E(\varphi\ U_I\ \varphi)$

where p denotes a proposition, φ denotes a formula and $I = [a, b]$ or $[a, \infty[$ with $a \in \mathbb{N}$ and $b \in \mathbb{N}$.

A and E are temporal quantifiers over the set of executions. $A\varphi$ announces that all the executions from the current state satisfy the property φ. $E\varphi$ states that from the current state, there exists an execution which satisfies φ. Finally $\varphi\ U_I \psi$ means that the property φ is true until ψ is true, and ψ will be true in the time interval I.

We can use other compositional temporal operators [5]: $\mathrm{EF}_I\ \varphi = \mathrm{E}(\ \mathrm{true}\ \mathrm{U}_I\ \varphi)$ (Possibility), $\mathrm{EG}_I\ \varphi = \neg\ \mathrm{AF}_{\ I}\ \neg\varphi$ (All locations along an execution), $\mathrm{AF}_I\ \varphi = \mathrm{A}(\ \mathrm{true}\ \mathrm{U}_I\ \varphi)$ (Locations along all executions), $\mathrm{AG}_I\ \varphi = \neg\ \mathrm{EF}_{\ I}\ \neg\varphi$ (All locations along all executions).

Semantically, TCTL formulas are interpreted on states and execution paths of a model $M = (S, V)$ where S is a transition system and V is a valuation function that associates with each state the set of atomic propositions it satisfies. [26]

To interpret a TCTL formula on an execution path, we introduce the notion of dense execution path. Let $s \in S$ be a state of S, $\pi(s)$ the set of all execution paths starting from s, and $\rho = s_0 \overset{d_0 t_0}{\rightarrow} s_1 \overset{d_1 t_1}{\rightarrow} s_2 ...$ an execution path of s. The dense path of the execution path ρ is the mapping $\hat{\rho} : R^+ \rightarrow S$ defined by: $\hat{\rho}(r) = s^i + \delta$ such that $r = \sum_{j=0}^{i-1} d_j + \delta$, $i \geq 0$ and $0 \leq \delta \leq d_i$.

The formal semantics of TCTL is given by the satisfaction relation defined as follows:

- $M, s \nvDash false$,
- $M, s \vDash \phi$ iff $\phi \in V(s)$,
- $M, s \vDash \neg\varphi$ iff $M, s \nvDash \varphi$,
- $M, s \vDash \varphi \wedge \psi$ iff $M, s \vDash \varphi$ and $M, s \vDash \psi$,
- $M, s \vDash \forall(\varphi \cup_I \psi)$ iff $\forall \rho \in \pi(s)\ \exists r \in I, M, \hat{\rho}(r) \vDash \psi$ and $\forall 0 \leq r' \leq r\ M, \hat{\rho}(r') \vDash \varphi$,
- $M, s \vDash \exists(\varphi \cup_I \psi)$ iff $\exists \rho \in \pi(s)\ \exists r \in I, M, \hat{\rho}(r) \vDash \psi$ and $\forall 0 \leq r' \leq r\ M, \hat{\rho}(r') \vDash \varphi$,

When interval I is omitted, its value is by default $[0, \infty[$.

The Time Petri net model is said to satisfy a TCTL formula φ iff $M,\ s_0 \vDash \varphi$.

The logic TCTL allows writing temporal properties with a quantification of the time. We chose this approach because it is decidable and PSPACE-complete for bounded Petri nets [14].

The authors of [24] have gone further by defining a sub-class of TCTL for time Petri nets in dense time, called TPN-TCTL. They proved the decidability of model-checking of TPN-TCTL on Petri nets and showed that its complexity is PSPACE.

Definition 4 *The temporal logic TPN-TCTL is defined inductively by:*

$TPN\text{-}TCTL ::= false \mid \varphi \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \varphi \Rightarrow \psi \mid E\varphi U_I \psi \mid A\varphi U_I \psi \mid EG_I\varphi \mid AG_I\varphi \mid AF_I\varphi \mid EF_I\varphi \mid AG(\phi_1 \Rightarrow AF_{[0,d]}\phi_2)$.

Where φ and $\psi \in$ TPN-TCTL,

$I = [a, b]$ or $[a, b[$ with $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$.

ϕ_1 and ϕ_2 are propositions on markings.

$\forall G(\phi_1 \Rightarrow \forall F_{[0,d]}\phi_2)$ means that from the current state, any occurrence of marking ϕ_1 is followed by an occurrence of marking ϕ_2 less of d units of time later.

Romeo permits a practical implementation of the verification of properties described in TPN-TCTL. It is therefore possible to model check on the fly temporal quantitative properties. That's why we investigate in the following section the TCTL expression of the compatibility property and hence its verification in Romeo.

Before this, let us recall the notation used by Romeo to implement a TPN-TCTL property:

$TPN\text{-}TCTL_{Romeo} = E(p)U[a, b](q) \mid A(p)U[a, b](q) \mid EF[a, b](p) \mid AF[a, b](p) \mid EG[a, b](p) \mid AG[a, b](p) \mid EF[a, b](p) \mid (p) \rightarrow [0, b](q)$.

where p, q: GMEC; U: until; E: exists; A: forall; F: eventually; G: always; $\rightarrow$: response; a: integer; b integer or inf (to denote ∞).

$GMEC = a{*}M(i)\{+, -\}b{*}M(j)\{<, <=, >, >=, =\}k \mid deadlock \mid bounded(k) \mid p\ and\ q \mid p\ or\ q \mid p \Rightarrow q \mid not\ p$.

M: keyword (marking); *deadlock*, *bounded*: keywords; i, j:place indexes; a, b, k :integers ; $*, +, -, and, or, \Rightarrow, not$: usual operators ; p, q: GMEC

The syntax $(p) \rightarrow [0, b](q)$ denotes a leads to property meaning $AG((p)$ imply $AE[0, b](q))$. E.g. $(p) \rightarrow [0, b](q)$ holds if and only if whenever p holds eventually q will hold as well in $[0, b]$ time units.

4.2 TCTL characterization of the compatibility property

From a behavioral point of view, two (or more) processes are said to be compatible if they can interact correctly: this means that they can exchange the same type of messages and the composite system does not suffer from the deadlock problem. This leads us to distinguish between a syntactic compatibility which

concerns the verification of the interfaces conformance and a semantic compatibility which is related to check the absence of deadlocks. We investigate in this paper the analysis of the semantic compatibility.

But before this, let us define the composite system obtained from the superposition of a number of syntactically compatible ToWF-nets. The composed system N of nbX ToWF-nets $N_1 ... N_{nbX}$ consists of all ToWF-nets which share interface places, i.e. every place of N which is an input interface of a WF-net is also an output place of another WF-net in the composition. Trivially, N can be seen as a time Petri net with nbX input places and nbX output places. The initial marking of N is $M_0 = \sum_{s=1}^{nbX} i_s$.

According to [11,20,28], the compatibility is closely related to the absence of deadlock in the composite system. They considered that two oWF-nets are compatible if they can reach their final states. In addition to this condition, we characterize the compatibility in ToWF-nets by the timing constraints respect.

In this direction, we define three classes of compatibility:

- Partial compatibility: A composed system N is partially compatible if it is deadlock-free.
- Total Compatibility: A composed system N is compatible if N is already partially compatible and furthermore, it guarantees the proper termination.
- Perfect compatibility: A composed system is perfectly compatible if it verifies the total compatibility as well as the deadline constraints.

We focus here on formulating the three types of compatibility properties: partial compatibility, total compatibility and perfect compatibility. Let us consider the following:

- nbX: is the number of processes;
- nbp: is the number of places in a given process;
- nbi: is the number of interface places available in a composition;
- i_s: is the input place of the process number s.
- f_s: is the output place of the process number s.

– **Partial compatibility**

To assure its partial compatibility, we have to check the absence of deadlock in a composition. The process is deadlock-free if there is a transition allowed for any marking except the final marking M_{fin} in which all the final places f_s ($s = 1..nbX$) are marked. This property is expressed as follows:

$$\forall M \in [M_0\rangle,\ M_{fin} \in [M\rangle$$

In TCTL, the deadlock-freeness can be expressed as "for all the executions from the initial state, no deadlock will be encountered until the final state is reached". For the final state, it suffices to check if the final places are marked. Hence, the expression of the partial compatibility in TCTL is given as follows:

$$AG_{[0,\infty[}((not\ MF) \quad \Rightarrow \quad not\ deadlock)$$

where *deadlock* is a proposition which returns true iff there is no enabled transition from the current state; and MF is a proposition on the marking M_{fin} in which each final place contains at least one token.

$$MF = \overset{nbX}{\underset{s=1}{\wedge}} M(f_s) >= 1$$

Here we focus only on the arrival of tokens to final places and we don't care if the other places contains tokens or not.

– **Total compatibility**

Having expressed the partial compatibility, we focus here on the expression of the property of proper termination in TCTL. This property allows the process to complete its execution in any case, but at the time of termination, all places of ToWF-nets must be empty except for the final places which must have one token. Verifying the proper termination consists in checking the existence of a marking M for which all places are empty except the output ones. The expression of this property is given as follows:

$$\forall M \in [M_0\rangle : M(f_s) \geq 1 \; \forall s \in \{1, .., nbX\} \quad \Rightarrow \; M = \textstyle\sum_{s=1}^{nbX} f_s$$

In TCTL, this property (proper termination) is formulated as follows:

$$AF_{[0,\infty[} \; StrictMF$$

Where $StrictMF$ is a proposition on the marking ensuring exactly one token in each final place f_s and no tokens in all the other places including the interface places.

$$StrictMF = \overset{nbX}{\underset{s=1}{\wedge}} \; (\overset{nbip}{\underset{p=1}{\wedge}} (M(p) = 0) \quad \wedge \; (M(f_s) = 1)) \quad \wedge \quad (\overset{nbi}{\underset{i=1}{\wedge}} M(I_i) = 0)$$

In this definition, we used nbip to denote the number of places except the final place for a process.

– **Perfect compatibility**

Here, we have to check the deadlock-freeness and the proper termination taking into account the overall deadline constraint.

Let us consider that a process has to reach his final state in Tm time units. The proper termination within this delay is expressed as follows:

$$AF_{[0,Tm]} \; StrictMF$$

Hence, the perfect compatibility of a composition of ToWF-nets is ensured iff:

– $AG_{[0,Tm]}((not \; MF) \quad \Rightarrow \quad not \; deadlock)$
– $AF_{[0,Tm]} \; StrictMF$

4.3 On the fly model checking of ToWF-nets composition

We report in this section some results related to the verification of compatibility and soundness properties of the composition of ToWF-nets. This verification is ensured by Romeo since it implements an on the fly model checking algorithm of TPN-TCTL properties.

Let us study the simple composition of ToWF-nets of figure 3. One can easily see that no deadlock will be encountered until the final places will be marked. Hence the partial compatibility is satisfied as proven in figure 4. Nevertheless, the execution of transitions T_4 and T_5 of the second process leads to two tokens in the place f_2; which leads to violate the property of total compatibility. Figure 5 shows the negative result for this property and draws a trace.

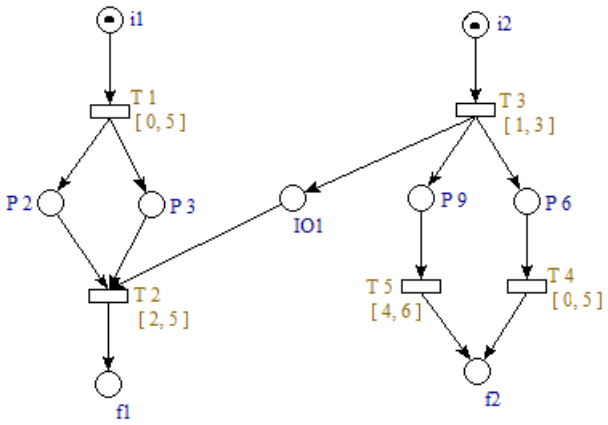


Fig. 3. A composition of ToWF-nets satisfying the partial compatibility but not the total compatibility

Let us now return to the case study given in figure 1. In order to check the partial compatibility of the involved processes, we formulate the correspondant TCTL formula as follows :

$$AG[0, inf]((not\ (M(30) >= 1 \text{ and } M(21) >= 1 \text{ and } M(23) >= 1 \text{ and } M(28) >= 1)) \Rightarrow not\ deadlock)$$

Where 30, 21, 23 and 28 are the indexes associated by Romeo to respectively the places f_1, f_4, f_3 and f_2.

Figure 6 draws a snapshot of a the verification of the partial compatibility of the four processes involved in the composition. As we can see in the figure, the result is *true* and hence the partial compatibility is ensured. The total compatibility characterized by the partial compatibility and the following formula is also verified for this example:

$$AF[0, inf](M(30) = 1 \text{ and } M(21) = 1 \text{ and } M(23) = 1 \text{ and } M(28) = 1 \text{ and } M(1) = 0 \text{ and } M(1) = 0 \text{ and } M(2) = 0 \text{ and } M(3) = 0 \text{ and } .. \text{ and } M(36) = 0)$$

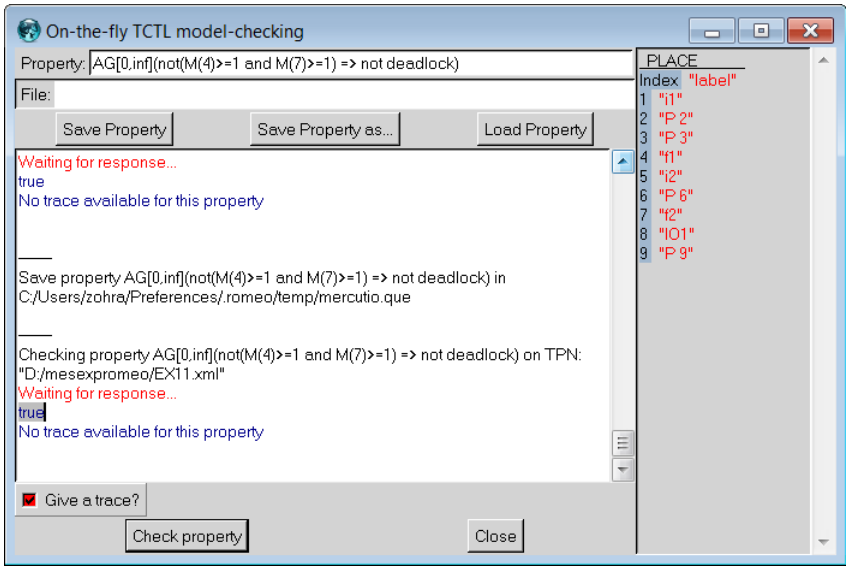


Fig. 4. Test of partial compatibility

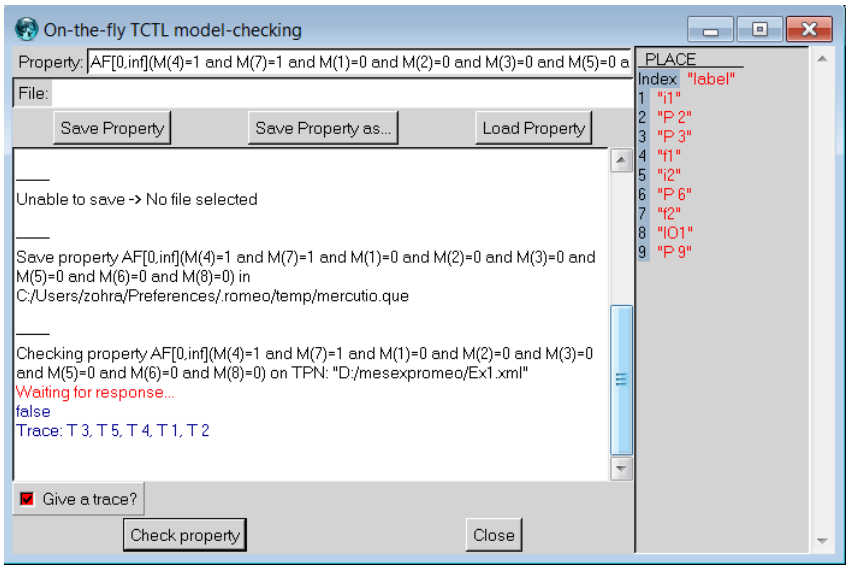


Fig. 5. Test of total compatibility

The perfect compatibility ensuring a proper termination with deadlock freeness within 210 hours is also verified for the example. However if we suggest a perfect compatibility in less than 210 hours, the result is "false".

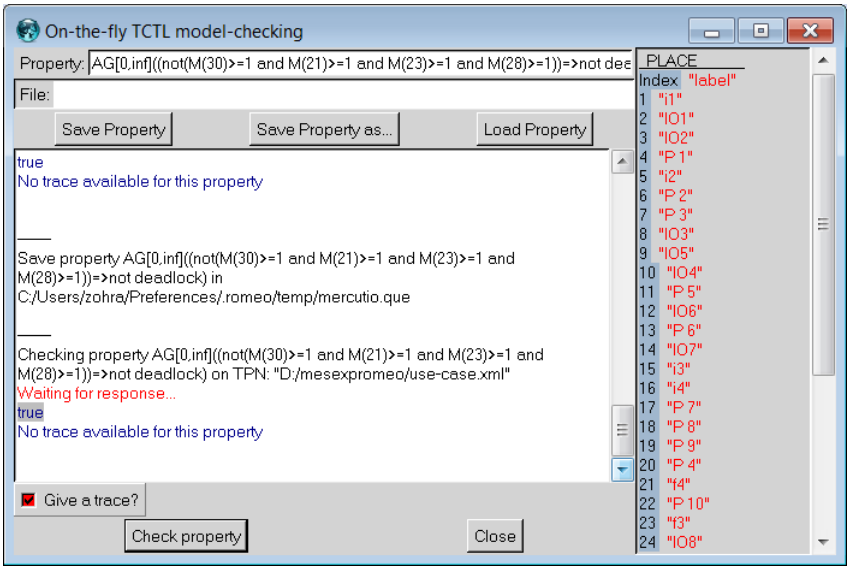


Fig. 6. Model checking the partial compatibility of the case study with Romeo

5 Related work

Several works dealt with compatibility analysis of Web services modeled either by open workflow nets or other formalisms. Wil M. P. van der Aalst and al. [4] considered that two services are compatible if their interfaces are compatible and if in addition the composition does not suffer from a deadlock. They also formalized other concepts related to the compatibility as strategy and controllability.

Lucas Bordeaux and al. [11] studied the verification of compatibility of Web services assuming that the messages exchanged are semantically of the same type and have the same name. They based their work on labeled transition systems (LTS) for the modeling of Web services. Three types of compatibility have been defined: the opposite behavior, unspecified reception and absence of deadlock.

Marlon Dumas and al. [17] have classified the incompatibility of Web services into two types: 1) Incompatibility of signatures (it occurs when a service request an operation from another service which can't provide it) and 2) Protocol incompatibility which occurs when a service A engages in a series of interactions

with a service B, but the order which undertakes the service A is not compatible with the service B. hence, they focused on the incompatibility of protocols in their article.

Wei Tan and al. [34] proposed an approach that checks interface compatibility of Web services described by BPEL, and corrects these services if they are not compatible. To do this, they modeled the composition by SWF-nets, a subclass of CPN (Colored Petri Nets). Then they checked the compatibility of interfaces.

These works dealt with non timed processes while we focus on those augmented by time information. Focusing on time constraints, Nawal Guermouche and al. [23] proposed an approach that allows the automatic verification of the compatibility taking into account their operations, the messages exchanged, the data associated with messages and time constraints. To check the compatibility of services using all of these properties, they proposed to extend the Web Services Timed Transition System (WSTTS), while we chose to extend oWF-nets with delays associated to activities. In addition, none of the approaches mentioned is based on the formal verification of compatibility while we have used this method in our approach. We mainly used the model checking formal method to check the compatibility classes of ToWF-nets, witch shows a counter example in case a property is violated allowing thus to recognize and correct the eventual errors as early as possible.

6 Conclusion

Open workflow nets form a sub class of Petri nets which has been widely and successfully used to model inter organizational business processes. In particular, they successfully form a solid theoretical basis for modeling and analysis of Web services composition. From a software engineering point of view, the construction of new services by composing existing ones raises a number of challenges. The most important is the challenge to guarantee a correct interaction of independent, communicating pieces of software. In deed, due to the message sending nature of service interaction, many delicate errors might take place when several services are put together (unreceived messages, deadlocks, contradictory behaviors, etc.). So far, it is necessary to ensure the proper functioning of each service involved in the composition as well as their ability to be composed, their good communication and the validity of their messages exchange.

In this context, we investigated in this paper the verification of open workflow nets compatibility as a main feature to ensure a correct composition and to prevent eventual errors from occurring. In addition, we extended the oWF-nets by timing constraints specifying the activities delays. For the proposed model baptised Time oWF-net, we studied its semantics in terms of states evolution. Then, we defined compatibility classes relative to ToWF-nets and emphasized a formal method of their verification based on TCTL model checking. We finally studied a case study in which four services interact with each other to reach a common goal which is the awarding of pensions of handicapped persons. We conducted a reachability analysis of this example in conformance with the method

we propose and we model checked some of the proposed properties with the time Petri net analyser Romeo. We presented, in addition, a simple example with a violated property in order to show the generation of a counter example.

As a perspective, we propose to study the parametric verification of ToWF-nets. In deed, this supposes to treat ToWF-nets modeling concurrent instances and thus the consistency of time properties is of great interest.

References

1. van der Aalst, W.: Interval timed coloured petri nets and their analysis. In: Proceedings of the 14th International Conference on Application and Theory of Petri Nets, London, Springer-Verlag. pp. 453–472 (1993)
2. van der Aalst, W.: Three good reasons for using a petri-net-based workflow management system. In: International Working Conference on Information and Process Integration in Enterprises (IPIC96). pp. 179–201 (1996)
3. van der Aalst, W.: Verification of workflow nets. In: ICATPN 97, LNCS, 1248 (1997)
4. van der Aalst, W., Arjan, J., Christian, S., Wolf, K.: Service interaction: Patterns, formalization, and analysis. In: 9th International School on Formal Methods for the design of Computer, Communication and Software Systems (2009)
5. Alur, R., Courchoubetis, C., Dill, D.: Model checking in dense real time. Information and computation. 104, 2–34 (1993)
6. Atluri, V., Huang, W.: An authorization model for workflows. In: Proceedings of the 4th European Symposium on Research in Computer Security, London, Springer-Verlag. pp. 44–64 (1996)
7. Barkaoui, K., Ben Ayed, R.: Uniform verification of workflow soundness. Transactions of the Institute of Measurement and Control Journal. 31, 1–16 (2010)
8. Barkaoui, K., Ben Ayed, R., Sbaï, Z.: Workflow soundness verification based on structure theory of petri nets. International Journal of Computing and Information Sciences (IJCIS). 5(1), 51–61 (2007)
9. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time petri nets. IEEE Transactions on Software Engineering. 17(3) (1991)
10. Berthomieu, B., Vernadat, F.: State class constructions for branching analysis of time petri nets. In: TACAS 2003, volume 2619 of Lecture Notes in Computer Science. pp. 442–457 (2003)
11. Bordeaux, L., Salaun, G., Berardi, D., Mecella, M.: When are two web services compatible ? Sapienza University. 3324 (2005)
12. Boucheneb, H., Barkaoui, K.: Parametric verification of time workflow nets. In: 24th International Conference on Software Engineering (SEKE). pp. 375–380 (2012)
13. Boucheneb, H., Barkaoui, K.: Reducing interleaving semantics redundancy in reachability analysis of time petri nets. ACM Transactions in Embedded Computing Systems (TECS). 12(1), 1–24 (2013)
14. Boucheneb, H., Gardey, G., Roux, O.: Tctl model-checking of time petri nets. Journal of Logic and Computation. 19(6), 1509–1540 (2009)
15. Camerzan, I.: On soundness for time workflow nets. Computer Science Journal of Moldova. 15(1), 74–87 (2007)
16. De Michelis, G., Ellis, C., Memmi, G.: In: Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms, Zaragoza, Spain (1994)

17. Dumas, M., Benatallah, B., Motahari Nezhad, H.: Web service protocols : Compatibility and adaptation. Institute of Electrical and Electronics Engineers. (2008)
18. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of conference on Organizational computing systems. pp. 10–21 (1995)
19. Esparza, J., Silva, M.: Circuits, handles, bridges and nets. In: Applications and Theory of Petri Nets. Lecture Notes in Computer Science, vol. 483, pp. 210–242. Springer (1989)
20. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Compatibility verification for web service choreography. In: Proceedings of IEEE International Conference on Web Services. pp. 738–741 (2004)
21. Gardey, G., Lime, D., Magnin, M., Roux, O.: Romeo: A tool for time petri nets analysis. In: Proceeding of 17th International Conference on Computer Aided Verfication (CAV'05), volume 3576 of Lecture Notes in Computer Science. pp. 418–423 (2005)
22. Gou, H., Huang, B., Liu, W., Li, Y., Ren, S.: Modeling distributed business processes of virtual enterprises based on the object-oriented approach and petri nets. Systems Man and Cybernetics. 3 (2001)
23. Guermouche, N., Perrin, O., Ringeissen, C.: Timed specification for web services compatibility analysis. Theoretical Computer Science. (2008)
24. Hadjidj, R., Boucheneb, H.: On-the-fly tctl model-checking for time petri nets. Theoretical Computer Science. 410(42), 4241–4261 (2009)
25. Karsten, S.: Controllability of open workflow nets. In: EMISA. LNI, Bonner Köllen Verlag. pp. 236–249 (2005)
26. Konur, S., Fisher, M., Schewe, S.: Combined model checking for temporal, probabilistic, and real-time logics. Theoretical Computer Science. 503, 61–88 (2013)
27. Ling, S., Schmidt, H.: Time petri nets for workflow modelling and analysis. In: IEEE International Conference on Systems, Man, and Cybernetics. pp. 3039–3044 (2000)
28. Martens, A.: On compatibility of web services. In: Petri Net Newsletter. pp. 12–20 (2003)
29. Martens, A.: Analyzing web service based business processes. In: Proceeding of International Conference on Fundamental Approaches to Software Engineering, Part of the European Joint Conferences on Theory and Practice of Software, Lecture Notes in Computer Science vol. 3442, Springer-Verlag, (2005)
30. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the soa. Annals of Mathematics, Computing and Teleinformatics 1(3), 35–43 (2005)
31. Merlin, P.M.: A study of the recoverability of computing systems. University of California (1974)
32. Ridi, L., Torrini, J., Vicario, E.: Developing a scheduler with difference-bound matrices and the floyd-warshall algorithm. IEEE SOFTWARE (2012)
33. Sbaï, Z., Barkaoui, K.: Vérification formelle des processus workflow - extension aux workflows inter-organisationnels. Revue Ingénierie des Systèmes d'Information: Ingénierie des systèmes collaboratifs. 18(5), 33–57 (2013)
34. Tan, W., Fan, Y., Zhou, M.: A petri net-based method for compatibility analysis and composition of web services in business process execution language. IEEE T. Automation Science and Engineering. 6(1), 94–106 (2009)
35. WFMC: Workflow management coalition terminology and glossary (wfmc-tc-1011). Tech. Rep., Workflow Management Coalition, Brussals. (1999)
36. Yoneda, T., Ryuba, H.: Ctl model checking of time petri nets using geometric regions. IEICE Transactions on Information and Systems. pp. 297–396 (1998)

Petra: A Tool for Analysing a Process Family

D.M.M. Schunselaar*, H.M.W. Verbeek*, W.M.P. van der Aalst*, and H.A. Reijers*

Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{d.m.m.schunselaar, h.m.w.verbeek, w.m.p.v.d.aalst, h.a.reijers}@tue.nl

Abstract. When looking for the best model, simulation is often used for "what-if" analysis. The properties of a modelled process are explored by repeatedly executing the model. Based on the outcomes, parts of the model may be manually modified to improve the process. This is iteratively done to obtain the model best suited to a user's requirements. In most cases, this is a labour-intensive and time-consuming task. To improve on the state of the art, we propose a framework where the user defines a *space of possible process models* with the use of a configurable process model. A configurable process model is a compact representation of a family of process models, i.e., a set of process models related to each other. Within the framework, different tools can be used to automatically compute characteristics of a model. We show that, when used on data from multiple real-life models, our framework can find better models in an automated way.

1 Introduction

Within the CoSeLoG project, we are cooperating with 10 Dutch municipalities. Each of these municipalities has to provide the same set of services to their citizens, but each may do so in its own way [1]. This "couleur locale" justifies that there may be different solutions to realise a particular process. The union of local variations for these services spans a solution space containing the process models currently selected by municipalities (and possibly more). When a municipality wants to improve their process model, they can use the solution space to find a better solution (process model) to replace their process model. In Fig. 1, we illustrate the solution space spanned by interpolating between the different process models from the municipalities A, B, and C. In [1], we have shown how to obtain this solution space. This initial solution space lacks the explicit information which of the models is the most desired one for an organisation (in Fig. 1 indicated by "?/?"). In this paper, we present *Petra*: A generic tool to explore a space of possible models by repeatedly analysing elements of the space and supporting the collection and comparison of analysis results. Using *Petra*, we transform the initial solution space to a solution space with explicit information

* This research has been carried out as part of the Configurable Services for Local Governments (CoSeLoG) project (http://www.win.tue.nl/coselog/).

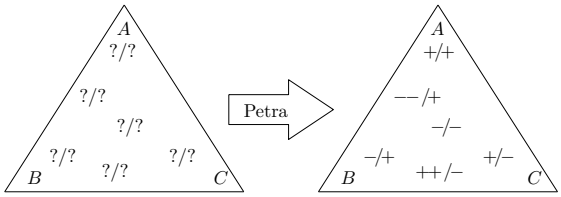


Fig. 1: Given a space of models spanned by a family of process models, we can compute characteristics (2 in this example) for all the process models. These characteristics can be used to find the "best" model.

about various aspects of the model, e.g., sojourn time, cost, and complexity (in Fig. 1 indicated with "+/-", "+/+", etc.).

Some approaches exist for the problem at hand. Unfortunately, these approaches are either only existent on paper, or they are limited to a single tool capable of computing a predetermined set of characteristics. Therefore, we introduce *Petra*[1], which stands for "Process model based Extensible Toolset for Redesign and Analysis". *Petra* is a generic and extensible framework. The solution space it operates on is defined using configurable process models. The values obtained from different analysis techniques are stored using *properties*. By providing a generic interface, any analysis tool can be used in our framework by implementing this interface.

Configurable process models are a compact representation for a family of process models, i.e., a set of process models which are related to each other. Different members of the family can be obtained by selecting different elements (from a predetermined set of elements) to be removed. This selection is called a configuration. This notion of configurable process models subsumes adjusting the model with the use of XOR/OR gateways. For our configurable process models, we use block-structured process models called Process Trees [2], which are a generalisation of the formalism presented in [1]. Process Trees are specifically developed within our project and all developed analysis techniques have been designed for Process Trees.

The executable model obtained after configuring the configurable process model is a model from the aforementioned solution space which one would like to analyse. Using the aforementioned properties, we can annotate a process model with analysis results. Properties are generic and can encode any analysis result obtainable, e.g., sojourn time, costs, etc. Apart from analysis results, properties are also used to encode run-time characteristics of a process model, e.g., the arrival process of cases, work schedules of resources, and variable working speeds of resources.

Since any analysis result can be encoded and we do not want to limit the analysis power of *Petra*, we provide a generic tool interface. On this interface, we provide a configured process model with relevant properties. The analysis

[1] *Petra* is implemented as a ProM plugin http://www.processmining.org/

tools interacting with *Petra* are expected to, if required, make a transformation from our Process Trees to the tool specific formalism. Conversions already exist from Process Trees to classical Petri nets, BPMN, and YAWL for structural analysis. When the tool finishes its analysis, we expect it to return a Process Tree annotated with analysis results. We have used CPN Tools [3] as a first example of an analysis tool in the context of *Petra*. We have selected CPN Tools for 3 reasons; (1) it provides flexibility in defining the to-be-computed characteristics, (2) by using Coloured Petri Nets, we can easily encode the more advanced insights in resource behaviour, and (3) through Access/CPN [4], it is possible to simulate all the models without human involvement.

Apart from providing the *Petra* framework and its implementation, we also conducted a case study in which we applied our tool to the models from the municipalities involved in the CoSeLoG project. In this case study, we have taken models from two municipalities and transformed these into a configurable process model. For each of the configured models in the solution space, we automatically created a CPN model and simulated the CPN model to obtain time-related information. Afterwards, we have enriched the analysed models with the output of the simulation. This case study shows that our simulation models approach reality and that we can find better models using *Petra*.

The remainder of this paper is organised as follows: Sect. 2 presents related work. In Sect. 3, we present our high-level architecture, Process Trees, and the properties currently used within our framework. The transformation of a Process Tree with properties to a CPN model is presented in Sect. 4. After presenting the key concepts of our framework, we use a case study to demonstrate the applicability of *Petra* (Sect. 5). Finally, in Sect. 6, we present our conclusions and sketch venues for extensions and research. A technical report supporting this paper can be found in [2].

2 Related work

In this section, we will first elaborate on the techniques applicable to our problem setting. Afterwards, different techniques are discussed which reason on a configurable process model to obtain a process model most desired by the user. Finally, we briefly discuss the used formalism for *Petra*, as well as its limitations and benefits relative to other formalisms.

2.1 Applicable techniques for our problem setting

In [5], an approach is presented that is based on configurable process models. As far as we know, it is the approach most similar to ours. The configurable process models that are used by it are modelled in Protos [5]. By converting the configurable process model to a CPN model, the authors can use the same model to analyse multiple variants of the same process model. The main limitation of their approach is the fact that the focus is on the use of a single tool (CPN Tools) which results in a non-extensible set of analysis results. Furthermore, the

resource model employed is rather simplistic (see [6], for an overview on the importance of correctly modelling resources), i.e., all resources are available all of the time. There is also no support for data, i.e., exclusive choices are chosen randomly. Finally, with respect to determining soundness, state space analysis has to be employed.

Another approach related to our approach is presented in [7]. On the existing process model, process measures are computed. Afterwards, different applicable best practices are evaluated and applied to obtain new process models. Finally, on these process models, process measures are computed, the best-performing process models are kept, and the steps are repeated. In [8], an implementation is presented. However, the focus is on simulation and time-related information. Furthermore, a single tool is selected for the analysis, prohibiting to gather information not provided by this specific tool.

In [9], there is a direct dependency on process mining techniques to obtain the process model as-is, and on enrichment of the process model with information from the event log. After obtaining the enriched model, there is an iterative approach of finding the malfunctioning parts of the process model, selecting transformation from a database to be applied, generating process models for the different redesign possibilities. The generated process models are stored, and, if required, the aforementioned steps can be re-executed to change another part of the process model. From all the generated process models, the best process model is selected and returned to the user. However, it is unclear how the database of redesigns is obtained, and it has not been implemented.

2.2 Configurable process models

In [10], a questionnaire-based approach is presented to deduce on a high level the requirements a user poses on the best instantiation of a configurable process model. This, in combination with functional requirements, results in an instantiated process model closest to the user's requirements. This approach does not give any performance information, but it can be used beforehand to limit the to-be-analysed solution space.

The `Provop` framework contains context-based configuration of the configurable process model [11]. Within the `Provop` framework, there are so-called context rules which limit the configuration possibilities by including contextual information. These context rules specify, for instance, that if the user requests high quality, then certain other activities have to be included in the process model. As with the approach in [10], the focus of this approach is not on collecting performance information. Yet, it can be used to limit the solution space.

2.3 Process Trees and properties

Configurable process models have been defined for different modelling formalisms, e.g., configurable EPC [12,13], configurable YAWL [14], configurable BPEL [14], CoSeNets [1], and Process Trees [2]. The first 3 formalism are more expressive

than the latter two, i.e., the CoSeNets and Process Trees are a subclass of Free-Choice Petri nets. However, with the first 3, explicit attention has to be paid to the soundness [1] of the configured model. Furthermore, CoSeNets only focus on control flow. Therefore, we use Process Trees as our formalism.

Process Trees are related to the RPST [15] in the sense that both are block-structured. However, the RPST is used to convert non-block structured models into block-structured models and focusses on the control-flow perspective. Furthermore, RPST can have so-called rigids which are non block-structured fragments of the process model. These are not present in Process Trees.

Efforts have been made on enriching BPMN models with simulation information by the *WfMC* standard *BPSim* [16]. However, we allow, amongst others, for a richer resource model through supporting arousal-based working speeds. Furthermore, *BPSim* is tailored towards simulation and thus abstracting from non-simulation related information, e.g., compliance. We would like to be able to encode this non-simulation related information as this might be of importance for different analysis techniques. The properties are related to and inspired by *BPSim* and a transformation from properties to *BPSim* has been made which works in conjunction with the L-SIM tool from Lanner[2].

3 Petra

In this section, we show the high-level architecture of *Petra*. We will also discuss the lefthand side of Fig. 2, i.e., the process model and the properties used.

3.1 Architecture

The architecture of *Petra*, including the use of our sample, analysis tool, is depicted in Fig. 2. In *Petra*, we have a family of process models defined by a configurable process model from which we want to select the "best". This is

[2] `http://www.lanner.com/en/l-sim.cfm`

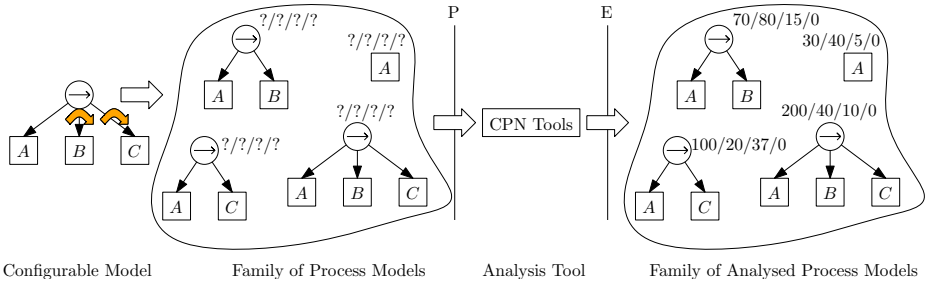


Fig. 2: In *Petra*, we start with a configurable process model which describes a family of process models. Each of these models is analysed resulting in a family of analysed process models.

subject to the exact requirements of the organisation. We extend every process model with relevant properties computed using different tools, e.g., sojourn time, queue time, costs, etc. After obtaining the relevant properties, we have a family of analysed models. At a later stage, the user can, via different views on this family of models, select the desired process model.

Petra consists of 3 key concepts: a family of process models, a set of properties, and a tool interface. We first elaborate on the properties and the tool interface. Afterwards, the Process Trees are discussed, and the family of process models last.

Properties Properties come in two different flavours; independent (facts), and dependent (deduced from facts). By not limiting the framework to a fixed set of properties, we keep our framework generic. Furthermore, properties can be part of every construct, e.g., resources, data, etc. In this paper, we focus on an often used quantitative measure namely time. But properties can also be used to encode for instance the model complexity.

Tool interface To the analysis tool employed, we offer a Process Tree annotated with properties on interface P. An analysis tool has to make a transformation from the Process Tree with properties, to the tool-specific formalism or use one of the currently available conversions, e.g., to classical Petri Nets or BPMN. Afterwards, if the tool has finished its computations, the output of the tool has to be transformed into dependent properties and the Process Tree should be enriched with this information (interface E). The properties offered on interface P can both be independent, and dependent. In our framework, we assume an incremental approach, i.e., tools may only modify or add dependent properties and are not allowed to remove (in)dependent properties. Since the amount of possible models can be exponential in the amount of configuration points, we require a tool to run automatically, i.e., without the need of human involvement. Interface P can also be used to query properties of a tool, e.g., properties the tool can compute (and for which nodes), and properties necessary for that.

3.2 Process Trees

Process Trees are block-structured process models, a subclass of Free-Choice Petri nets, in which each block has a single entry and a single exit. A Process Tree consists of 3 perspectives: control-flow, resource, and data. Next to this, we have two extra perspectives to encode contextual information namely: environment, and experiment. In Fig. 3, the different perspectives are depicted and their relation to each other. Each of the perspectives contains a set of configuration points with configuration options. In the remainder of this section, we deal with each of the perspectives and their properties, and show the configuration options.

Control-flow perspective The control-flow perspective consists of nodes, and directed edges between these nodes which denotes the order of execution. Nodes

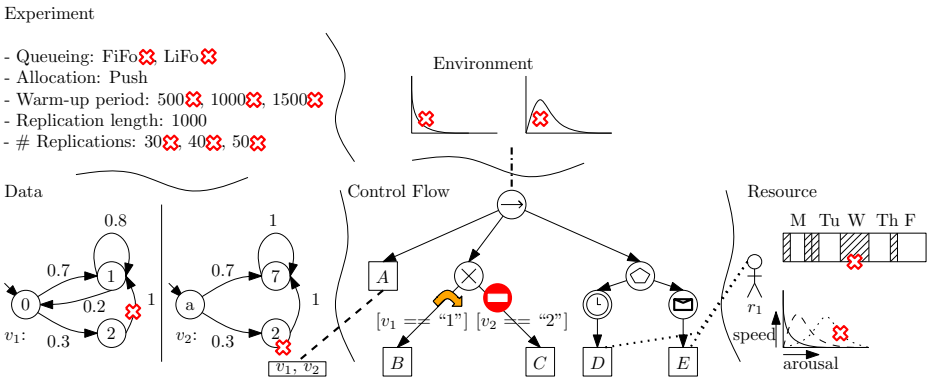


Fig. 3: The different perspectives in a Process Tree.

come in two different flavours: tasks and blocks. Tasks are the units of work which need to be executed, e.g., A in Fig. 3, and can be manual (with a resource) or automatic. Blocks specify the causal relationship between its children, e.g., the block with the arrow in Fig. 3 denotes a sequential execution. All the nodes are depicted in Fig. 4 with their Petri net semantics[3]. Note that, we use some of the notations for events in Petri nets from [17].

If there is an edge from a block to a node, then we say that this node is a child of that block. We have a total order on the outgoing edges as for most blocks the order has semantics, e.g., SEQ. In general, all nodes can have any number of children excepts for the EVENT (letter or clock), LOOPXOR, and LOOPDEF (see Fig. 4). Finally, the set of nodes and edges forms a connected Directed Acyclic Graph (DAG) with a single root node, i.e., a node without any incoming edges. Process Trees are encoded as a DAG to minimise duplication of behaviour and thus increasing maintainability.

There are three types of configuration options: (1) hiding, (2) blocking, and (3) substitution. Substitution entails the option to replace a part of the process model with a subprocess from a predetermined collection of subprocesses (see [1]). Hiding, which is shown in Fig. 5 with the curved arrow, entails the option to abstract from a part of the process model by substituting the subprocess with an automatic task, e.g., Fig. 5(b). Blocking, shown with a no-entry sign in Fig. 5, denotes the option to prevent the flow of entering a part of the process model, e.g., Fig. 5(a). Note that blocking has non-local effects, e.g., if a part of a sequence is blocked, then the entire sequence becomes blocked.

In Fig. 5, the space of models is depicted ((a)-(d)) using hiding and blocking. Figure 5(a) and (c), shows the configured model if we select blocking (note that we have removed the XOR as it is redundant), Fig. 5(b) and (c) shows the configured model if hiding is selected and finally, Fig. 5(d) shows the configured model when none of the configuration options is selected.

[3] Note that, for many nodes, Fig. 4 shows the semantics for the case with only two children. However, it is trivial to extend this to more children.

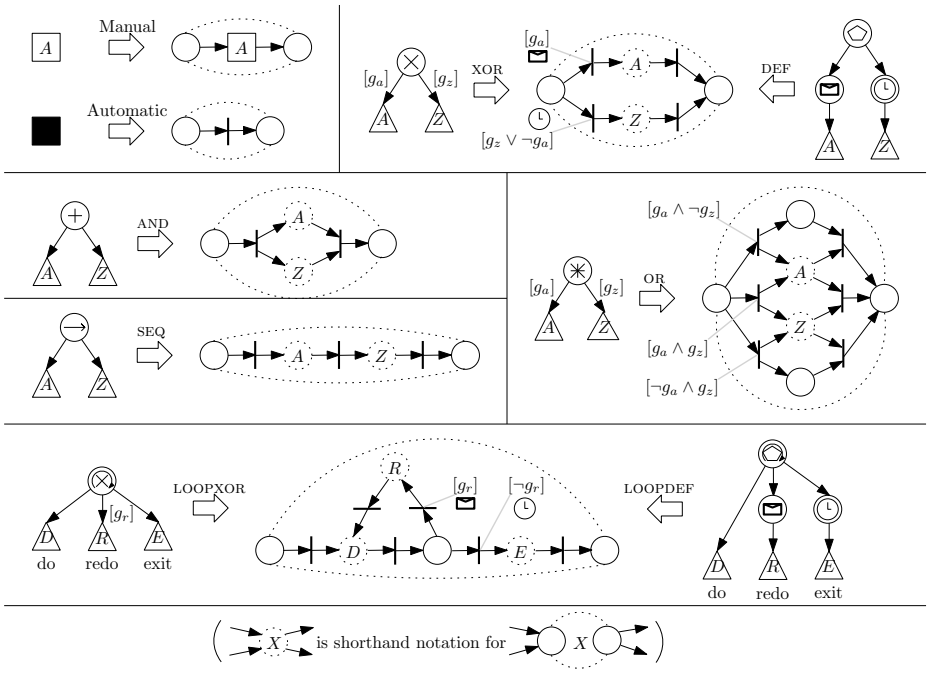


Fig. 4: The different nodes and their (abstract) Petri net semantics.

Data perspective The data perspective specifies which expressions are used for the outgoing edges of a choice (between "[" and "]"), which variables are read and written (line from A to v_1 and to v_2 in Fig. 3). Furthermore, we have variables encoded as Markov chains denoting which values a variable might take, with which probability it may take this value, and what the initial value is of a variable. For instance, in Fig. 3 (left-hand side), the variable v_1 has initial value "0", it may take the values "0", "1", and "2", and it changes value according to the probabilities on the edges.

Expressions reason over variables and values for those variables. They have the following operators: conjunction, disjunction, and negation. Furthermore, variables may be compared to values on equality and inequality.

On the outgoing edges of choice constructs, there is the option to select an expression from a set of expressions. For variables, we can change which nodes read/write this variable. Furthermore we have the following properties for variables: we have the option to change the initial values, which values may be taken, and whether an edge with a certain probability exists between two values. In Fig. 6, the different possibilities are shown for removing a potential value and removing an edge between two values for a variable. The family of data perspectives is spanned by the cartesian product of the options for the variables and expressions.

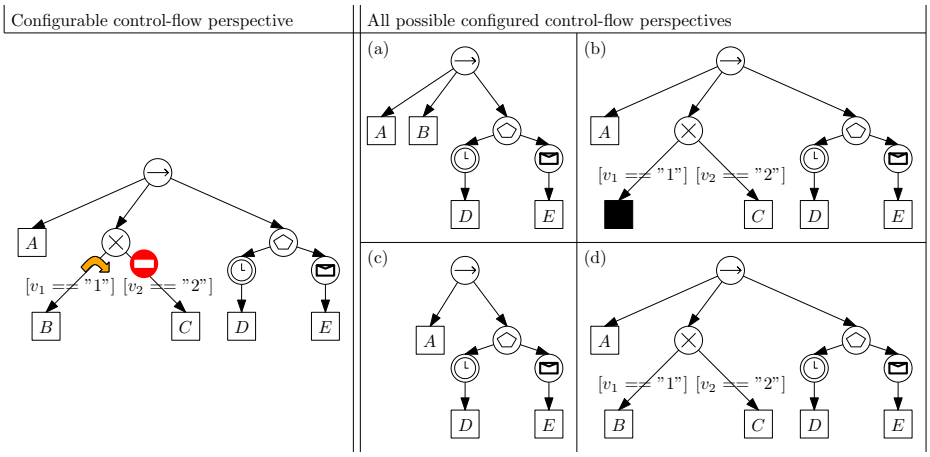


Fig. 5: An example family of control flows.

Resource perspective The resource perspective specifies which resource may execute which activity, e.g., in Fig. 3, we have a resource r_1 and she may execute D and E. Furthermore, the properties of the resource perspective specify (1) when a resource is available to work on the process (closely related and inspired by [6]), e.g., r_1 is available on Monday morning and evening, but not on Friday, and (2) how many resources there are of a particular kind. Finally, different working speeds can be specified based on the busyness of a resource, this to model effects such as the phenomenon described by the Yerkes-Dodson law of arousal [18][4]. For the work schedule, one can remove intervals in which the resource is available to the process (Fig. 7 at the top). The number of resources is selected from a predetermined set of values. Finally, the arousal based work speed offers the option to remove parts of the work speeds associated with the arousal levels (Fig. 7 at the bottom). The family of resource perspectives is spanned by the cartesian product of the options for the work schedule, arousal based work speed, and number of resources.

Environment perspective Currently, we only have the option to select the arrival process (a property) for the environment perspective. The arrival process specifies the distribution of arrivals of new cases to the process. Configuring the arrival process entails selecting a distribution describing the arrival of cases.

Experiment perspective The experiment perspective consists of the simulation property (Fig. 3 top left), which specifies the arguments for the simulation tool. We can select a queuing principle, e.g., FiFo, and whether push or pull allocation is used. Furthermore, we can set the warm-up period, replication length, and

[4] The Yerkes-Dodson law of arousal describes the correlation between the arousal (pressure) and the working speed of a person

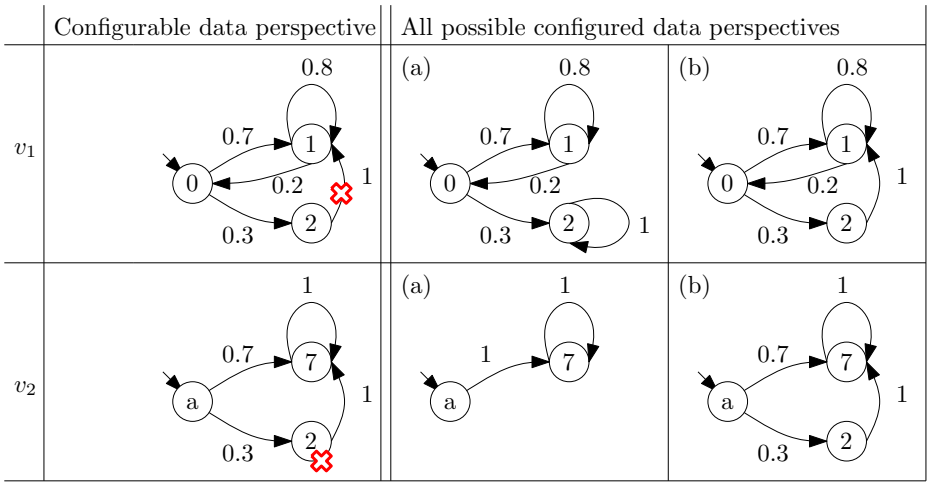


Fig. 6: Example families of Markov chains for variables v_1 and v_2.

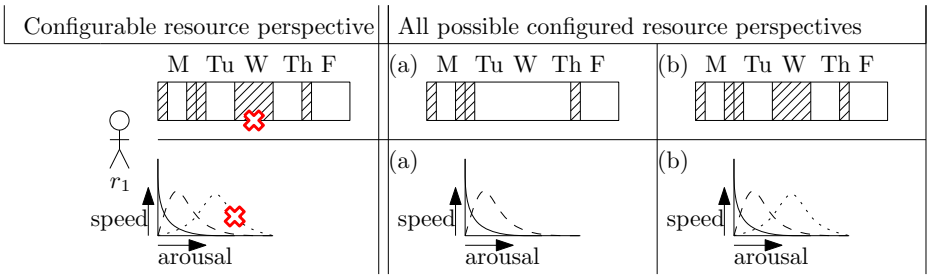


Fig. 7: Example families of work schedules and work speeds for resource r_1.

number of replications. For each of the arguments, a single value has to be selected from a predetermined set of values.

The properties inside of the experiment perspective are not entirely independent of the used tool, e.g., the simulation property uses concepts from the simulation domain. When analysis techniques from another domain is used, there should be a property with common concepts from that domain.

3.3 Solution space

The solution space of a Process Tree is spanned by the cartesian product of possible selections of configuration options for the various configuration points in the various perspectives. In case of Fig. 3, we have a total of $2304 = 2 \cdot 4 \cdot 4 \cdot 4 \cdot 18$ possible models, i.e., 2 possibilities for the environment perspective, 4 for the data perspective, 4 for the control-flow perspective, 4 for the resource perspective, and 18 for the experiment perspective. *Petra* is able to traverse this solution space automatically irrespective of the used properties.

4 Sample tool: CPN Tools

In this section, we elaborate on the transformation of a configured Process Tree to a CPN model. Since the implementation for controlling CPN Tools using Access/CPN and parsing the output is relatively straightforward [2], we do not elaborate on it here.

4.1 Transforming a Process Tree to a CPN model

Transforming a Process Tree to a CPN model is done along four perspectives: control-flow, data, resources, and environment; the experiment perspective consists mainly of controlling CPN Tools itself. Each of the four perspectives is treated separately. Modifications that only impact on a single perspective also only modify the CPN model in that perspective. Between different perspectives, we have introduced the notion of a controller such that the perspectives can communicate with each other. For example, the control-flow perspective uses the allocation controller to signal that a resource is necessary.

Data perspective The transformation of the expressions in the Process Tree is straightforward, i.e., we have a variable (v_{expr}) storing whether an expression evaluated to true of false. A transition guarded with the expression itself is in charge of updating v_{expr} based on the values of the variables in the expression and based on the previous version of the expression.

Modelling the variables themselves is a bit more involved. When a new case is started, the variable is initialised with the initial value. If a variable is written, we determine the new value of a variable. Most notable for the variable is the fact that we employ two different views on the variables: a control-flow view, and a guard view. We made the distinction because the different views serve two different purposes. For the control-flow view, it is irrelevant what the value of a variable is but it is important to know whether an update has been performed. For the guard view, it is important to know the exact value of a variable as these are used to evaluate guards.

Resource perspective The resource perspective mainly comprises of the work schedules of the resources. The number of resources available to the process is a modification of the initial marking. The task controller computing the resource arousal level and thus the duration of a task is discussed at the control-flow perspective (the computation is in the task controller).

A work schedule is a list of reals denoting the length of the intervals of being present and absent. An integer denotes at which index the schedule currently is, and a timestamp denotes at what time the last change took place. For instance, if we have `(2, [11.3, 4.1, 7, 4], 15.4)`, the "2" denotes that we are at index 2, i.e., the value "7", the values between "[" and "]" are the encoding of the work schedule (available, unavailable), and the "15.4" at the end denotes the timestamp of the last change. Note that since CPN Tools does not allow

guards based on time[5], we cannot directly use the time of the last change added to the duration of the interval as a guard for a transition making a resource (un)available to the process.

Control-flow perspective Within the control-flow perspective, certain parts of the process model do not always contribute to a case. For instance, for an OR construct only the selected branches (at run-time) contribute. To prevent the explicit modelling of all possible paths through the OR, we have introduced the notion of true/false tokens. If the true/false token is true, then that part of the process contributes to the case; otherwise it does not [19].

To be able to cope with multiple cases, we add to each token the notion of a case identifier. Furthermore, this means that transitions for synchronising a subprocess, e.g., the parallel execution of a number of tasks, are now guarded with the requirement that they can only synchronise if all the tokens have the same identifier. With the aforementioned directed acyclic graph of the Process Tree, we might have tokens with the same identifier but belonging to different instantiations of the subprocess. To solve this issue, we unfold the directed acyclic graph of the Process Tree into a tree prior to applying the transformation.

In order to obtain timing information from the simulation, we have extended every token with a timestamp, duration, and performance information about the sojourn time, processing time, queue time, and wait time. The timestamp denotes when a token entered a subprocess rooted at the node. The duration denotes the processing time for a task and is updated just before the tasks starts. The duration of a task is computed by the controller of a task since the duration is based on the task and the arousal level of the allocated resource. Finally, the performance information is used to obtain information about subprocesses and can be used to compute the performance characteristics of a node based on its children. The sojourn time is the time from the start of a subprocess for a case until the end of that subprocess, processing time is the total amount of time resources spent working on a case within a subprocess, queue time is the total amount of time a case waited for a resource in a subprocess, and wait time is the time spend in synchronising parallel branches for a subprocess.

Nodes All nodes offer the same interface places to their parent in the CPN encoding. The interface places offered to the parent are: *scheduled*, *started*, *withdrawn*, and *completed*. Scheduled denotes that a subprocess may start. Started denotes that a subprocess actually has started. Withdrawn is specifically for accommodating events, in which case all events are scheduled at the same time and the event that starts first withdraws the others. Finally, completed denotes that the subprocess has finished its execution. The interface places are encoded using fusion places in CPN Tools [3].

Connected to the interface places are some standard transitions. The *withdraw* transition fires when a token is received on interface place withdrawn, and

[5] Guards are only reevaluated when tokens are added/removed in the surrounding places.

we have not started the execution of this subprocess. If the subprocess has already been started, then the withdrawn token is forwarded with high priority to the children. By using priorities on the transitions, we guarantee that nothing ordinary can happen between obtaining and forwarding the withdrawn token. When a subprocess is scheduled with a true token, then first the *init* transition is fired. The init transition is linked to another fusion place which forms a hook for a controller to notice that a subprocess has been scheduled. Similar, we have a *final* transition to consume the token after a controller has been notified on the completion of this subprocess. Finally, there is a *skip* transition in case a false token is received in the scheduled place.

Tasks As tasks do not have children, we cannot receive a withdraw token after starting the task, i.e., we can only start the task after all unprocessed withdraw tokens have been processed. In case we receive a true token, the task signals the controller of this task being scheduled. As soon as the task is allowed to start, a token is produced signalling that the initialisation has been finished, and the task starts and signals its parent by producing a token on the started interface place. Afterwards, the task signals the task controller that it needs a resource and a duration for its work item. The task controller adds the work item to the list of currently unallocated work items. As soon as a resource has been allocated to a work item (by the *allocation controller*, which we explain later on), the task controller notifies the task and determines the processing time for the task based on the arousal level of the allocated resource. After the duration has elapsed, a token is made available and the task can complete. During the completion of a task, the used resource is released, the read and written variables are updated, and the statistics of the task are computed.

The statistics for a task are computed as follows: The queue time is the elapsed time between a token entering the task's subprocess and the moment a resource has been allocated to the work item. The processing time is the duration of a task. The wait time for a task in isolation is 0 by definition. Finally, the sojourn time is the sum of the queue time and the processing time, i.e., the total time spent in the task subprocess.

Blocks Blocks have the same set of places exposed to their parent as a task has, e.g., a block is also scheduled, and a block signals her parent that she has completed. Dependent on the type of block, different children receive a true token based on different information, e.g., in case of an AND block all the children receive a true token, in case of a XOR block only the first child for which the expression evaluates to true receives a true token, etc.

Due to space limitations, we show the transformation of the XOR block and the LOOPXOR block. The reason for this is that the AND, OR, and DEF blocks can be easily inferred from the XOR block. In turn, the SEQ and LOOPDEF blocks can be easily inferred from the LOOPXOR block. For the encoding of the EVENT block, we refer the reader to [2].

The most interesting part of the XOR is in the selection of which children should receive a true token and which should receive a false token. The relevant

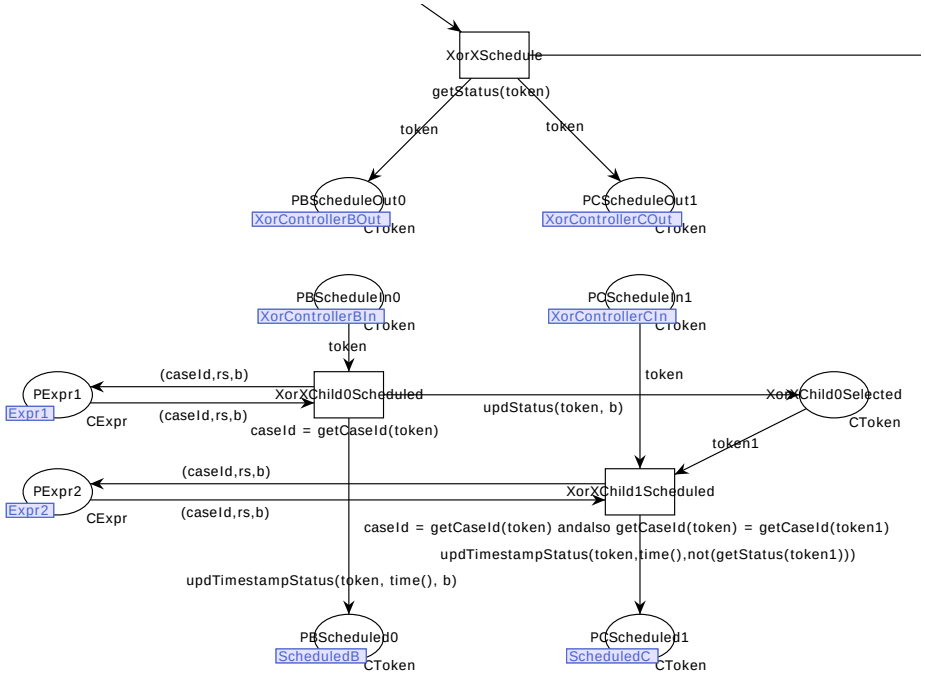


Fig. 8: Fragment in XOR dealing with guards.

part is depicted in Fig. 8. After the XOR has been initialised and scheduled, the XOR signals the fragments corresponding to the guards in the data perspective that it is going to schedule its children. After these fragments are done, the expressions are evaluated in the order of the outgoing edges. If an expression evaluates to true, then we forward a true token to the corresponding child and all other children obtain a false token. We have a special case when all the expressions evaluate to false. In that case, we forward, similar to YAWL [20], a true token to the last child (instead of a false token).

For the statistics for the XOR, we take the statistic of the child which received a true token, i.e., we do not want non-executed subprocesses to interfere with the statistics.

For the LOOPXOR, the most interesting part is the iteration of the loop (Fig. 9). If a loop is scheduled, we first schedule the *do* child of the loop. When the *do* child has completed, we schedule both the *redo* child and the *exit* child. Based on the guard of the *redo* child, we either send the true token to the *redo* child or the *exit* child, the other will be send a false token. Note that we employ a similar strategy as for the XOR, i.e., if all guards are false then the *exit* child receives a true token hence we only need to evaluate the guard of the *redo* child. If the *redo* child has received the true token, we execute the *do* child again. If the *exit* child received the true token, then we can exit the loop.

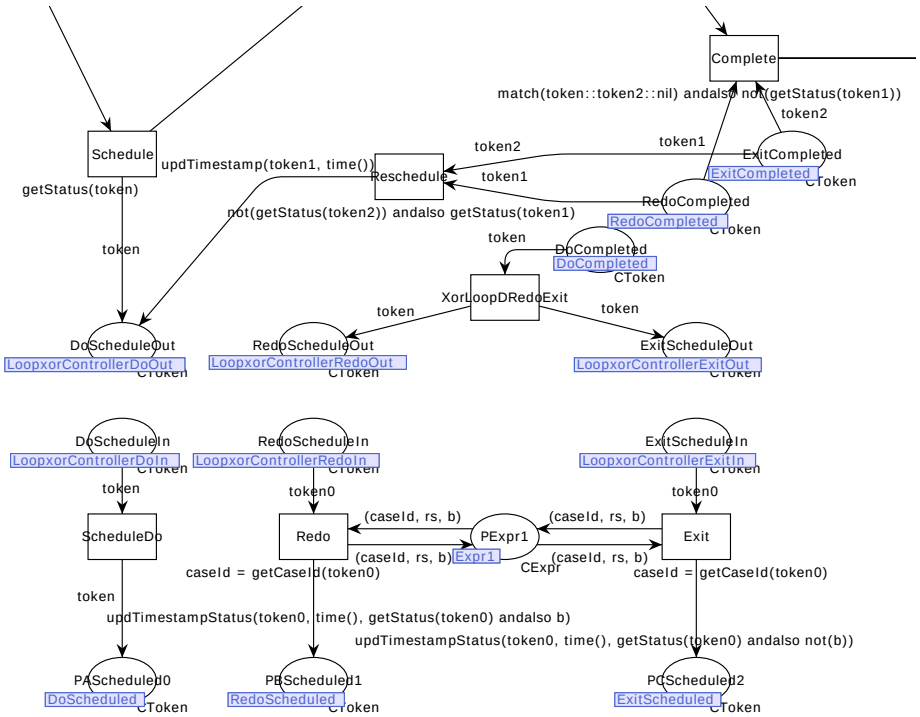


Fig. 9: Fragment in LOOPXOR dealing with guards.

The statistics of the LOOPXOR are stored in the token of the *exit* child. In the tasks, we already increment the values for the different performance measures and due to the fact that the children of the LOOPXOR are sequential, e.g., *do*, *redo*, *do*, and *exit*, there is no need to aggregate the values.

Environment Perspective The environment controller currently consists of the arrival controller only. The arrival controller realises a token generator and a token de-generator. The token generator generates new cases based on the distribution of the arrival process. The token de-generator removes finished cases from the model.

Extra Controllers A controller not belonging exclusively to any of the aforementioned perspectives is the *allocation controller*. The allocation controller allocates work items to resources. It operates on the global list of work items, which contains all cases waiting to be processed by an activity, and on the list of currently available resources to the process. The allocation can either push or pull items by taking either the work item or resource point of view. For instance, if we take the resource point of view (pull), then the resource selects the work

item most desired by her. When taking the work item point of view (push), we select the resource best fitting the work item.

5 Case study

We have taken the building permit process of two Dutch municipalities, M_A and M_B, participating in the CoSeLoG project. Instead of simulating the entire building permit process, we focus in particular on the "Bezwaar en Beroep" (*objection and appeal*) subprocess[6]. This subprocess allows people to object and appeal to the possible conferment[7] of a building permit, and starts after the municipality has published the disposal[8] of a building permit, which gives people the opportunity to object or appeal before the disposal becomes a conferment. The disposal of a building permit means that the municipality agrees with the building permit but it is not yet definitive, i.e., based on objections and appeals the municipality may decide to disagree with the building permit.

In our case study, we want to show *Petra* can indeed analyse a family of process models and find a better process model. From the municipality event logs, we obtained the different perspectives for each of the municipalities. These perspectives have been combined into a Process Tree using the aforementioned configuration options. In order to verify that the perspectives were obtained correctly, we first try to reproduce the behaviour recorded in the original logs from the municipalities. This way we can verify that the perspectives are encoded correctly.

Verifying perspectives The characteristics of the logs are listed in Table 1. The log spans a time period of roughly 2 years. The log consists of *create* events and of *completed* events, and every event has an executor associated with it. The occurrences of the different event classes varies significantly. The least occurring event class occurs just once, while the most occurring event class occurs 262 times for M_A and 451 times for M_B. We only estimate processing times of activities with at least 3 observations in the log, this to have somewhat reliable values for the sojourn time of events whilst not disposing too many activities.

From the event logs, we have constructed Process Trees only consisting of the control-flow perspectives. Since all the choices in the Process Tree are binary, we use variables which with a certain probability are "0" or "1" denoting left and right respectively.

Apart from the flow of the different cases, we also need to estimate the resource behaviour. First, we estimated the work schedule of the resources. The work schedule is estimated based on observations from the log. We have taken the timestamps of all events and made the following assumptions; (1) people work in shifts, (2) there is a morning shift and an afternoon shift, (3) as there

[6] The case study data can be found at: https://svn.win.tue.nl/trac/prom/browser/Documentation/Petra

[7] Conferment means that a building permit is granted to the requester.

[8] Disposal means that a building permit is about to be granted to the requester.

Table 1: The characteristics of the logs used in the case study

Characteristics	Cases	Events	Event classes	Resources
M_A	302	586	15	5
M_B	481	845	23	4

is no clear signal of breaks, we assume the shifts are consecutive, (4) if we have at least one measurement in a shift, we assume that person is available for the process during that whole shift, and finally, (5) we assume a weekly iteration, e.g., if someone worked on a Monday, we assume she can work every Monday.

For the throughput times of the activities, we have used two sources of information. First, by using alignments [21], we aligned the log to the earlier obtained control-flow perspective in order to see where time is spent. Second, there is legislation specifying legal time intervals, e.g., there is a legal limit within which the municipality has to respond. Unfortunately, the law allows for exceptions which are not observable in the log. To take the legal constraints into account, we have explicitly added activities to model these time intervals, and where the law is unclear, we have estimated the time interval. We have also analysed the resource performance between resources when they are allowed to executed the same activity. If there was a significant difference, we have encoded the sojourn time per resource, else we estimated the same sojourn time for all the resources. Furthermore, we have abstracted from outliers.

Using the aforementioned information, we were able to construct the simulation models for both M_A and M_B. In the simulation, we used push allocation, FiFo queues for the work items, a warmup period and replication length of $150,000$ steps in the simulator, and, we generated 30 replications for each.

Prior to comparing the simulation results to the logs, we first have to compute the statistics of the log. Since the log can be seen as a single long replication, we used batch means to be able to compute replications and to be able to compare the log to the simulation results. See [22] for an overview of the batch means method. Fig. 10 shows the results, where M_A is shown left and M_B is shown right and times on the y-axis are in hours.

As one can clearly see, there is overlap in the box plots of the logs and the simulations. Hence, we cannot conclude that our simulation model and reality differ in an unacceptable manner.

Finding a better model After illustrating that the models used in *Petra* can indeed mimic reality, we now combine the models from M_A and M_B into a single Process Tree. This Process Tree is the union of M_A and M_B but allows for more than just M_A and M_B. We want to use this combined Process Tree to improve the sojourn time for M_A by letting *Petra* automatically traverse the family of process models.

Not all combinations make sense, e.g., M_A will not hire the employees of M_B. Hence, we limit the configurability of the Process Tree. This means that we take the characteristics of the employees of M_B into account, e.g., work speed,

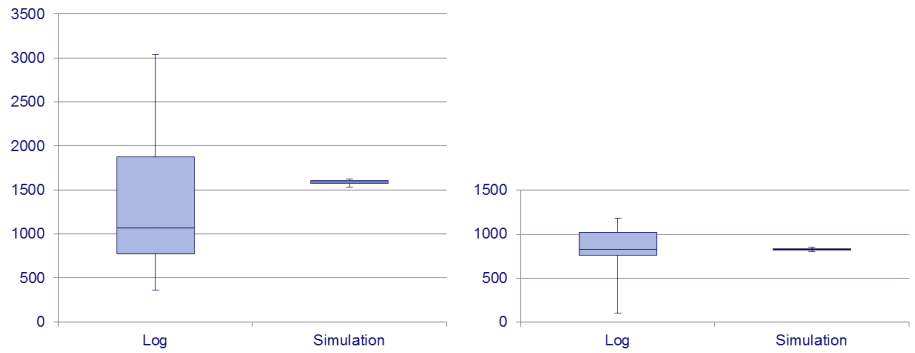


Fig. 10: Validation results.

Table 2: Case study results.

	1	2	3	4	5	6	7	8
540	A	A	A	B	A	B	B	B
630	A	A	B	A	B	A	B	B
770	B	A	A	B	B	A	B	A
Avg	1488.78	1572.48	1570.91	806.60	1496.11	1217.32	803.67	1224.50
Std. dev.	29.55	33.82	18.58	22.93	24.81	34.80	20.24	33.10

but do not put the employees of M_B into the Process Tree. Furthermore, since the building permit process is heavily subjective to legal requirements, there is not much room to change what is done but there is room to change how it is done. Therefore, we focus on the activities *540 Objection to disposal submitted*, *630 Appeal set*, and *770 Establish decision phase original decree*, since these activities embody part of the significant difference between M_A and M_B, and are related to how things are done.

With the focus on the activities *540*, *630*, and *770*, we have a solution space of 8 models. After *Petra* traversed through the family of process models, transformed these models to CPN models, simulated each of them, and enriched the family of process models, we obtain the results (throughput times of the entire process in hours) as in Table 2.

As one can see, working on the activities *540* and *770* in the same way as M_B can already significantly decrease the sojourn time for M_A. Furthermore, one can clearly see that changing *770* and *630* without changing *540* will not yield any significant improvements. Finally, *630* does not have any significant impact on the performance of the process.

6 Conclusion and future work

Using *Petra*, we can take a family of process models, as captured by a Process Tree, traverse this family, and enrich every process model from this family with

KPIs using external analysis tool like CPN Tools. Based on these KPI values, a process owner can then decide which process model suits her best.

Petra is generic and extensible. The genericity is achieved by not limiting our set of performance indicators and set of properties. The extensibility is achieved by allowing any tool to be used. Currently, we have two tools in our framework: CPN Tools and L-SIM. Finally, the implementation of our framework has been applied in a case study using the data from two municipalities resulting in an improved model. The case study shows that it is indeed possible to obtain an improved model. A technical report supporting this paper can be found at [2].

Our framework can be extended in various of directions. Currently, the simulation of each of the CPN models is a time-consuming task. Although *Petra* is multi-threaded, in the experiments, simulating a single CPN model took around 3 days on a core of 2.80 GHz. Since simulation takes such a long time, we would like to minimise the amount of Process Trees to be analysed. In our current implementation, we naively iterate through all possible instantiations of the configurable process model. This means that equivalent models, obtained by different configurations, are analysed multiple times. Defining equivalence classes on configurations and taking these into account in the iteration through the Process Trees could already result in a significant speed up. Another approach to minimising the amount of to-be-analysed models, is to have knowledge beforehand on the performance measures a user wants to optimise. To exploit this knowledge, we also want to have different (faster but imprecise) analysis tools in our framework. This way it would be possible to obtain quick estimates in the relevant measure to decide whether simulating the Process Tree will be worthwhile. Also, simulations could be aborted once it is clear that the model under investigation will never be Pareto optimal. With the use of *Petra*, these things can be easily incorporated without changing any of the aforementioned.

References

1. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P. van der, Reijers, H.A.: Creating Sound and Reversible Configurable Process Models Using CoSeNets. In Abramowicz, W., Kriksciuniene, D., Sakalauskas, V., eds.: BIS. Volume 117 of Lecture Notes in Business Information Processing., Springer (2012) 24–35
2. Schunselaar, D.M.M., Verbeek, H.M.W., Aalst, W.M.P. van der, Reijers, H.A.: Petra: Process model based Extensible Toolset for Redesign and Analysis. Technical Report BPM Center Report BPM-14-01, BPMcenter.org (2014)
3. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer (2009)
4. Westergaard, M.: Access/CPN 2.0: A High-Level Interface to Coloured Petri Net Models. In Kristensen, L.M., Petrucci, L., eds.: Petri Nets. Volume 6709 of Lecture Notes in Computer Science., Springer (2011) 328–337
5. Gottschalk, F., Aalst, W. M. P. van der, Jansen-Vullers, M.H., Verbeek, H.M.W.: Protos2CPN: Using Colored Petri Nets for Configuring and Testing Business Processes. International Journal on Software Tools for Technology Transfer **10**(1) (2008) 95–110

6. Aalst, W.M.P. van der, Nakatumba, J., Rozinat, A., Russell, N.: Business Process Simulation. In Brocke, J., Rosemann, M., eds.: Handbook on Business Process Management 1. International Handbooks on Information Systems. Springer Berlin Heidelberg (2010) 313–338
7. Netjes, M., Mansar, S.L., Reijers, H.A., Aalst, W.M.P. van der: Performing Business Process Redesign with Best Practices: An Evolutionary Approach. In Filipe, J., Cordeiro, J., Cardoso, J., eds.: ICEIS (Selected Papers). Volume 12 of Lecture Notes in Business Information Processing., Springer (2007) 199–211
8. Netjes, M., Reijers, H.A., Aalst, W.M.P. van der: The PrICE Tool Kit: Tool Support for Process Improvement. (2010)
9. Essam, M.M., Mansar, S.L.: Towards a Software Framework for Automatic Business Process Redesign. ACEEE International Journal on Communication **2**(1) (March 2011) 6
10. La Rosa, M., Lux, J., Seidel, S., Dumas, M., Hofstede, A.H.M. ter: Questionnaire-driven Configuration of Reference Process Models. Advanced Information Systems Engineering **4495** (2007) 424–438
11. Hallerbach, A., Bauer, T., Reichert, M.: Capturing Variability in Business Process Models: The Provop Approach. Journal of Software Maintenance and Evolution: Research and Practice **22**(6-7) (November 2010) 519–546
12. Rosemann, M., Aalst, W.M.P. van der: A Configurable Reference Modelling Language. Information Systems **32**(1) (2007) 1–23
13. La Rosa, M., Dumas, M., Hofstede, A.H.M. ter, Mendling, J.: Configurable multi-perspective business process models. Inf. Syst. **36**(2) (2011) 313–340
14. Gottschalk, F.: Configurable Process Models. PhD thesis, Eindhoven University of Technology, The Netherlands (2009)
15. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In Dumas, M., Reichert, M., Shan, M.C., eds.: BPM. Volume 5240 of Lecture Notes in Computer Science., Springer (2008) 100–115
16. Gagne, D., Shapiro, R.: BPSim 1.0. http://bpsim.org/specifications/1.0/WFMC-BPSWG-2012-01.pdf (Feb 2013)
17. Aalst, W. M. P. van der, Hee, K. M. van: Workflow Management: Models, Methods, and Systems. The MIT Press (January 2002)
18. Yerkes, R.M., Dodson, J.D.: The Relation of Strength of Stimulus to Rapidity of Habit-Formation. Journal of Comparative Neurology and Psychology **18**(5) (1908) 459–482
19. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice Hall PTR (September 1999)
20. Hofstede, A.H.M. ter, Aalst, W.M.P. van der, Adams, M., Russell, N., eds.: Modern Business Process Automation: YAWL and its Support Environment. Springer (2010)
21. Aalst, W.M.P. van der, Adriansyah, A., Dongen, B.F. van : Replaying History on Process Models for Conformance Checking and Performance Analysis. Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery **2**(2) (2012) 182–192
22. Fishman, G.S.: Grouping Observations in Digital Simulation. Management Science **24**(5) (1978) pp. 510–521

An Evaluation of Automated Code Generation with the PetriCode Approach

Kent Inge Fagerland Simonsen[1,2]

[1] Department of Computing, Bergen University College, Norway
Email: kifs@hib.no
[2] DTU Compute, Technical University of Denmark, Denmark

Abstract. Automated code generation is an important element of model driven development methodologies. We have previously proposed an approach for code generation based on Coloured Petri Net models annotated with textual pragmatics for the network protocol domain. In this paper, we present and evaluate three important properties of our approach: platform independence, code integratability, and code readability. The evaluation shows that our approach can generate code for a wide range of platforms which is integratable and readable.

1 Introduction

Coloured Petri Nets (CPNs) [5] is a general purpose formal modelling language for concurrent systems based on Petri Nets and the Standard ML programming language. CPNs and CPN Tools have been widely used to model and validate network protocol models [6]. In previous works [14], we have proposed an approach to automatically generate network protocol implementations based on a subclass of CPN models. We have implemented the approach in the PetriCode tool [13]. In this approach, CPN models are annotated with syntactical annotations called *pragmatics* that guide the code generation process and have no other impact on the CPN model. Code is then generated based on the pragmatics and code generation templates that are bound to each pragmatic through template bindings. This paper presents an evaluation of the PetriCode code generation approach and tool.

The four main objectives of our approach are: platform independence, code integratability, code readability, and verifiability of the CPN models. The contribution of this paper is an evaluation of the first three of these objectives. In this study, we used the PetriCode [13] tool to evaluate our code generation approach. Platform independence, i.e., the ability to generate code for several platforms, is an important feature of our approach. For the purposes of this study, a platform is a programming language and adjoining APIs. Being able to generate protocol implementations for several platforms allows us to automatically obtain implementations for many platforms based on the same underlying CPN model. Platform independence also contributes to making sure that implementations for different platforms are interoperable. Another aspect is to have

models that are independent of platform specific details. Integrateability, i.e., the ability to integrate generated code with third-party code, is important since the protocols must be used by other software components written for the platform under consideration (upwards integratability). It is also important to be able to support different underlying libraries so that the generated code can be referred to by other components (downwards integrateability). Readability is important in order to gain confidence that the implementation of a protocol is as expected. While being able to verify the formal protocol models also contribute to this, inspecting and reviewing the final code further strengthens confidence in the correctness of the implementations. The ability to manually inspect the generated code is useful since, in our approach, we only verify the model which is not sufficient to remove local errors in the code.

The rest of this paper is organized as follows. Section 2 describes the example protocol used throughout this paper, and illustrates the code generation process for the Groovy platform. Section 3 evaluates platform independence by considering the Java, Clojure, and Python platforms. Section 4 evaluates integrateability, and Section 5 evaluates readability of the code generated by our approach. Section 6 presents related work, sums up conclusions and discusses directions for future work. Due to space limitation we provide little on CPNs and Petri Nets. The reader is referred to [5] for details on CPNs and Petri Nets. The PetriCode tool as well as the model, template and bindings used in this paper are available at [10].

2 Example and Code Generation

In this section, we present an example CPN model which is an extension of the protocol model we have used in previous work [14]. The example allows us to introduce the concepts and foundations of our approach and the PetriCode tool as well PA-CPNs [14], the CPN sub-class that has been defined for this approach. The example is a well established and used in the literature to describe CPNs [5]. It is also a natural extension of the example we have been using in previous works [14].

This example is a simple framing protocol which is tolerant to packet loss, reordering and allows a limited number of retransmissions. The top level of the CPN model is shown in Fig. 1. The model consists of three sub-modules. Sender and Receiver represents each of the principal actors of the protocol, and Channel connects the two principals.

The protocol uses sequence numbers and a flag to indicate the last message of a frame. After a frame has been sent, the receiver, if it receives the frame, sends an acknowledgement consisting of the sequence number of the frame expected next. If the acknowledgement is not received, the sender will retransmit the frame until an acknowledgement is received or the protocol fails sending the message.

In the Sender module, shown in Fig.2, there are two sub-modules. The send sub-module is annotated with a $\langle\langle$service$\rangle\rangle$[1] pragmatic and represents a service

[1] Pragmatics in the model and in the text are by convention written inside $\langle\langle\rangle\rangle$.

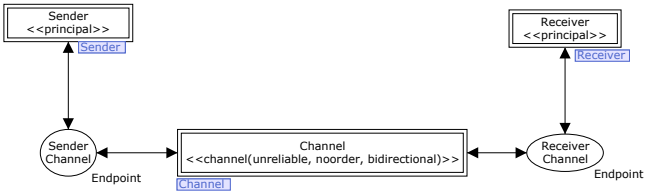


Fig. 1: The protocol system level

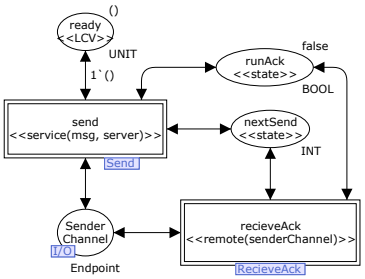


Fig. 2: The Sender principal module

provided by this principal for sending a message. The other substitution transition receiveAck, annotated with an ⟨⟨internal⟩⟩ pragmatic, represents an internal service which is to be invoked by another service of the principal. In this example, the receiveAck service is invoked from the send service.

The Sender module also contains two places, runAck and nextSend, annotated with a ⟨⟨state⟩⟩ pragmatic which contains shared data between the two services. The ready place, annotated with a ⟨⟨LCV⟩⟩ pragmatic, is used to model the lifecycle of the Sender principal and makes sure that only a single message is sent at a time.

The send service, shown in Fig. 3, starts at the transition send which opens the channel, initializes the content of the message to be sent and the sequence number. Also, at this transition, the receiveAck internal service is started by placing a token with the colour true at the ⟨⟨state⟩⟩ place runAck. The service continues from send to enter a loop at the start place. Inside the loop, the sendFrame transition retrieves the next frame to be sent based on the sequence number of the frame which is matched against the sequence number incoming from the place start. The limit place is updated with the sequence number of the current frame, and the number of times the frame has been retransmitted. Then, the current frame is sent. Due to the ⟨⟨wait⟩⟩ pragmatic at the sendFrame transition, the system waits in order to allow acknowledgements to be received. The loop ends at place frameSent. If a token is present on the place frameSent the loop will either continue with the transition nextFrame firing or end by firing the return transition. At the return transition, state places and the channel are cleared and

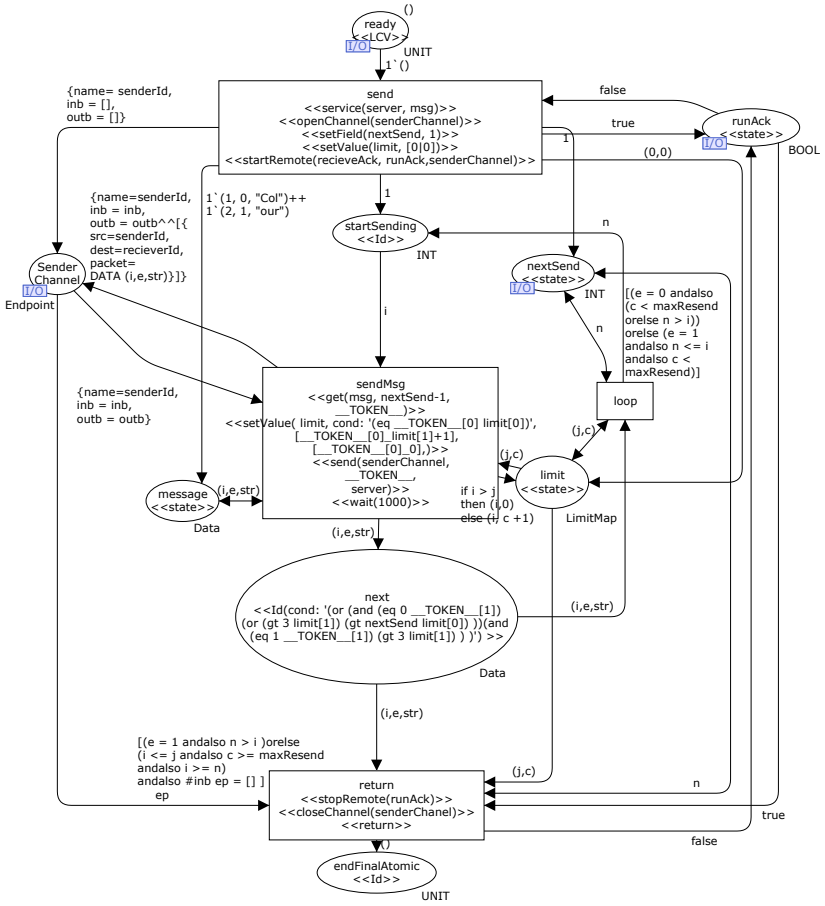


Fig. 3: The Send service module

the service terminates. In the model, we have not shown the pragmatics that can be automatically derived from the CPN model structure, see [14] for details.

The code generation approach is template-based and uses pragmatics to guide the code generation in two ways. The first way is by having structural pragmatics define the principals, services, and control-flow path within each service. The $\langle\langle$principal$\rangle\rangle$, $\langle\langle$service$\rangle\rangle$, and $\langle\langle$Id$\rangle\rangle$ pragmatics in Figs. 1-3. The second way is to define the operations that should occur at each transition. The pragmatics are described in a domain specific language (DSL) and can often be derived from the CPN model structure. Structural pragmatics are used to generate the Abstract Template Tree (ATT), an intermediary representation of the pragmatics annotated CPN model. Each node in the ATT has pragmatics attached. Pragmatics are bound to code generation templates by template bindings. The generation

Listing 1: The Groovy template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
1  def ${name}(${binding.getVariables()
2    .containsKey("params")
3    ?params.join(", "):""}){
4  <%if(binding.variables
5       .containsKey('pre_conds')){
6    for(pre_cond in pre_conds){
7      %>if(!$pre_cond) throw new
8        Exception('...')
9  <% if(!pre_sets.contains("$pre_cond"))
10    {%>$pre_cond = false<%}
11   } }%>
12 %%yield_declarations%%
13 %%yield%%
14 <%if(binding.variables
15       .containsKey('post_sets')){
16   for(post_set in post_sets){
17     %>$post_set = true<%
18   }}%>}
```

```
def bos = new ByteArrayOutputStream()
def oos = new ObjectOutputStream(bos)
oos.writeObject(${params[1]})
_msg_ = bos.toByteArray()
DatagramPacket pack =
  new DatagramPacket(_msg_, _msg_.length,
  InetAddress.getByName(${params[2]}.host),
  ${params[2]}.port)
${params[0]}.send pack
%%VARS:_msg_%%
```

uses these bindings to generate code for each pragmatic at each ATT node. Finally, the code is stitched together using special tags in the templates.

In order to give an overview of the code generation process, we use two templates as examples. The templates are the template for the ⟨⟨service⟩⟩ pragmatic (Listing 1 (left)) and the ⟨⟨send⟩⟩ pragmatic (Listing 1 (right)).

The service template for the Groovy platform is shown in Listing 1 (left). The first line of the template creates the signature of a method what will implement the service. Lines 4 to 10 iterates over preconditions to the ⟨⟨service⟩⟩. Each precondition is checked to make sure that the service may execute. In lines 11-12 two special tags %%yield%% and %%yield_declarations%% indicates the places where the method body and the declarations will be inserted from nodes coming from the sub-nodes in the ATT.

The template for ⟨⟨send⟩⟩ is shown in Listing 1 (right). The template first creates a byte array from the data to be sent and then creates an appropriate data packet and, finally, sends the datagram packet. The template uses UDP as the underlying transport protocol, which is why the packet is created in the form of a `DatagramPacket`.

The Groovy code shown here provides a baseline implementation for the protocol. In the next section we show ho we can generate code from the same model for three other platforms.

3 Evaluating Platform Independence

In order to demonstrate the platform independence of our approach, we have generated code for the Java, Clojure and Python platforms in addition to the Groovy platform. The platforms have been chosen in order to cover three main programming languages and paradigms. Java is an imperative and object oriented programming language. Clojure is a Lisp dialect for the Java Virtual Machine (JVM). It is a functional language, however it is able to utilize Java objects

and the Java API. Python is a multi-paradigm language and, as the only language in this survey, does not rely on the JVM. Python also uses significant white-spaces which makes Python unique in this evaluation in both respects. For each of the platforms, we show selected templates corresponding to the ones shown for the Groovy platform in Sect. 2. In addition, we show an exerpt of the generated code for the Java platform since this was used as part of the evaluation of readability presented in Sect. 5

Listing 2: The Java template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
public Object ${name}(<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "")
      params2 << "Object $it"
  }
  paramsVal = params2.join(", ")
}%>$paramsVal) throws Exception {
<%if(binding.variables
    .containsKey('pre_conds')){
 for(pre_cond in pre_conds){
 %>if(!$pre_cond)
  throw new RuntimeException("...");
  <%if(!pre_sets
      .contains("$pre_cond"))
    {%>$pre_cond=false;<%}
 }}%>
  %%yield_declarations%%
  %%yield%% }
```

```
1  ByteArrayOutputStream bos = new
2      ByteArrayOutputStream();
3  ObjectOutputStream oos = new
4        ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  DatagramPacket pack = new
8    DatagramPacket(_msg_, _msg_.length,
9    InetAddress.getByName((String)
10   ((Map)${params[2]}).get("host")),
11   (Integer) ((Map)${params[2]})
12   .get("port"));
13 ((DatagramSocket)${params[0]})
14 .send(pack);
```

The Java Platform. The ⟨⟨service⟩⟩ template for the Java platform is shown in Listing 2 (left). The main difference from the Groovy service template is that, in the first line, the return type and visibility protection is explicit.

The ⟨⟨send⟩⟩ template (see Listing 2 (right)) is similar to the Groovy ⟨⟨send⟩⟩ template. The differences are mainly caused by the fact that Java is explicitly typed and, at times, requires explicit casts.

Excerpts of the Java code for the Sender principal is shown in Listing 3. The first part is generated from the service template. Lines 1-5 are generated by the ⟨⟨service⟩⟩ template (Listing 2 (left)) and lines 10-17 are generated by the ⟨⟨send⟩⟩ template (Listing 2 (right)).

The Clojure Platform. The Clojure ⟨⟨service⟩⟩ template is shown in Listing 4 (left). It begins by defining a function with the name set to the name parameter. Then it creates a vector which holds incoming variables. Finally, it yields for declarations and the body of the function.

The networking templates for Clojure uses the Java networking API and the ⟨⟨send⟩⟩ template (see Listing 4 (right)) and is therefore reminiscent of Groovy

Listing 3: The Java code for the send service.

```
1  public Object send(Object msg, Object  server) throws
2        Exception { /*[msg,  server]*/ /*[Object msg, Object  server]*/
3    if(!ready) throw new RuntimeException(
4              "unfulfilled precondition: ready");
5    ready = false;
6    ...
7    __LOOP_VAR__ = true;
8    do{
9      ...
10     ByteArrayOutputStream bos = new ByteArrayOutputStream();
11     ObjectOutputStream oos = new ObjectOutputStream(bos);
12     oos.writeObject(__TOKEN__);
13     byte[] _msg_ = bos.toByteArray();
14     DatagramPacket pack = new DatagramPacket(_msg_, _msg_.length,
15         InetAddress.getByName((String)((Map)
16         server).get("host")),(Integer) ((Map)server).get("port"));
17     ((DatagramSocket)senderChannel).send(pack);
18     ...
19   }while(__LOOP_VAR__);
20   ...
21 }
```

Listing 4: The Clojure template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
(defn ${name} <%
def paramsVal = ""
def params2 = []
if(binding.getVariables().
 containsKey("params")){
 params.each{
   if(it.trim() != "") params2 << "$it"
 }
 paramsVal = params2.join(", ")
%>[$paramsVal]<%}%>
(%%yield_declarations%%
%%yield%%))
```

```
(def bos (ByteArrayOutputStream.))
(.writeObject
 (ObjectOutputStream. bos)
 @${params[1]})
(def _msg_ (.toByteArray bos))
(.send  ${params[0]}
  (DatagramPacket.
    _msg_ (alength _msg_)
    (InetAddress/getByName
    (.get ${params[2]} "host") )
    (.get ${params[2]} "port"))
)
```

and Java templates. First, the message is converted into a byte array using `java.io` streams. Then a data packet is constructed and sent using the socket given as a parameter.

The Python Platform. The Python ⟨⟨service⟩⟩ template is shown in Listing 5 (left). The template defines the method in line 2 and adds parameters, given by the template variable paramsVal in line 10. Finally, the template yields for declarations and the method body in lines 12-13.

The Python template for ⟨⟨send⟩⟩ is shown in Listing 5(right). The data using Python is a simple call to the sendto function of a socket given as params[0] with the serialized data given in params[1] and the host and port from params[2] in a tuple.

Discussion. The examples above demonstrate that our approach allows us to generate code for several platforms by providing a selection of templates for

each platform. The platforms considered, spanning several popular paradigms, gives us confidence that our approach and tool can also be applied to generate code for many other platforms. Furthermore, we are able to generate the code for each of the platforms using the same model with the same annotations and the same code generator while only varying the code generation templates and the mappings between the pragmatics and mappings between pragmatics and code templates.

Adapting the Groovy templates to Java was, for the most part simple since the two languages are similar in several respects. However, whereas Groovy is optionally typed, Java is statically typed and requires all variables to be typed or to be cast to specific types when accessing methods. Fulfilling Java's requirements for explicit types requires functionality from PetriCode so that the templates are aware of the type of variables.

Clojure is a functional language with a different control flow from languages such as Java. The main issue, compared with Groovy and Java, was related to using immutable data-structures. In Clojure all data types are, in principle, immutable. However, there is an Atom type in which values may be swapped. This was challenging because Atom values must be treated differently from pure values and lead to somewhat more verbose code than what could otherwise have been written. Also, Clojure allows the use of Java data structures, which are mutable and thus easier to work with in this case.

Python, as Groovy, is a multi-paradigm language combining the features of object oriented and functional paradigms. Creating the templates of the Python code was, although being the only language in this survey not based on the JVM, no more difficult than for the other languages. The main challenge was to handle the significant white-spaces of the Python syntax. To support this, PetriCode contains functionality to keep track of the current indentation level. This required no special treatment and was not strictly necessary, but allowed for much cleaner templates.

Table 1 shows the sizes of the Sender and Receiver principal code (measured in code lines) for each of the platforms considered. As can be seen, the code for

Listing 5: The Python template for ⟨⟨service⟩⟩ (left) and for ⟨⟨send⟩⟩ (right).

```
<%import static org.kls.petriCode.
         generation.CodeGenerator.indent
%>${indent(indentLevel)}def ${name}(self,<%
def paramsVal = ""
def params2 = []
if(binding.getVariables()
   .containsKey("params")){
  params.each{
    if(it.trim() != "") params2 << "$it"
  }
  paramsVal = params2.join(", ")
%>$paramsVal<%}%>):
%%yield_declarations%%
%%yield%%
```

```
<%import static org.kls.petriCode.
   generation.CodeGenerator.indent
%>${indent(indentLevel)}
   ${params[0]}.sendto(
     pickle.dumps(${params[1]}),
     (${params[2]}["host"],
     ${params[2]}["port"]))
```

Python is much smaller than the others. This is due to the efficient libraries in Python and that the Python code, for technical reasons, have much fewer blank lines which is also reflected in the templates. Table 2 shows the sizes, in lines, for selected templates and all the templates for each platform. The sizes reported are the sizes in the actual code and may not correspond to the templates as they are formatted in this paper. In this example, there was the same number of templates for each platform, but this is not necessarily always the case. As can be seen in Table 2, there is not a perfect correlation between the size of templates and the size of the generated code. This is due to, in part, some templates being more complex for some languages than others and template reuse being possible for some languages. An example is the Clojure templates, where the templates for the $\langle\langle$setField$\rangle\rangle$ and $\langle\langle$setValue$\rangle\rangle$ pragmatics are the same, but since the $\langle\langle$setValue$\rangle\rangle$ template has more functionality than the $\langle\langle$setField$\rangle\rangle$ template for all platforms, this results in a higher total number of template lines for Clojure. For each of the languages eleven new templates were constructed while ten templates were already provided as part of the PetriCode tool. The new templates were templates that are specific to the pragmatics applied for the protocol considered.

4 Evaluating Intergrateability

It should be possible to integrate code generated by our approach with existing software. We evaluate two types of integration with other software. The first type can be exemplified by having our generated cod use another library for sending and recieving data from the network. We call this type of integration downwards integration (i.e, generated code can use different third-party libraries). The other type can be exemplified by creating a runner program that employs the generated protocol for sending a message to a server. This type is called upwards integration (i.e, applications can use services provided by the generated code). We have evaluated integratability using the code generated for the Java platform based on the example in Sect. 2. However, the results are applicable for other platforms as well.

Downwards Integration. We have already shown that by changing templates, our approach can be used to generated code for different platforms. The same technique can be used to employ various libraries on the same platform to perform the same task. We illustrate this by changing the network library from the standard `java.net` library to Netty [16]. This example was chosen because

Language	Groovy	Java	Clojure	Python
Sender	131	132	119	66
Receiver	81	78	68	38
Total	212	210	187	104

Table 1: Sizes of the generated code.

Language	Groovy	Java	Clojure	Python
service	19	28	15	15
runInternal	4	10	4	3
send	9	9	8	2
All templates	154	219	251	112

Table 2: Size of code generation templates.

Listing 6: The Java template for ⟨⟨send⟩⟩ with Netty (left) and the runner for the generated Java code (right).

```
1  ByteArrayOutputStream bos =
2    new ByteArrayOutputStream();
3  ObjectOutputStream oos =
4    new ObjectOutputStream(bos);
5  oos.writeObject(${params[1]});
6  byte[] _msg_ = bos.toByteArray();
7  ((io.netty.channel.Channel)
8    ${params[0]}[0]).writeAndFlush(
9      new io.netty.channel.socket
10       .DatagramPacket(
11       io.netty.buffer.Unpooled
12       .copiedBuffer(_msg_),
13     new InetSocketAddress(InetAddress
14       .getByName((String)((Map)
15         ${params[2]}).get("host"))
16     ,(Integer)((Map)${params[2]})
17     .get("port")))).sync();
```

```
1  def sender = new Sender.Sender()
2  def reciever =
3      new Receiver.Receiver()
4  t = new Thread().start {
5    def ret = reciever.receive(31339)
6    println "Recieved: ${ret}"
7  }
8  def msg = [
9   [1,0,'Col'],[2,0,'our'],[3,0,'ed '],
10  [4,0,'Pet'],[5,0,'ri '],[6,0,'Net'],
11  [7,1,'s']
12 ]
13 sender.send(
14   msg,[host:"localhost", port:31339])
```

networking is an important function of the network protocol domain that we consider, and because Netty is substantially different from `java.net` as it is an event driven library.

Three out of twenty-one templates had to be altered to accommodate Netty as the network library for the sender principal. These were the templates that generate code for sending and receiving data from the network. We show the Netty variant of the send template from Listing 2 (right) in Listing 6 (left). The main differences is the call to the socket (or channel in the terminology of Netty) to send the message (lines 6-12).

Upwards Integration. The ability to call the generated code is necessary for the code to be useful in many instances. Our approach allows this by explicitly modelling the API in the CPN protocol model in the form of services which defines the class and method names. To demonstrate upwards integration, we have created runners for the generated implementations for each of the platforms considered. The runner for the Java platform can be seen in Listing 6 (right). This demonstrates that it is possible to use the generated services from third party software. It is worth noticing that the explicit modelling of services in the CPN model implies that it is simple to invoke the generated code.

5 Evaluating Readability

We have evaluated the readability of code generated by PetriCode in two ways. One way is that we applied a code readability metric [2] to selected snippets of the generated classes from the example described in Sect. 2, and the example described in previous works [14]. Furthermore, we have conducted a field study

The Buse-Weimer experiment (BWE)	The experiment conducted by Buse and Weimer to create the BWM. The snippets were selected from open source experiments.
The metric experiment (ME)	Our experiment to validate the results from BWE for professional developers. This experiment evaluated the twenty first snippets evaluated in the *BWE*.
The code generation experiment (CGE)	Our experiment to evaluate readability of generated code compared to non-generated code. Eight snippets were randomly selected from generated code and twelve from the open source projects in the network protocol domain.

Table 3: Overview of the experiments conducted and discussed in this section

where software engineers were asked to evaluate the readability of the generated code. This study was also used to evaluate the code readability metric.

We use the *Buse-Weimer metric* [2] (BWM) as a code readability metric. This metric was constructed by Buse and Weimer based on an experiment (the *Buse-Weimer experiment* (BWE), see Table 3) asking students at the University of Virginia to evaluate short code snippets with regards to readability on a scale of one to five. The experiment was used to construct the metric using machine learning methods to compute weights on various factors that have an impact on code readability. The final metric scores code snippets on a scale from zero to one where values close to zero indicates low readability and values close to one indicates a high degree of readability.

Our field study with software engineers took place at the JavaZone software developer conference in Oslo, Norway in September 2013. The experiment was organized into two parts. One part (*the metric experiment* (ME), see Table 3) evaluated the BWM. The other part (the *code generation experiment* (CGE), see Table 3) evaluated the readability of the generated code compared to non-generated code. Both experiments were conducted by asking software developers to evaluate twenty small code snippets with regards to readability by assigning values, on a scale from one to five, to each code snippet. The experimental set-up was created to mimic the BWE. The main advantage of our experiment over the BWE is that the dominating majority of the participants were professional software developers instead of students. The ME had 33 participants while the CGE had 30 participants.

For the CGE, we randomly selected code snippets from code generated for the Java platform based on the example described in Section 2, and the example described in [14]. We use code for the Java platform because it was used in the BWE, and the subjects of our experiments knew Java. Also, there exist several Open Source projects from which to obtain snippets for our experiments. In addition to the generated snippets, we selected, as controls, snippets from three Open Source projects in the network protocol domain. These were the Apache FtpServer, HttpCore and Commons Net [15]. All three are part of the Apache project, and we consider them to be high quality projects within the network protocol domain.

In the ME, we used the first twenty snippets from the BWE. Since we did our experiment at a conference, we could not redo the experiment with all the

Snippet	1	2	3	4	5	6	7	8	Mean	Median
Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,58	0,63

Table 4: The results for the BWM on generated code

Snippet	1	2	3	4	5	6	7	8	9	10	11	12	Mean	Median
Score	0,54	0,95	0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65	0,40	0,33

Table 5: The results for the BWM on selected hand-written protocol software snippets

one hundred snippets from the BWE and still expect enough software engineers to participate.

Applying the Buse-Weimer Metric. The BWM is based on the scores of hundred small code snippets. Even though the size of the snippets are not scored directly, some of the factors are highly correlated with the snippet size [11]. This makes it inappropriate to measure entire applications. Therefore, we applied the metric to the snippets selected for the CGE.

Table 4 shows the results of running the BWM tool on each of the generated code snippets. The mean and median score is above 0.5, indicating that the code is fairly readable. Also the mean and median of the generated code is higher than for the non-generated protocol-code as can be seen in Table 5.

Although the scores of the BWM on the generated code are highly encouraging, the scores are either very high or very low. This motivates an independent evaluation of the readability of the generated code (see below), as we have done with the CGE, and to validate the BWM, as we have done with the ME (see below).

The Metric Experiment: Validating the Buse-Weimer Metric. The ME was conducted to validate the BWM and the BWE. This experiment measured twenty of the code snippets that were measured in the BWE in a similar manner. The goal was to determine whether the results of the BWE holds for professional software developers.

Figure 4 shows the means of the BWE (blue/solid) and our repeat (red/-dashed) for the selected snippets. The figure suggests significant covariance even if the students in the BWE tended to judge snippets higher than the software developers in our ME. We computed three statistical tests on the correlation between the means of the two experiments (see Table 6). The correlation tests show that there is strong to medium correlations between the means and that the correlation is statistically significant($p<0.05$).The correlation tests were carried

Method	**Corrolation**	**P-value**
Pearson	cor = 0,82	$9{,}28 \cdot 10^{-06}$
Spearman	rho = 0,79	$2{,}94 \cdot 10^{-05}$
Kendall	tau = 0,61	$1{,}65 \cdot 10^{-04}$

Table 6: Correlations between the means of the ME and the BWE

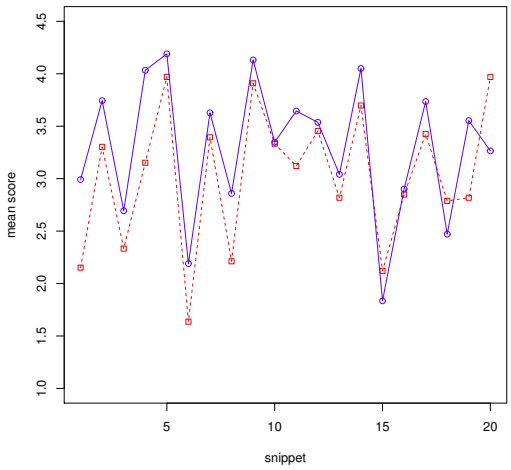


Fig. 4: The values of the selected snippets for the BWE and the ME.

out using the R [12] tool and the standard correlation test call, `corr.test()`, with all the methods available for the call. These results indicate that the BWE is relevant to professional software developers.

Mean values for the original BWE and for the ME are shown in Table 7. As can be seen, *the ME* resulted in somewhat lower scores than that of the BWE, in fact it is lower in 17 out of 20 instances. In order to determine the significance of this observation we conducted a T-test. The results of the T-test does not allow us to rule out that the means are not equal (p=0,21), although it does not give us statistically significant results on the repeat always being higher either (p=0,10), although that may be more likely.

The ME showed that there is a significant correlation between the results of the BWE (conducted with students) and the *ME* (conducted with software development professionals). This can be interpreted as evidence that the results from BWE also has validity for professional developers, although the metric based on it might be in need of some minor adjustments.

Snippet	1	2	3	4	5	6	7	8	9	10
Metric Experiment	2,15	3,30	2,33	3,15	3,97	1,64	3,39	2,21	3,91	3,33
Buse-Weimer Experiment	3,02	3,78	2,72	4,07	4,23	2,21	3,66	2,88	4,17	3,38
	11	12	13	14	15	16	17	18	19	20
	3,12	3,45	2,82	3,70	2,12	2,85	3,42	2,79	2,82	3,97
	3,68	3,57	3,07	4,08	1,85	2,93	3,77	2,49	3,58	3,29

Table 7: Snippet means for the metric and BWE.

The Code Generation Experiment: Comparing Generated and Handwritten Code. We expected that the generated code would not do quite as well as the hand-written high-quality code used as control. Therefore, our hypothesis was that the generated code would be within the standard deviation of the hand-written written code. The mean score for each of the snippets in the CGE are shown in Table 8. Snippets one to eight are generated code while snippets eight to twenty are hand-written. To check our hypothesis, we ran Welch's T-test on the results which is useful for determining the difference between the two samples. The first hypothesis we checked was whether the generated snippets are less readable than the hand-written ones. The results of a Welch's Two Sample t-test showed that the generated code-snippets scored below that of the hand-written code ($p=4{,}81 \cdot 10^{-05}$).

Then we checked the hypothesis when reducing the score of the measurements from the Apache projects by the standard deviation of those measurements. The Welch's Two Sample t-test indicates that the generated code scores better than one standard deviation below the hand-written code (p=0,03). This indicates that our goal of being readable within a standard deviation of non-generated code is met both by measuring via the BWM and experimentally.

Table 9 shows normalized means for each snippet from the CGE and the results of running the BWM on the corresponding snippets. As can be seen, the correlation is less than strong as confirmed by running correlation tests (see Table 10). Even though the results of the ME indicates that the BWE, which the BWM is based on, is valid even for professional software developers, we content that the results of the CGE are more reliable than the BWM. This is because the BWM is derived from software from different domains and that it is sensitive to snippet length. This indicates that the BWM is not relevant to code for network protocols.

Assessment of Validity of Our Results. As with most experimental approaches, this evaluation has some threats to the validity of the results. These are issues we have identified that might skew our results. One such threat to validity for the original BWE was that they used student as subjects who may or may not disagree with professional software developers on the readability of code. We have tried to alleviate this threat in the ME by repeating part of the BWE with professional developers. Further threats to validity to the experiments and results described in this section are discussed in the following.

Small sample size and limited number of participants may skew the results. Since we conducted this experiment at a software developer conference where people tended to be on their way to some lecture, we had to limit the number

Snippet	1	2	3	4	5	6	7	8	9	10
Mean	2,40	2,10	3,83	3,23	2,67	3,13	2,97	2,73	3,90	3,97
	11	12	13	14	15	16	17	18	19	20
	2,67	3,13	2,73	3,43	3,67	3,07	3,83	2,00	3,20	3,93

Table 8: Means of results for generated code (1-8) and Apache projects code (9-20).

Snippet	1	2	3	4	5	6	7	8	9	10
Experiment Score	0,48	0,42	0,77	0,65	0,53	0,63	0,59	0,55	0,78	0,79
Metric Score	0,14	0,03	0,19	0,28	1,00	1,00	1,00	0,99	0,54	0,95
	11	12	13	14	15	16	171	18	19	20
	0,53	0,63	0,55	0,69	0,73	0,61	0,77	0,40	0,64	0,79
	0,15	0,79	0,01	0,40	0,26	0,04	0,00	0,01	0,96	0,65

Table 9: Normalized means from the CGE and results from applying the BWM

Method	**Corrolation**	**P-value**
Pearson	cor = 0,21	0,37
Spearman	rho = 0,20	0,40
Kendall	tau = 0,14	0,40

Table 10: Correlation between normalized experimental scores and the BWM applied to the same snippets

of snippets we asked each participant to evaluate. Also, because professional software developers are harder to recruit than students, the number of participants was limited. Furthermore, it is possible, albeit unlikely, that the people participating in the experiment are not representative for software developers as a whole. These threats can be alleviated by conducting broader studies on larger groups of developers and using interviews.

In our experiments, we used small randomly selected code snippets as proxies for code readability. We do this both for practical and conceptual reasons. The practical reasons revolve around what we realistically could expect participants to score. If they had to read entire classes or software projects in order to score the code, this would have taken to much time and could have resulted in getting too few participants in our experiments. Furthermore, we wanted to evaluate the BWM since it is the only implemented metric we could find in the literature. The more conceptual reason is that if each snippet is readable, then the whole code is likely to be readable as well. In our approach, high-level understanding is based more on the CPN models of the protocols than on the implementation, so it makes sense for us to concentrate on low level, snippet-sized readability, since readability in the large is intended to be considered at the level of the model.

6 Conclusions and Related Work

In this paper, we have evaluated our code generation approach and supporting software, with respect to platform independence, the integratability of the generated code as well as the readability of the generated code.

Platform independence was evaluated by generating code for a protocol for three platforms in addition to the Groovy platform from a single CPN model. The number of and differences between the platforms gives us confidence that our approach and the PetriCode tool can be used to generate protocol implementations for many target platforms. All the platforms considered have automatic memory management in the form of garbage collection. This is convenient, but

we intend to support platforms without automatic memory management in the future.

Platform independence is especially important for network protocols since they are used to communicate between two or more hosts that often run on different underlying platforms. Although there exists many tools that allow generating code from models claiming to be platform independent, few studies seem to have been made actually generating code for several platforms.

MDA [8] and associated tools rely on different platform specific models (PSM) to be derived for platforms before generating code for each platform. This adds an extra modelling step compared to our approach and may require somewhat different PSMs for different platforms. The Eclipse Model To Text (M2T) [3] project provides several template languages for code generation from Ecore models. In general, M2T languages can generate code for several platforms. However, to go beyond pure structural features and standard behaviour, the developer must create customized code generators. In [9] code is for protocol is generated using UML stereotypes and various UML diagram types. The UML diagrams, annotated with stereotypes according to a custom made UML profile, combined with a textual language named GAEL are used to obtain protocol specification in the Specification and Description Language (SDL) [1, 4]. The authors also conjecture that the approach can be used to generate code for any platform. The use of stereotypes in the approach presented in [9] is similar to the pragmatics that our approach uses. However, a difference is that several diagram types are used in the UML based approach in contrast to our approach where we use CPNs to describe both structure and behaviour.

MetaEdit+ [17] allows code generation of visual Domain Specific Modelling Languages (DSMLs). MetaEdit+ and the DSML approach is similar to the PetriCode approach since CPNs and pragmatics constitute a DSML. A main difference is that MetaEdit+ allows users to generate custom graphical languages while PetriCode uses CPN, but extends CPNs using pragmatics. This allows us to use the properties of CPNs for verification and validation, and also to use a single syntax for different domains.

The Renew [7] tool uses a simulation-based approach where annotated Petri Nets can be run as stand-alone applications. The simulation-based approach is fundamentally different from our approach where the generated code can be inspected and compiled in the same way as computer programs created with traditional programming languages. A detailed comparison between these two approaches would be an interesting avenue for future work.

We evaluated the integratability of the generated code in two directions: upwards and downwards integratability. Upwards integratability was evaluated by showing that the generated protocol software can be called by programs running the protocols. Downwards integratability was evaluated by showing how we can change the network API for the Java platform by binding different templates to some of the pragmatics.

Readability of the generated code was evaluated by an automatic metric and an experiment. According to the BWM, the generated code is as, or possibly

even more, readable than the samples of high quality code in the same domain that we used for comparison. Based on our experiment with software developers, however, the generated code is somewhat less readable but within a standard deviation of the non-generated code. A contribution of this paper is also to provide evidence that the experimental results from the BWE are relevant to professional software developers in addition to the students. However, based on the discrepancy between the experimental evaluation, it seems that the BWM may not be applicable to code in the network protocol domain. To the best of our knowledge, there are no previous work evaluating intergrateability and readability of automatically generated software.

In the future we will evaluate the verifiability of the models used in our approach by applying verification techniques to example protocols. We also intend to develop a set of template libraries that can be used for code generation as well as procedures for testing code generation templates. Another possible direction for future work is to apply our code generation approach to other domain.

References

1. F. Babich and L. Deotto. Formal methods for specification and analysis of communication protocols. *Communications Surveys Tutorials, IEEE*, 4(1):2–20, 2002.
2. R.P.L. Buse and W.R. Weimer. A metric for software readability. In *Proc. of ISSTA'08*, pages 121–130, NY, USA, 2008. ACM.
3. IBM. *Eclipse Model To Text (M2T)*. `http://www.eclipse.org/modeling/m2t/`.
4. ITU-T. Recommendation z.100 (11/99) specification and description language (sdl), 1999.
5. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
6. L.M. Kristensen and K.I.F. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *ToPNoc VII*, volume 7480 of *LNCS*, pages 56–115. Springer, 2013.
7. O. Kummer et al. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In *Proc. of ICATPN '04*, volume 3099 of *LNCS*, pages 484–493. Springer, 2004.
8. Object Management Group. *MDA Guide*, June 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.
9. J. Parssinen, N. von Knorring, J. Heinonen, and M. Turunen. UML for protocol engineering-extensions and experiences. In *Proc. of TOOLS '00*, pages 82–93, 2000.
10. PetriCode. *Example protocol*. `http://bit.ly/19HU8U4`.
11. D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proc. of MSR '11*, pages 73–82. ACM, 2011.
12. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2013.
13. K. I. F. Simonsen. Petricode: A tool for template-based code generation from cpn models. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *LNCS*, pages 151–163. Springer, 2014.
14. K. I. F. Simonsen, L. M. Kristensen, and E. Kindler. Generating Protocol Software from CPN Models Annotated with Pragmatics. In *Formal Methods: Foundations and Applications*, volume 8195 of *LNCS*, pages 227–242. Springer, 2013.

15. The Apache Software Foundation. FtpServer http://mina.apache.org/ftpserver-project/, HttpCore https://hc.apache.org/, Commons Net http://commons.apache.org/proper/commons-net/.
16. The Netty project. *Netty*. http://netty.io.
17. J.P. Tolvanen. MetaEdit+: domain-specific modeling for full code generation demonstrated. In *Proc of SIGPLAN '04*, pages 39–40. ACM, 2004.

Computing Minimal Siphons in Petri Net Models of Resource Allocation Systems: An Evolutionary Approach

Fernando Tricas[1], José Manuel Colom[1], and Juan Julián Merelo[2]

[1] Depto de Informática e Ingeniería de Sistemas
Universidad de Zaragoza
{ftricas,jm}@unizar.es
[2] Depto. ATC/CITIC
Universidad de Granada
jmerelo@geneura.ugr.es

Abstract. Petri Nets are graph based tools to model and study concurrent systems and their properties; one of them is *liveness*, which is related to the possibility of every part of the system to be activated eventually. Siphons are sets of places that have been related to liveness properties. When we need to deal with realistic problems its computation is hard or even impossible and this is why in this paper we are approaching it using evolutionary computation, a meta-heuristic that has proved it can successfully find solutions when the search space is big. In this work a formulation of the siphon property using linear constraints is presented for general Petri Nets. We will also present an evaluation for a family of resource allocation systems (RAS). The proposed solution is based on a genetic algorithm (GA); we will show how siphons can be computed using it, with experiments showing that in some cases they are able to find a few solutions in less time than previous deterministic algorithms.

Keywords: Siphons, genetic algorithms, computing, deadlock prevention

1 Introduction

A Resource Allocation System (RAS) is a discrete event system in which a finite set of concurrent processes shares in a competitive way a finite set of resources. RAS are usually complex enough to take advantage of the use of formal methods, which can help to improve its understanding, providing tools for the analysis and implementation steps. They also help in the dialog between people involved in the design, construction and system management. Our proposal is to use Petri (or Place/Transition) Nets as a tool for this purpose. They are used to visualize and, through formal analysis, describe structural properties of the system they represent[1].

Software systems are also complex systems that can be seen as a set of processes sharing (and competing for) resources. There is some recent work in this

area, such as [2] where a more detailed discussion of similarities and differences with Flexible Manufacturing Systems (FMS) as the archetypal example of RAS can be seen. In [3, 4] there is some work related to software systems and special classes of Petri Nets for concurrency problems. The competition for resources implies the existence of deadlocks; they occur when some processes are waiting for the evolution of other processes, that are also waiting for the former ones to evolve (the dependence does not need to be direct). RAS have proved to be specially useful when synthesizing deadlock avoidance and prevention policies, and many of the published work relies on minimal siphons for this [5–10]. A minimal siphon is a set of places such that existence of any edge from a transition t to a place of D implies that there is an edge from some place of D to t. When a siphon reaches a state with no tokens, it will never become marked again; for this reason they are related to liveness properties. In consequence, some (efficient) methods to compute these structural components are needed.

In [10] some promising work has been done in the field of Flexible Manufacturing Systems. They propose to reduce the number of siphons to be considered for deadlock prevention, but they do not avoid the computation of the whole set of minimal siphons. In most cases siphon enumeration cannot be avoided, and this makes interesting to obtain better methods to find them ([5, 11, 12, 9]).

In this work we are going to propose a genetic algorithm (GA). that uses a formulation of the siphon property by means on linear constraints. This implementation has been tested in a well-known family of RAS. We will show how we can compute siphons using a genetic algorithm with an existing generic package. This approach opens the door to adapt another siphon-based techniques for deadlock prevention.

The contents of this paper are organized as follows. Section 2 provides an introduction to Petri Nets and the main concepts related to the problem, Section 3 presents the standard Genetic Algorithm. There is also some information about methods existing in the literature for solving the same problem, Section 4 presents the adapted method, Section 5 shows our experimental setup and the experimental results, together with some discussion about them. Finally, some conclusions are presented.

2 Petri Nets

A Petri net (or Place/Transition net) is a 3-tuple $\mathcal{N} = \langle P, T, W \rangle$ where P and T are two non-empty disjoint sets whose elements are called *places* and *transitions*, respectively. In a generic way, elements belonging to $P \cup T$ are called *nodes*. $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ defines the *weighted flow relation*: if $W(x, y) > 0$, then we say that there is an arc from x to y, with weight or multiplicity $W(x, y)$. Ordinary nets are those where $W : (P \times T) \cup (T \times P) \to \{0, 1\}$.

Given a net $\mathcal{N} = \langle P, T, W \rangle$ and a node $x \in P \cup T$, ${}^\bullet x = \{y \in P \cup T \mid W(y, x) > 0\}$ is the *pre-set* of x, while $x^\bullet = \{y \in P \cup T \mid W(x, y) > 0\}$ is the *post-set* of x. This notation is extended to a set of nodes as follows: given $X \subseteq P \cup T$, ${}^\bullet X = \bigcup_{x \in X} {}^\bullet x$, $X^\bullet = \bigcup_{x \in X} x^\bullet$.

A Petri net is *self–loop free* when $W(x,y) \neq 0$ implies that $W(y,x) = 0$. The Pre–incidence matrix $\mathbf{Pre} : P \times T \to \mathbb{N}$ of $\mathcal{N}$ is $\mathbf{Pre}[p,t] = W(p,t)$. The Post–incidence matrix $\mathbf{Post} : P \times T \to \mathbb{N}$ of $\mathcal{N}$ is $\mathbf{Post}[p,t] = W(t,p)$. A self–loop free Petri net $\mathcal{N} = \langle P, T, W \rangle$ can be alternatively represented as $\mathcal{N} = \langle P, T, \mathbf{C} \rangle$ where $\mathbf{C}$ is the incidence matrix: a $P \times T$ indexed matrix such that $\mathbf{C}[p,t] = W(t,p) - W(p,t) = \mathbf{Post}[p,t] - \mathbf{Pre}[p,t]$. A *marking* is a mapping $\mathbf{m} : P \to \mathbb{N}$; in general, markings are represented in vector form. A transition $t \in T$ is *enabled* for a marking $\mathbf{m}$ if and only if $\forall p \in {}^\bullet t \,.\, \mathbf{m}[p] \geq W(p,t)$; this fact will be denoted as $\mathbf{m} \xrightarrow{t}$ (or $\mathbf{m}[t\!>$). If t is enabled at $\mathbf{m}$, it can *occur*; when it occurs, this gives a new marking $\mathbf{m'} = \mathbf{m} + \mathbf{C}[P,t]$; this will be denoted as $\mathbf{m} \xrightarrow{t} \mathbf{m'}$ (or $\mathbf{m}[t\!>\!\mathbf{m'}$), and we say that $\mathbf{m}$' is reached from $\mathbf{m}$ by the occurrence of t. The *state equation* of a marked net is an algebraic equation that gives a necessary condition for the reachability of a marking from the initial marking: a markings $\mathbf{m} \in \mathbb{N}^{|P|}$ such that $\exists \boldsymbol{\sigma} \in \mathbb{N}^{|T|} \,.\, \mathbf{m} = \mathbf{m_0} + \mathbf{C} \cdot \boldsymbol{\sigma}$ is said to be *potentially reachable*. The *potentially reachability set* of a net is the set of solutions for the *state equation*. *Flows* (*Semiflows*) are integer (natural) annullers of matrix $\mathbf{C}$ (That is, a vector, $\mathbf{y} \neq \mathbf{0}$ such that $\mathbf{y} \cdot \mathbf{C} = \mathbf{0}$). Right and left annullers are called *T–(Semi)flows* and *P–(Semi)flows*, respectively. The *support* of P–(Semi)flows is given by: $\|\mathbf{y}\| = \{p \in P \mid \mathbf{y}[p] > 0\}$. Let $\mathcal{PS}$ be the set of minimal P–Semiflows of $\mathcal{N}$. A (Semi)flow is called *minimal* when its support is not a strict super-set of the support of any other, and the greatest common divisor of its elements is one. A P–Semiflow $\mathbf{y}$ defines the following invariant property: $\forall \mathbf{m_0} \,.\, \forall \mathbf{m} \in \mathrm{PRS}(\mathcal{N}, \mathbf{m_0}) \,.\, \mathbf{y} \cdot \mathbf{m} = \mathbf{y} \cdot \mathbf{m_0}$ (cyclic behavior law).

Given $\mathcal{N}$ an ordinary Petri net, a subset of places $D \subseteq P$ is a *siphon* ($E \subseteq P$ is a *trap*) of the net $\mathcal{N}$ if, and only if, ${}^\bullet D \subseteq D^\bullet$ ($E^\bullet \subseteq {}^\bullet E$). A siphon (trap) is minimal if, and only if, it does not properly contain another siphon (trap). Siphons have the important property that, if at a given marking the siphon is unmarked, it will never be marked. Researchers have considered and studied different methods for finding siphons and traps. Among them let us present the main types, that we will classify based on the underlying techniques used for their computation: Algebraic methods compute families of siphons by means of the solution of a set of linear equations or inequalities. They use the net incidence–matrix or a transformation of it. Methods using this approach can be found in [13]. Methods based on graph theory directly use the graph representation of the Petri net to compute siphons: methods using this approach can be found in [14, 15]. Methods based on logic formulas are based on characterizing siphons by means of boolean variables, which typically represent places or transitions and their relations. Methods using this approach can be found in [16, 17].

3 Genetic Algorithms

Genetic algorithms [18] are inspired by Darwin's theory about evolution and its genetic-molecular basis. More technically the genetic algorithm is a search heuristic that mimics the process of natural selection. A random population

of candidate solutions is evolved trying to explore the search space looking for better solutions. The sketch of the basic genetic algorithm is [19]:

1. (Start) Generate random population of n chromosomes (suitable solutions)
2. (Fitness) Evaluate the fitness of each chromosome in the population
3. (New population) Create a new population by repeating the following steps until the new population is complete
   (a) (Selection) Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
   (b) (Crossover) With some probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
   (c) (Mutation) With a mutation probability mutate new offspring at each locus (position in chromosome).
   (d) (Accepting) Place new offspring in a new population
4. (Replace) Use new generated population for a further run of algorithm
5. (Test) If the end condition is satisfied, stop, and return the best solution in current population
6. (Loop) Go to step 2

The main task of a genetic algorithms designer is to find good parameter settings (population size, encoding, selection criteria, genetic operator probabilities, fitness evaluation, ...). We have used the `Algorithm::Evolutionary` [20] implementation following the example `tide_bitstring.pl` for the experiments. There are many other available implementations, but this one is known by the authors, is written in Perl and needs just a few lines of code to be adapted to new problems. Since it is written in an interpreted scripting language it can be, in general, slower than other libraries written in Java or C++.

4 The Proposed Approach

As far as we know, there are no approaches using genetic algorithms to compute structural properties of Petri Net models. Some work has been done on process mining and scheduling [21–24]. A siphon is a special set of places, as defined above. In [25] the method presented in [26] (algebraic based) was selected, taking advantage of a parallel approach. Here we will explore a logic formula based approach: with the formulation for siphons presented in [27] we will explore the space state by means of the use of a genetic algorithm in its more classic way.

It is straightforward to try to use the standard GA without much difficulty: each place p of the Petri net will be represented by means of a binary variable v_p. The siphon property can be represented as follows:

$$\forall p \in P, \forall t \in {}^\bullet p, v_p \leq \sum_{q \in {}^\bullet t} v_q, \text{with } v_q, v_p \in \{0, 1\} \tag{1}$$

The siphon would be composed of the set of places whose corresponding variable equals to one, $v_p = 1$. The meaning of each equation is that if place p is in the set (it belongs to the siphon) it must contain, at least, one of the

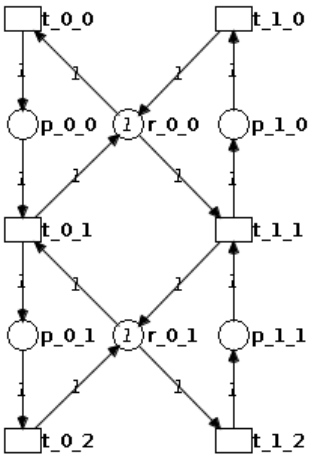
places that are in the pre–set of each of its entry transitions. This needs to be completed with some restrictions that avoid undesired situations:

$$\sum_{p \in P \setminus P_0} v_p < |P \setminus P_0| \tag{2}$$

That is, we are not interested in the whole set of places since it is a siphon but it is an uninteresting one. Finally,

$$\forall \mathbf{Y} \in \mathcal{PS}, \sum_{p \in \mathbf{Y}} v_p < \|\mathbf{Y}\| \tag{3}$$

In this case, the selected set of places cannot be a P-Semiflow, since they are uninteresting siphons. P-Semiflows cannot be emptied because of the cyclic behavior law described above. Moreover, they are much less expensive from a computational point of view. For the Figure 1 and the set of equations shown there the assignment $v_{p\_0\_1} = v_{p\_1\_0} = v_{r\_0\_0} = v_{r\_0\_1} = 1$ is a solution and the set of places defined by them is a minimal siphon ($\{p\_0\_1$, p_1_0, r_0_0, $r\_0\_1\}$). It is easy to see that if we add $v_{p\_0\_0} = 1$ to the previous solution the equations remain true. This is one of the problems of this method: these equations can describe siphons, but they do not need to be minimal.



Set of equations:
Related to Equation 1:
$v_{p\_0\_0} \leq v_{r\_0\_0}$
$v_{p\_0\_1} \leq v_{p\_0\_0} + v_{r\_0\_1}$
$v_{p\_1\_0} \leq v_{p\_1\_1} + v_{r\_0\_0}$
$v_{p\_1\_1} \leq v_{r\_0\_1}$
$v_{r\_0\_0} \leq v_{p\_0\_0} + v_{r\_0\_1}$
$v_{r\_0\_0} \leq v_{p\_1\_0}$
$v_{r\_0\_1} \leq v_{p\_1\_1} + v_{r\_0\_0}$
$v_{r\_0\_1} \leq v_{p\_0\_1}$
Related to Equation 2:
$v_{p\_0\_0}+v_{p\_0\_1}+v_{p\_1\_0}+v_{p\_1\_1}+v_{r\_0\_0}+$
$v_{r\_0\_1} < 6$
Related to Equation 3:
$v_{p\_0\_0} + v_{p\_1\_0} + v_{r\_0\_0} < 3$
$v_{p\_0\_1} + v_{p\_1\_1} + v_{r\_0\_1} < 3$

Fig. 1. A very simple Petri net and the equations that represent its siphons. The idle places of the system have not been represented for the shake of brevity.

With this formulation we can construct a fitness function for the genetic algorithm that can guide the system towards a solution. As each variable can have a value of 0 or 1, this approach is well-suited to be formulated as a genetic algorithm. A final remark is that the genetic algorithm is an optimization algorithm

so some objective function is needed. We have decided to minimize the number of active variables. Since we cannot state by means of a simple logical formula the minimality property, we have chosen to compute the smaller siphons. We can imagine alternative objective functions that would take into account just the number of resource places, the number of process places, or some more complex measurements. The complete system would be:

$$\begin{array}{l} min \sum_{p \in P} v_p \\ \forall p \in P, \forall t \in {}^\bullet p, v_p \leq \sum_{q \in {}^\bullet t} v_q, v_p \in \{0,1\} \\ \sum_{p \in P \setminus P_0} v_p < |P \setminus P_0| \\ \forall \mathbf{Y} \in \mathcal{PS}, \sum_{p \in \mathbf{Y}} v_p < \|\mathbf{Y}\| \end{array} \tag{4}$$

Since we want to obtain a result that minimizes the function and that satisfies the restrictions we need to combine this information. When we have an individual which represents an empty siphon or a siphon composed by all the places of the net, we can return a negative number, equivalent to twice the number of restrictions (the idea is to help de GA to avoid these solutions). For the other restrictions, we can just count the number of places in the siphon when they are met. When there are unmet restrictions, we just return the difference between the number of such restrictions and the total number of restrictions (this is a negative number, that grows when more restrictions are met). We have tried several configurations giving more weight to the number of places in the siphon or to the number of satisfied restrictions but not significant differences appear.

5 The Experiments

We have compared the nets used in [25] as a benchmark of the performance of the methods. These nets belong to S^4PR class. It is a well–know subclass for the modeling of a wide set of RAS with a well-defined and easy to understand structure. Even the proposed method should allow us to look for siphons in any general PN, our previous work has concentrated in this class of nets and our examples belong to it. S^4PR nets allow the modeling of concurrent sequential processes with routing decisions and a general conservative use of resources.

There is a more detailed presentation of some of these models in [29, 25]. The first and second classes of systems are obtained by means of the composition of a set of sequential processes: each process, at each processing step, has attached a single (and different) resource. An instance of the Petri net representing two of such sequential processes of length two would follow the structure of the net in Figure 1 (only the resources for the first process are shown). There are two ways to study size variations in this family of systems: one of them is changing the length of the process; that is, the number of processing steps (two in the figure). The second one is changing the number of processes to be composed (in the figure two processes are shown). For the experiment, the sequential processes are composed with other processes according to the following rules: The first process shares its resources with the second one in reverse order: the resource used at the first step in the first process is used at the last step of the second

Table 1. Computing the minimal siphons with the algebraic methods

Name	Size	Number of siphons	[28]	[26]
FMSAD	3	42	0	0.07
	4	78	0.04	0.17
	5	150	0.69	0.72
	6	250	13.83	6.27
	7	490	466.95	84.54
	8	906	11127.44	1169.57
FMSLD	3	24	0	0.01
	4	54	0	0.02
	5	116	0.08	0.06
	6	242	2.27	0.12
	7	496	71.78	0.29
	8	1006	2570.88	0.87
Phil	3	10	0	0.02
	4	17	0.02	0.03
	5	26	0.34	0.04
	6	37	6.88	0.07
	7	50	291.58	0.08
	8	65	6930.71	0.13

Column 1: Name as in [25].
Column 2: Size of the problem (number of processes in FMSAD; length of two parallel processes in FMSLD; number of philosophers in Phil).
Column 3: Number of minimal siphons (computed by means of an algebraic algorithm).
Column 4: Time needed to compute all the siphons (in seconds) with the method described by Lautenbach in [28].
Column 5: Time needed to compute all the siphons (in seconds) with the method described by Boer and Murata in [26].

process; the resource used at the second step of the first process is used by the one that is previous to the last step in the second process, and so on. The second process is composed with the third one in a similar way and so on, until we reach the total number of composed processes. The last process is composed with the first one, in a similar way. Using the previous ideas two different families of S^4PR nets have been generated, labeled as *FMSAD* and *FMSLD* in the tables.

FMSAD nets are obtained by means of the composition of a variable number of sequential processes as the ones depicted in Figure 1, with a fixed length of 3. The number of processes to be composed in parallel is the parameter. In the experiments, the number of composed processes is varying from 3 to 8. *FMSLD* nets are obtained by means of the composition of a fixed number of two sequential processes as the ones depicted in Figure 1, with a variable length which is the parameter. In the experiments, this length is varying from 3 to 8.

The last one corresponds to an implementation of the well known dining philosophers problem. The parameter corresponds to the number of philosophers.

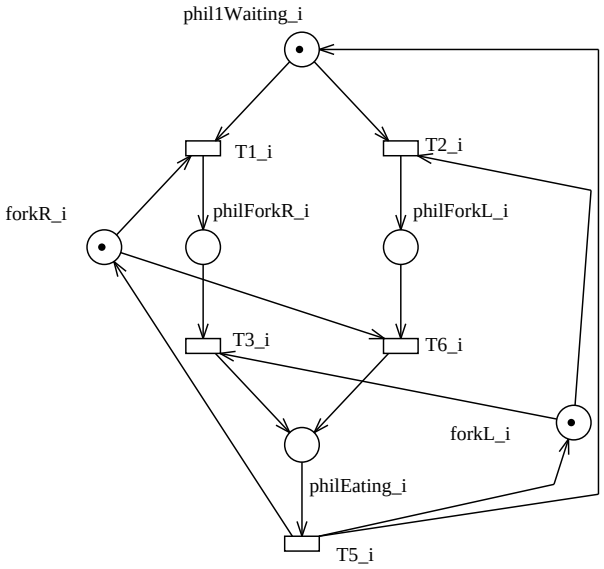


Fig. 2. Petri net model of the i–th philosopher

Figure 2 shows the model of the ith philosopher. Places $forkR\_i$ and $forkL\_i$ model, respectively, the state (free/engaged) of its right and left forks. The fork $forkR\_i$ will be shared with the philosopher on his right, and the $forkL\_i$ will be shared with the philosopher on his left. The results obtained for this family of nets are entitled *Phil* in the tables.

We have recomputed the values shown in [25] to include time results obtained with the same computer for all the experiments. This has been done with a desktop computer, Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz, with 4Gb of RAM. These results can be seen in Table 1. They are shown in a graphical way in the first column of Figure 3. Notice also that for each type of problem the next size takes more than 24 hours to finish the computation with the first method.

For these new experiments, we have measured the time used by the genetic algorithm to obtain at least one siphon; the GA should be able to compute more than one (just selecting the adequate set of best fit individuals) and we also measured this. In any case, it would be difficult to predict the number of good siphons, so let us use this time as a conservative measure.

The genetic algorithm has several parameters that need to be adjusted. We have used elitism and rank-based selection. For mutation we have used a bitblip operation with 33% of probability and we have selected a two-point crossover operator with probability of 66%. Then, we have concentrated on the size of the

initial population and the number of evaluations. We start the experiment for each example with an initial population of size 8 and we run the program thirty times; if it fails (does not compute a siphon) more than once, we double the size of the initial population and repeat until we can reach thirty iterations with at most one failed result. We also established a maximum number of evaluations: if no solution is found after this number of evaluations the algorithm stops (and we consider this run a failure).

We have tested two approaches for the initial population: First, introducing the P–Semiflows (when there are less P–Semiflows than the size of the initial population we add the needed individuals at random; when there are more, we add all of them and we complete the population until we reach a multiple of eight individuals). Second, using a fully random initial population. The reason for trying the first approach is that P–Semiflows could guide the algorithm toward interesting places in the net (in some classes of nets it is possible to construct them as a seed [30]). Notice that they cannot be part of the solution (they are explicitly forbidden, see equation (3) in Section 4).

When the algorithm stops, we can check whether the solution with best fitness is a siphon or not: if it has not positive fitness it won't be a siphon.

The results obtained can be seen in Table 2. They also can be seen in a graphical way in the second column of Figure 3 and in Figure 4. In the Figure 4 we have also included the standard deviation of the thirty runs of the program. We have included in the Figure 3 the graphics for the algebraic methods as a baseline. The times provided in the table and in the figures for the genetic algorithms are the average of the thirty runs of each experiment with the smaller acceptable initial population for each size of each problem.

We can see that the genetic algorithm is slower, in general, than the algebraic methods except for the case of FMSAD example, where the genetic algorithm seems to obtain its solution in less time (and it grows slowly if we compare with the algebraic method, which seems to grow exponentially).

There are two things to remark here: the results should not be compared directly, since the algebraic implementations where done in C, and the genetic algorithm has been programmed using Perl (an interpreted language). Nevertheless, putting the results together helps us to see that they are not so far away and that the approach can be adequate for some types of problems or when the size grows in such a way that it cannot be managed with deterministic methods. The second thing to note is that the genetic algorithm does not obtain all the siphons but a number of them (as the best fitted members of the final population).

In Table 2 we can see that there are no relevant differences in time when using the P–Semiflows as the initial population and when we use a random initial population. In the FMSAD example there is a small difference in the number of evaluations, which tend to be bigger (but the differences are small and there are cases when there are less evaluations with the random population -sizes 4, 5-). In the FMSLD example the number of evaluations tends to be lower for the initial random population (except for sizes 5,7). Finally, in the Phil example, the cases

Table 2. Times for siphon computation with the proposed method

			Initial Population			
			P–Semiflows		Random	
	Size	Pop.	Time	Eval.	Time	Eval.
FMSAD	3	64	0.93 (0.22)	912 (214.83)	0.96 (0.18)	938 (169.27)
	4	64	1.97 (0.33)	1,102 (185.03)	1.92 (0.32)	1,082 (185.82)
	5	64	3.97 (1.01)	1,448 (361.33)	3.33 (0.41)	1,222 (152.79)
	6	128	10.16 (1.19)	2,627 (305.12)	12.61 (12.70)	2,657 (394.46)
	7	256	31.07 (4.65)	5,877 (785.53)	31.71 (3.47)	5,954 (644.96)
	8	256	43.18 (4.65)	6,299 (677.50)	50.95 (17.79)	7,009 (1,231.68)
FMSLD	3	32	0.19 (0.03)	408 (72.73)	0.50 (1.73)	379 (73.60)
	4	32	0.37 (0.09)	458 (112.10)	0.36 (0.07)	447 (84.49)
	5	32	0.65 (0.10)	519 (79.26)	0.65 (0.10)	526 (79.82)
	6	32	1.01 (0.22)	571 (124.39)	1.00 (0.21)	562 (117.14)
	7	32	1.59 (0.34)	654 (137.96)	1.59 (0.85)	661 (343.29)
	8	64	4.14 (1.18)	1,349 (385.48)	4.05 (0.78)	1,310 (248.29)
Phil	3	64	0.33 (0.05)	790 (106.88)	0.34 (0.03)	812 (64.70)
	4	64	0.64 (0.06)	887 (92.53)	0.62 (0.07)	863 (91.17)
	5	128	2.02 (0.20)	1,852 (195.65)	2.03 (0.23)	1,859 (195.90)
	6	128	3.08 (0.35)	1,988 (217.09)	3.18 (0.38)	2,042 (240.61)
	7	128	4.77 (0.52)	2,260 (237.10)	4.60 (0.41)	2,185 (183.80)
	8	128	6.61 (0.60)	2,454 (221.54)	6.40 (0.67)	2,362 (234.06)

Column 1: Name (as in [25])
Column 2: Size of the problem.
Column 3: Population of the instance.
Column 4: Average time. P–Semiflows in the initial population. Random initial population.
Column 5: Average number of evaluations (rounded). P–Semiflows in the initial population.
Column 6: Average time. Random initial population.
Column 7: Average number of evaluations (rounded). Random initial population.
In all the cases, in parentheses, the standard deviation.

where the number of evaluations is better is the same for both initial types of initial population.

The results obtained show that the approach is suitable: we can compute (minimal) siphons with the proposed method. Comparing with traditional methods the genetic approach does not provide better time computation except for one example (but they are implementations in different languages) and the behavior is better with more complex problems (as one would expect). If we were interested in computing all the siphons, we could add the computed ones as negative restrictions (this set of places cannot be a solution, as we have done with P–Semiflows) and apply again the GA.

As another way to evaluate the approach we computed Table 3 where we can see the total number of different siphons obtained with the proposed method compared to the total number of siphons for each system. For this we have used

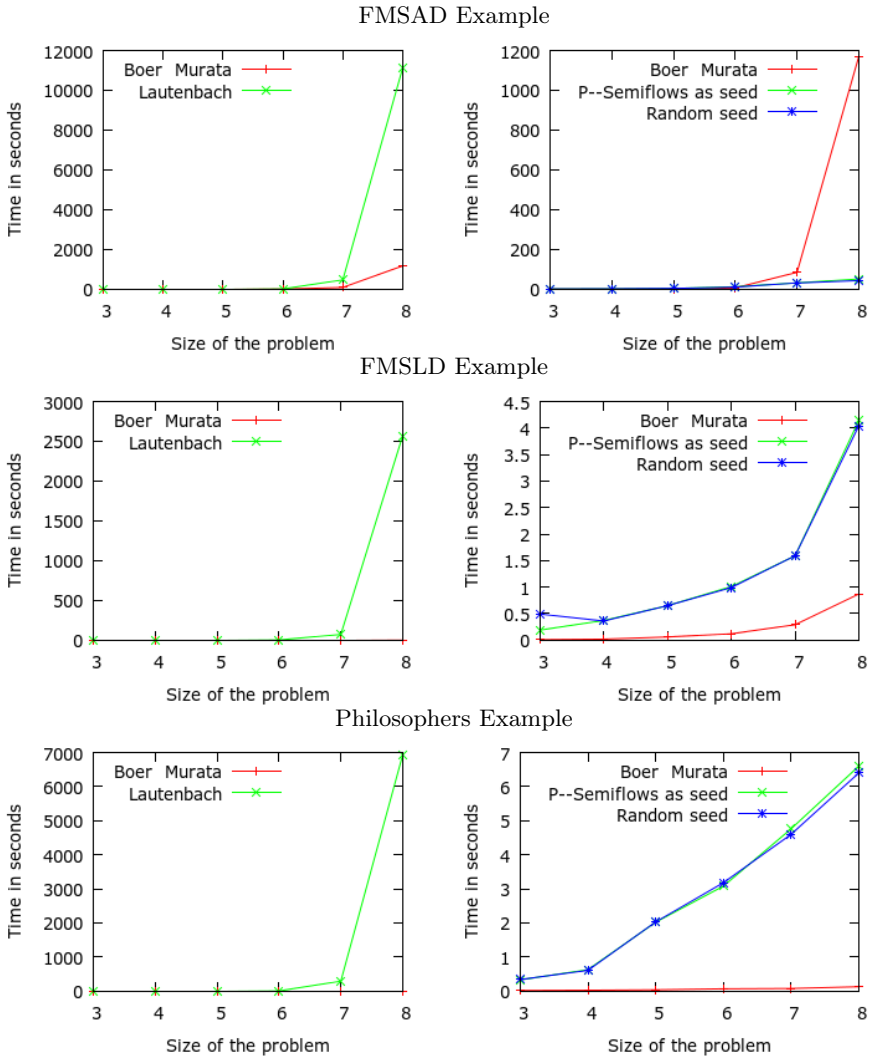


Fig. 3. Comparison of times for different examples and sizes

the same experiments as in the previous table: we can count the number of different siphons for each size of each problem in the 30 runs of the experiment. With this we can show that the genetic algorithm has a good behavior (different runs examine different parts of the solutions space) but we are not measuring what would happen with the addition of new restrictions to forbid siphons that have been computed previously. Moreover, when the size of the problem increases the method computes less siphons. Our feeling is that this is due to the size of the population (the size is small compared to the number of total siphons when

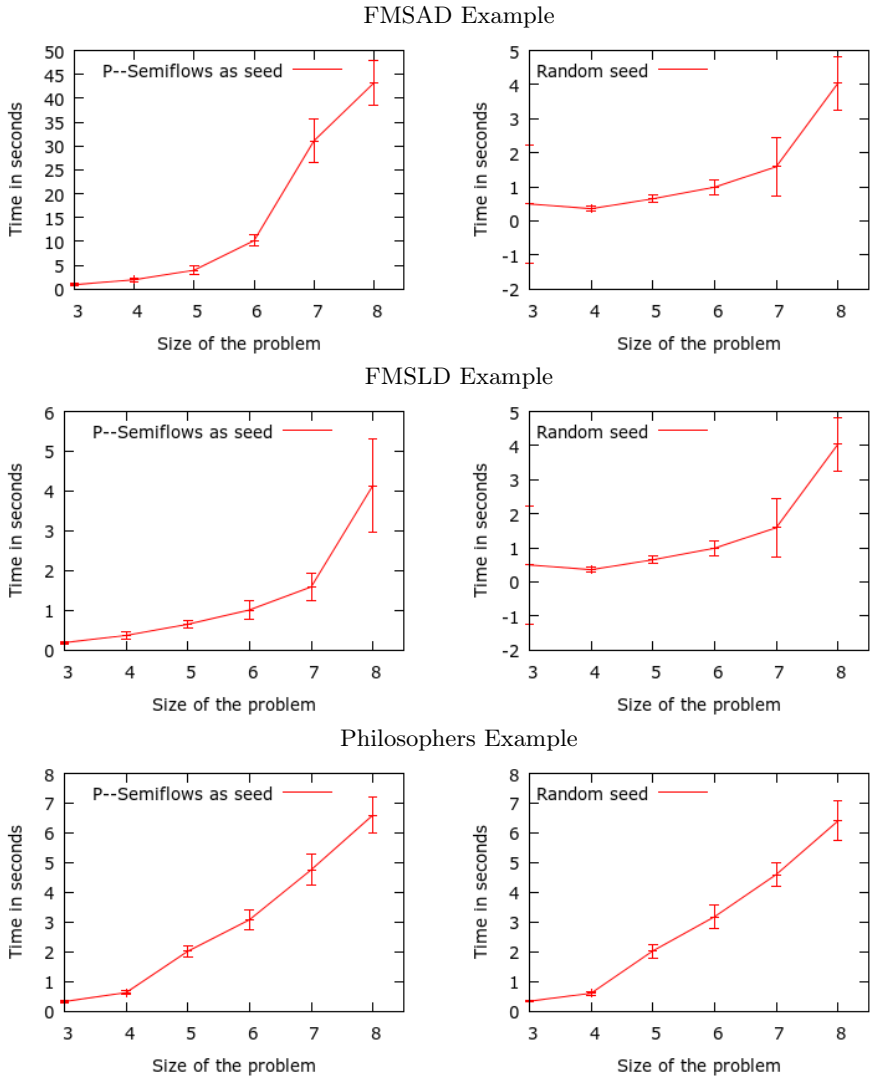


Fig. 4. Comparison of times for different examples and sizes

the size of the problem grows). For this reason we have added columns 6 and 7. There we can see that the number of different siphons computed increases with a bigger initial population. We can also see in that Table that the random initial population tends to produce more different siphons across different experiments.

Table 3. Number of siphons with different methods and populations

		Number of siphons	P	R	P	R	Percentage			
FMSAD	3	42	22	28	25	28	52.38%	66.67%	59.52%	66.67%
	4	78	29	32	34	42	37.18%	41.03%	43.59%	53.85%
	5	150	42	48	44	49	28%	32%	29.33%	32.67%
	6	250	48	53	55	68	19.20%	21.20%	22%	27.20%
	7	490	75	70	90	83	15.31%	14.29%	18.37%	16.94%
	8	906	59	67	110	78	6.51%	7.40%	12.14%	8.61%
FMSLD	3	24	14	11			58.33%	45.83%		
	4	54	28	32	32	36	51.85%	59.26%	59.26%	66.67%
	5	116	34	31	38	45	29.31%	26.72%	32.76%	38.79%
	6	242	31	37	43	48	12.81%	15.29%	17.77%	19.83%
	7	496	35	36	48	49	7.06%	7.26%	9.68%	9.88%
	8	1006	38	48	64	58	3.78%	4.77%	6.36%	5.77%
Phil	3	10	4	6	2	2	40%	60%	20%	20%
	4	17	5	5	5	8	29.41%	29.41%	29.41%	47.06%
	5	26	6	9	7	10	23.08%	34.62%	26.92%	38.46%
	6	37	10	12	10	12	27.03%	32.43%	27.03%	32.43%
	7	50	12	15	11	11	24%	30%	22%	22%
	8	65	11	15	10	13	16.92%	23.08%	15.38%	20%

Column 1: Name (as in [25])
Column 2: Size of the problem.
Columns 3: Number of siphons obtained by means of an algebraic algorithm.
Columns 4: Number of siphons obtained with the proposed method. P–Semiflows as initial population.
Column 5: Number of siphons obtained with the proposed method. Random initial population.
Columns 6-7: The same as columns 4-5 but doubling the size of the initial population.
Columns 8-11: The results of columns 4-7 as a percentage of Column 3.

6 Conclusions and further work

Some deadlock prevention control policies need the set of minimal siphons to be computed. It is well known that this is a very hard task because the number of such components can be very high. This paper has concentrated on the study of such question.

An adaptation of a set of logical formulas has been provided in order to try the genetic algorithm to search for a structural component of the net (siphons).

Even when it is not clear if the method has a good enough performance, it opens the door to further work. It is our intention to try to apply it to some deadlock prevention methods proposed in the past, where some special siphons need to be computed at each step.

Moreover, when the computation of all the siphons becomes prohibitively expensive, the genetic algorithm can still deal with bigger problems if it is acceptable for us to have a partial set of the siphons instead of the whole set provided by algebraic approaches.

In this sense, our proposal for further work will follow several ideas: First of all, the genetic algorithm is well suited for parallelization as in [25]. Second, the problem can be formulated not only in terms of siphon computation but in terms of a problem with more information. In the last years some ideas have been proposed in order to avoid the computation of all the minimal siphons. The methods rely on the computation of some special bad siphons together with bad markings (structural objects and bad states information is merged): if we introduce the state equation the genetic algorithm will have more information and, hopefully, it will be an alternative method to the one proposed in previously published work. [31]. Finally, we feel that adding more information about the siphon properties to the method (siphonosity?) it would work better. Other improvements for the GA need to be tested.

Acknowledgments

The authors are indebted to the anonymous referees and the PC who have helped us to improve the quality and presentation of this paper. This work is supported in part by project ANYSELF (TIN2011-28627-C04-02) by the Spanish Mineco and TIN2011-27479-C04-01 by the Spanish Ministry of Science and Innovation; P08-TIC-03903 awarded by the Andalusian Regional Government, project 83, Campus CEI BioTIC, and by Group of Discrete Event Systems Engineering (GISED) awarded by Aragonese Government.

References

1. Murata, T.: Petri nets: properties, analysis and applications. Proceedings of the IEEE **77**(4) (April 1989) 541–580
2. López-Grao, J.P., Colom, J.M.: A Petri Net Perspective on the Resource Allocation Problem in Software Engineering. Transactions on Petri Nets and Other Models of Concurrency V. Springer-Verlag, Berlin, Heidelberg (2012) 181–200
3. Liao, H., Wang, Y., Stanley, J., Lafortune, S., Reveliotis, S., Kelly, T., Mahlke, S.: Eliminating Concurrency Bugs in Multithreaded Software: A New Approach Based on Discrete-Event Control. Control Systems Technology, IEEE Trans. **PP**(99)
4. Liao, H., Lafortune, S., Reveliotis, S., Wang, Y., Mahlke, S.: Optimal Liveness-Enforcing Control for a Class of Petri Nets Arising in Multithreaded Software. Automatic Control, IEEE Transactions on **58**(5) (May 2013) 1123–1138
5. Ezpeleta, J., Colom, J., Martínez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Trans. Rob. Aut. **11**(2) (1995) 173–184
6. Barkaoui, K., Pradat-Peyre, J.: On Liveness and Controlled Siphons in Petri Nets. In Billington, J., Reisig, W., eds.: Proceedings of the 1996 International Conference on Aplications and Theory of Petri Nets, Springer Verlag (June 1996)
7. Tricas, F., García-Vallés, F., Colom, J., Ezpeleta, J.: An Iterative Method for Deadlock Prevention in FMS. In Boel, R., Stremersch, G., eds.: Discrete Event Systems: Analysis and Control. Proc. of WODES, Ghent, Belgium (2000) 139–148
8. Huang, Y., Jeng, M.D., Xie, Z., Chung, S.: Deadlock prevention policy based on Petri nets and siphons. Int. Journal of Production Research **39**(2) (2001) 283–305

9. Iordache, M.V., Moody, J.O., Antsaklis, P.: Synthesis of Deadlock Prevention Supervisors Using Petri Nets. IEEE Trans. Rob. Automat. **18**(1) (2002) 59–68
10. Li, Z., Zhou, M.C.: Elementary Siphons of Petri Nets and Their Applications to Deadlock Prevention in Flexible Manufacturing Systems. IEEE Trans. on Systems, Man, and Cybernetics **34**(1) (January 2004) 38–51
11. Barkaoui, K., Chaoui, A., Zouari, B.: Supervisory Control of Discrete Event Systems Based on Structure of Petri Nets. In: Proceedings of the 1997 IEEE International Conference on Systems, Man and Cybernetics. Computational Cybernetics and Simulation, Orlando, Florida, USA, IEEE (October 1997) 3750–3755
12. Tricas, F., Colom, J., Ezpeleta, J.: A solution to the problem of deadlocks in concurrent systems using Petri nets and integer linear programming. In Horton, G., Moller, D., Rude, U., eds.: Proc. of the 11th European Simulation Symposium, Erlangen, Germany, The society for Computer Simulation International (oct 1999)
13. Li, S., Li, Z., Hu, H., Al-Ahmari, A., An, A.: An extraction algorithm for a set of elementary siphons based on mixed-integer programming. Journal of Systems Science and Systems Engineering **21**(1) (March 2012) 106–125
14. Jeng, M., Peng, M., Huang, Y.: An algorithm for calculating minimal siphons and traps of Petri nets. Int. J. of Intelligent Control and Systems **3**(3) (1999) 263–275
15. Barkaoui, K., Lemaire, B.: An effective characterization of minimal deadlocks and traps in petri nets based on graph theory. In: Proceedings of the 10th International Conference on Application and Theory of Petri Nets, 1989. (1989) 1–21
16. Tricas, F.: Deadlock Analysis, Prevention and Avoidance in Sequential Resource Allocation Systems, Ph.D. Thesis. Dep. Inf. e Ing. de Sist. U. Zaragoza (May 2003)
17. Cordone, R., Ferrarini, L., Piroddi, L.: Characterization of minimal and basis siphons with predicate logic and binary programming. In: IEEE Int. Symposium on Computer Aided Control System Design, 2002, IEEE (2002) 193–198
18. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Oxford, England: U Michigan Press (1975)
19. Merelo, J.J.: A Perl Primer for Evolutionary Algorithm Practitioners. SIGEVOlution **4**(4) (March 2010) 12–19
20. Merelo-Guervós, J.J., Castillo, P.A., Alba, E.: `Algorithm::Evolutionary`, a flexible Perl module for evolutionary computation. Soft Computing **14**(10) (2010) 1091–1109 Accesible at http://sl.ugr.es/000K [sl.ugr.es].
21. Prashant Reddy, J., Kumanan, S., Krishnaiah Chetty, O.V.: Application of Petri Nets and a Genetic Algorithm to Multi-Mode Multi-Resource Constrained Project Scheduling. The Int. J. of Advanced Manufacturing Tech. **17**(4) (2001) 305–314
22. Lim, A.H.L., Lee, C.S., Raman, M.: Hybrid genetic algorithm and association rules for mining workflow best practices. Expert Systems with Applications **39**(12) (September 2012) 10544–10551
23. Xing, K., Han, L., Zhou, M., Wang, F.: Deadlock-Free Genetic Scheduling Algorithm for Automated Manufacturing Systems Based on Deadlock Control Policy. Systems, Man, and Cybernetics, Part B, IEEE Trans. **42**(3) (2012) 603–615
24. Han, L., Xing, K., Chen, X., Lei, H., Wang, F.: Deadlock-free genetic scheduling for flexible manufacturing systems using Petri nets and deadlock controllers. International Journal of Production Research **52**(5) (October 2013) 1557–1572
25. Tricas, F., Ezpeleta, J.: Computing minimal siphons in Petri net models of resource allocation systems: a parallel solution. Sys. Man Cyber. Part A: Systems and Humans, IEEE Trans. on **36**(3) (2006) 532–539

26. Boer, E.R., Murata, T.: Generating basis siphons and traps of Petri nets using the sign incidence matrix. IEEE Trans. on Circuits and Systems, I – Fundamental Theory and Applications **41**(4) (1994) 266–271
27. Silva, M.: Las Redes de Petri en la Automática y la Informática. Ed. AC, Madrid (1985)
28. Lautenbach, K.: Linear algebraic calculation of deadlocks and traps. In Voss, K., Genrich, H., Rozemberg, G., eds.: Concurrency and Nets. Springer Verlag (1987) 315–336
29. Tricas, F., Ezpeleta, J.: RessAllocation Petri net Model. In Kordon, F., et al., eds.: Model Checking Contest 2013, Milano, Italy (June 2013)
30. Cano, E.E., Rovetto, C.A., Colom, J.M.: An algorithm to compute the minimal siphons in S^4PR nets. Discrete Event Dynamic Systems **22**(4) (2012) 403–428
31. Tricas, F., García-Vallés, F., Colom, J., Ezpeleta, J.: A Petri Net Structure–Based Deadlock Prevention Solution for Sequential Resource Allocation Systems. In: Proc of 2005 Int. Conf. on Robotics and Automation, Barcelona, Spain (2005) 272–278

Part IV

PNSE'14: Short Papers

Persistency and Nonviolence Decision Problems in P/T-nets with Step Semantics*

Kamila Barylska
kamila.barylska@mat.umk.pl

Faculty of Mathematics and Computer Science, Nicolaus Copernicus University,
Chopina 12/18, 87-100 Toruń, Poland

Abstract. Persistency is one of the notions widely investigated due to its application in concurrent systems. The classical notion refers to nets with a standard sequential semantics. We will present two approaches to the issue (nonviolence and persistency). The classes of different types of nonviolence and persistency will be defined for nets with step semantics. We will prove that decision problem concerning all the defined types are decidable.

1 Introduction

The notion of persistency has been extensively studied for past 40 years, as a highly desirable property of concurrent systems. A system is persistent (in a classical meaning) when none of its components can be prevented from being executed by other components. This property is often needed during the implementation of systems in hardware [3]. The classical notion can be split into two notions: persistency (no action is disabled by another one) and nonviolence (no action disables another one).

The standard approach to Petri nets provides a sequential semantics - only single actions can be executed at a time. We choose a different semantics (real concurrency), in which a step, that is a set of actions, can be executed simultaneously as a unique atom of a computation.

In [6] different types of persistency and nonviolence notions for p/t-nets with step semantics were presented. In [1] levels of persistency were introduced for nets with sequential semantics. In this paper we combine both approaches and define classes (not only enabling-oriented but also life-oriented) of nonviolence and persistency of different types for nets with step semantics. We prove that all defined kinds of nonviolence and persistency are decidable for place/transition nets.

* This research was supported by the National Science Center under the grant No.2013/09/D/ST6/03928.

2 Basic definitions and denotations

We assume that basic notions concerning Petri Nets are known to the reader. Their definitions are omitted here due to the page limit, and can be found in any monograph or survey about Petri Nets.

Classical p/t-nets provide a sequential semantics of action's executions. It means that only one action can be executed as a single atom of a computation. In this paper we assume a different semantics: subset of actions called *steps* can be enabled and executed as an atomic operation of a net. Basic definitions concerning p/t-nets adapted to nets with step semantics can be found in [6]. All the definitions and facts required in the paper are presented in its longer version and posted on the author's website[1].

3 The Monoid $\mathbb{N}^{\mathbf{k}}$

See [1] for definitions and facts concerning the monoid $\mathbb{N}^{\mathrm{k}}$, rational subsets of $\mathbb{N}^{\mathrm{k}}$, ω-vectors, sets of all minimal/maximal members of X ($Min(X)/Max(X)$), and closures, convex sets, bottom and cover.

4 Levels of persistency and nonviolence

In [1] one can find definitions of three classes of persistency for nets with sequential semantics: the first one (corresponding to the classical notion): "no action can disable another one", and two ways of generalization of this notion: "no action can kill another one" and "no action can kill another enabled one".

In [6] a thorough analysis of persistent nets with step semantics was conducted. It was pointed out there that the existing concept of persistency [7] can be separated into two concepts, namely nonviolence (previously called persistency in [1]) and persistency (or robust persistency).

It is shown there that one can consider three classes of nonviolence and persistency steps: A - where after the execution of one step we take into consideration only the remaining part of the other step, B - if two steps do not have any common action, then after the execution of one of them we consider the whole second step, and C - after the execution of one step we take into account the whole second step. As it is proved in [6], the notions of A and B persistency (nonviolence, respectively) steps coincide, in the remaining we will only consider the classes of A and C steps.

[1] www.mat.umk.pl/~khama/Barylska-PersistencyAndNonviolenceDecisionProblems.pdf

Let us define classes of both types persistency and nonviolence with step semantics.

Definition 1. *Let* $\mathrm{S} = (P, T, W, M_0)$ *be a place/transition net. For* $M \in [M_0\rangle$ *and steps* $\alpha, \beta \subseteq T$*, such that* $\alpha \neq \beta$*, the step* α *in* M *is:*

- A-e/e-nonviolent *iff* $M\alpha \wedge M\beta \Rightarrow M\alpha(\beta \setminus \alpha)$
- A-l/l-nonviolent *iff* $M\alpha \wedge (\exists u)Mu\beta \Rightarrow (\exists v)M\alpha v(\beta \setminus \alpha)$*, where* $u, v \in (2^T)^*$
- A-e/l-nonviolent *iff* $M\alpha \wedge M\beta \Rightarrow (\exists v)M\alpha v(\beta \setminus \alpha)$*, where* $v \in (2^T)^*$
- A-e/e-persistent *iff* $M\alpha \wedge M\beta \Rightarrow M\beta(\alpha \setminus \beta)$
- A-l/l-persistent *iff* $(\exists u)Mu\alpha \wedge M\beta \Rightarrow (\exists v)M\beta v(\alpha \setminus \beta)$*, where* $u, v \in (2^T)^*$
- A-e/l-persistent *iff* $M\alpha \wedge M\beta \Rightarrow (\exists v)M\beta v(\alpha \setminus \beta)$*, where* $v \in (2^T)^*$
- C-e/e-nonviolent *iff* $M\alpha \wedge M\beta \Rightarrow M\alpha\beta$
- C-l/l-nonviolent *iff* $M\alpha \wedge (\exists u)Mu\beta \Rightarrow (\exists v)M\alpha v\beta$*, where* $u, v \in (2^T)^*$
- C-e/l-nonviolent *iff* $M\alpha \wedge M\beta \Rightarrow (\exists v)M\alpha v\beta$*, where* $v \in (2^T)^*$
- C-e/e-persistent *iff* $M\alpha \wedge M\beta \Rightarrow M\beta\alpha$
- C-l/l-persistent *iff* $(\exists u)Mu\alpha \wedge M\beta \Rightarrow (\exists v)M\beta v\alpha$*, where* $u, v \in (2^T)^*$
- C-e/l-persistent *iff* $M\alpha \wedge M\beta \Rightarrow (\exists v)M\beta v\alpha$*, where* $v \in (2^T)^*$

Let $\mathrm{S} = (P, T, W, M_0)$ be a place/transition net and $M \in [M_0\rangle$.
We say that a marking M is [A/C]-[(e/e)/(l/l)/(e/l)]-[persistent/nonviolent] iff the step α in M is [A/C]-[(e/e)/(l/l)/(e/l)]-[persistent/nonviolent] for every enabled $\alpha \subseteq T$.
We say that the net S is [A/C]-[(e/e)/(l/l)/(e/l)]-[persistent/nonviolent] iff every reachable marking $M \in [M_0\rangle$ is [A/C]-[(e/e)/(l/l)/(e/l)]-[persistent/nonviolent].
The classes of [A/C]-[(e/e)/(l/l)/(e/l)]-[persistent/nonviolent] p/t-nets will by denoted by $\mathbf{P}_{[\mathbf{A/C}]-[\mathbf{(e/e)/(l/l)/(e/l)}]-[\mathbf{p/n}]}$.

5 Decision Problems

Let us recall the famous decidable (Mayr [8], Kosaraju [5]) problem called **Marking Reachability Problem**:

Instance: A p/t-net $\mathrm{S} = (P, T, W, M_0)$, and a marking $M \in \mathbb{N}^{|P|}$.
Question: Is M reachable in S?

Let us formulate a more general **Set Reachability Problem**

Instance: A p/t-net $\mathrm{S} = (P, T, W, M_0)$, and a set $X \subseteq \mathbb{N}^{|P|}$.
Question: Is there a marking $M \in X$, reachable in S?

Theorem 1 ([1]). *If* $X \subseteq \mathbb{N}^k$ *is a rational convex set, then the* X*-Reachability Problem is decidable in the class of p/t-nets.*

In order to formulate precisely decision problems concerning classes of nonviolence and persistency types described in definition 1, let us define the following sets of markings (for a given steps α and β):

$E_\alpha = \{M \in \mathbb{N}^k \mid M\alpha\}$ $E_\beta = \{M \in \mathbb{N}^k \mid M\beta\}$
$E_{\alpha\beta} = \{M \in \mathbb{N}^k \mid M\alpha\beta\}$ $E_{\beta\alpha} = \{M \in \mathbb{N}^k \mid M\beta\alpha\}$
$E_{\alpha(\beta\setminus\alpha)} = \{M \in \mathbb{N}^k \mid M\alpha(\beta \setminus \alpha)\}$ $E_{\beta(\alpha\setminus\beta)} = \{M \in \mathbb{N}^k \mid M\beta(\alpha \setminus \beta)\}$
$E_{..\alpha} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)Mw\alpha\}$ $E_{..\beta} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)Mw\beta\}$
$E_{..(\alpha\setminus\beta)} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)Mw(\alpha \setminus \beta)\}$
$E_{..(\beta\setminus\alpha)} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)Mw(\beta \setminus \alpha)\}$
$E_{\alpha..\beta} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)M\alpha w\beta\}$
$E_{\beta..\alpha} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)M\beta w\alpha\}$
$E_{\alpha..(\beta\setminus\alpha)} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)M\alpha w(\beta \setminus \alpha)\}$
$E_{\beta..(\alpha\setminus\beta)} = \{M \in \mathbb{N}^k \mid (\exists w \in (2^T)^*)M\beta w(\alpha \setminus \beta)\}$

Remark:
Let us note the following equalities:
$E_\alpha = en\alpha + \mathbb{N}^k$
$E_\beta = en\beta + \mathbb{N}^k$
$E_{\alpha\beta} = max(en\alpha, en\alpha - ex\alpha + en\beta) + \mathbb{N}^k$
$E_{\beta\alpha} = max(en\beta, en\beta - ex\beta + en\alpha) + \mathbb{N}^k$
$E_{\alpha(\beta\setminus\alpha)} = max(en\alpha, en\alpha - ex\alpha + en(\beta \setminus \alpha)) + \mathbb{N}^k$
$E_{\beta(\alpha\setminus\beta)} = max(en\beta, en\beta - ex\beta + en(\alpha \setminus \beta)) + \mathbb{N}^k$
where $en\alpha$ and $ex\alpha$ are *vectors* of *entries* and *exits* of α .

Thanks to the equalities it is easy to find a rational expressions for the listed sets. It is much more difficult to find rational expressions for the rest of the markings listed above.

Let us notice that a step is not nonviolent/persistent in a distinct sense when a certain "unwanted" marking is reachable in a given net. It is easy to see, that we can connect the above sets of markings with classes of nonviolent/persistent steps. The table below shows the connections.

Z=Nonviolence			Z=Persistency		
X	Y	E_{X-Y-Z}	X	Y	E_{X-Y-Z}
A	EE	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\alpha(\beta\setminus\alpha)})$	A	EE	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta(\alpha\setminus\beta)})$
A	LL	$E_\alpha \cap E_{..\beta} \cap (\mathbb{N}^k \setminus E_{\alpha..(\beta\setminus\alpha)})$	A	LL	$E_{..\alpha} \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta..(\alpha\setminus\beta)})$
A	EL	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\alpha..(\beta\setminus\alpha)})$	A	LL	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta..(\alpha\setminus\beta)})$
C	EE	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\alpha\beta})$	C	EE	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta\alpha})$
C	LL	$E_\alpha \cap E_{..\beta} \cap (\mathbb{N}^k \setminus E_{\alpha..\beta})$	C	LL	$E_{..\alpha} \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta..\alpha})$
C	EL	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\alpha..\beta})$	C	LL	$E_\alpha \cap E_\beta \cap (\mathbb{N}^k \setminus E_{\beta..\alpha})$

Denotation: $US = \{E_{\{X-Y-X\}} \mid X \in \{A, C\}, Y \in \{EE, LL, EL\}, Z \in \{\text{N/P}\}\}$[2]
- the set of *undesirable sets*.

[2] where N=Nonviolence and P=Persistency

Now we are ready to formulate the decision problems. Let us notice that a particular decision problem is decidable when we can settle whether any marking from the undesirable set associated to the problem is reachable.

***X-Y-Z* Problem:** (for $X \in \{A, C\}, Y \in \{EE, LL, EL\}, Z \in \{\text{N/P}\}$)
Instance: A p/t-net $\text{S} = (P, T, W, M_0)$, and steps $\alpha, \beta \subseteq T$.
Question: Is the set E_{X-Y-Z} reachable in S?

Informally, if any marking from the set E_{X-Y-Z} is reachable in S, some "unwanted" situation takes place, for example when$X - Y - Z = A - EL - \text{P}$, then it means that a subset $(\alpha \setminus \beta)$ of an enabled α is killed by the execution of β . Such a situation is depicted in Fig.1 with $\alpha = \{a, c\}$, $\beta = \{a, b\}$, and $(\alpha \setminus \beta) = \{c\}$. One can easily see, that $M\alpha$ and $M\beta$. Let $M' = M\beta$, then the step $(\alpha \setminus \beta)$ is dead in M'.

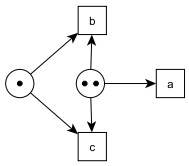


Fig. 1. Not A-EL-P p/t-net.

Theorem 2. *The Decision Problems described above are decidable in the class of p/t-nets.*

Sketch of the proof:

1. We put into work the theory of residual sets of Valk/Jantzen [9] and thanks to their results we show that Bottoms (the set of all minimal members) of the sets $E_{..\alpha}$, $E_{..\beta}$, $E_{..(\alpha\setminus\beta)}$, $E_{..(\beta\setminus\alpha)}$, $E_{\alpha..\beta}$, $E_{\beta..\alpha}$, $E_{\alpha..(\beta\setminus\alpha)}$, $E_{\beta..(\alpha\setminus\beta)}$ are effectively computable.
2. We obtain rational expressions for the sets as follows:
$E_X = \text{Bottom}(E_X) + \mathbb{N}^k$, where $X \in \{..\alpha, ..\beta, ..(\alpha \setminus \beta), ..(\beta \setminus \alpha), \alpha..\beta, \beta..\alpha, \alpha..(\beta \setminus \alpha), \beta..(\alpha \setminus \beta)\}$.
3. Using the Ginsburg/Spanier Theorem [4], which says that rational subsets of $\mathbb{N}^k$ are closed under union, intersection and difference we compute rational expressions for the undesirable sets. Let us notice that undesirable sets are convex.
4. We check whether any marking from the undesirable set connected to a distinct decision problem is reachable in a given net. The Theorem 1 yields decidability of all the problems.

Let us now formulate net-oriented versions of the problems.

The Net Nonviolence and Persistency Problems
Instance: A p/t-net S = (P, T, W, M_0).
Question:
[A/C**]-[**$EE/LL/EL$**]-Net-[Nonviolence/Persistency] Problem:**
Is the net [A/C]-[(e/e)/(l/l)/(e/l)]-[nonviolent/persistent]?

Of course the problems are decidable, as it is enough to check an adequate transition oriented problem for every pair of steps.

6 Plans for Further Investigations

- In [1] inclusions between defined there kinds of nonviolence (called there persistency) were investigated. One can examine the relationships between the presented in definition 1 types of nonviolent and persistent nets.
- In [2] levels of e/l-k-persistency were defined. It would be useful to investigate the notions in p/t-nets with step semantics.

References

1. Kamila Barylska and Edward Ochmanski. Levels of persistency in place/transition nets. *Fundam. Inform*, 93(1-3):33–43, 2009.
2. Kamila Barylska and Edward Ochmanski. Hierarchy of persistency with respect to the length of actions disability. *Proceedings of the International Workshop on Petri Nets and Software Engineering, Hamburg, Germany*, pages 125–137, 2012.
3. Johnson Fernandes, Maciej Koutny, Marta Pietkiewicz-Koutny, Danil Sokolov, and Alex Yakovlev. Step persistence in the design of gals systems. *Lecture Notes in Computer Science*, 7927:190–209, 2013.
4. Seymour Ginsburg and Edwin H. Spanier. Bounded algol-like languages. *TRANS AMER MATH SOC*, (113):333–368, 1964.
5. S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 267–281, New York, NY, USA, 1982. ACM.
6. Maciej Koutny, Lukasz Mikulski, and Marta Pietkiewicz-Koutny. A taxonomy of persistent and nonviolent steps. *Lecture Notes in Computer Science*, 7927:210–229, 2013.
7. Lawrence H. Landweber and Edward L. Robertson. Properties of conflict-free and persistent petri nets. *JACM: Journal of the ACM*, 25, 1978.
8. Ernst W. Mayr. An algorithm for the general petri net reachability problem. *SIAM J. Comput.*, 13(3):441–460, 1984.
9. Rudiger Valk and Matthias Jantzen. The residue of vector sets with applications to decidability problems in petri nets. *ACTAINF: Acta Informatica*, 21, 1985.

Part V

PNSE'14: Poster Abstracts

Construction of Data Streams Applications from Functional, Non-Functional and Resource Requirements for Electric Vehicle Aggregators. The COSMOS Vision

J.A. Bañares, R. Tolosana-Calasanz, F. Tricas, U. Arronategui, J. Celaya, and J.M. Colom

Depto de Informática e Ingeniería de Sistemas - Universidad de Zaragoza
banares@unizar.es

Abstract. COSMOS, Computer Science for Complex System Modeling, is a research team that has the *mission* of bridging the gap between formal methods and real problems. The goal is twofold: (1) a better management of the growing complexity of current systems; (2) a high quality of the implementation reducing the time to market. The COSMOS *vision* is to prove this approach in non-trivial industrial problems leveraging technologies such as software engineering, cloud computing, or workflows. In particular, we are interested in the technological challenges arising from the Electric Vehicle (EV) industry, around the EV-charging and control IT infrastructure.

Keywords: Continuous Data Streams, Executable High-Level Specification, Scheduling, Non-functional specifications, Distributed Systems, Resource Allocation Systems, Cloud Computing, Smart Grid, Electric Vehicle

1 Introduction

Electric Vehicles (EVs) give rise to computational challenging problems. EVs will be plugged into the electric infrastructure (distribution networks) for the re-charging of their batteries, and they will share capacity with other users of electricity. On other hand, electricity distribution and generation infrastructures are currently turning into Smart Grids for more efficient management. From a computational point of view, this involves demand forecast methods, state estimation techniques, and real-time monitoring, leading to communication and control of residential and commercial areas taking place at a fine granularity. Critical characteristics of the involved computational problems are related to large volumes of data being generated in a distributed, stream fashion, and real-time –which is often agnostic of demand variation in a given geographical area. The main scientific and technical objectives are: 1) The development of a methodology for the construction of applications for Continuous Data Streams Processing. It must cover all Software Engineering phases of the life cycle, and

must be able to address functional and non-functional requirements together with the specification of the execution infrastructure and the involved resources. 2) The previous methodology requires the definition of an executable specification language across all the software architectural levels. It will support modular and hierarchical specification of these types of applications, together with the associated tools, analysis, verification, simulation, implementation, execution and monitoring. A set of mechanisms, taking into account different architectural configurations that can be used in the implementation, will be designed to support the studied policies and mechanisms. 3) Analysis, design and development of a proof of concept Autonomic Smart Energy Management System for Electric Vehicles infrastructure management.

2 Expected Scientific Impact

Recent studies estimate that uncontrolled re-charging processes of EV batteries can lead to significant increase in the electricity demand peaks. Moreover, they anticipate that EVs will impact the local level, where hotspots will be created depending on how EVs cluster within a particular geographical location. These hotspots may eventually overload the low voltage distribution networks. The re-charging of EV batteries will generate large-scale volumes of information, in a distributed fashion, that need to be processed. This project looks for computational solutions to manage the large amounts of re-charging information coming from the EVs. It uses a single computational infrastructure that enforces an established Service Level Agreement, while adapting the computational power for the processing of the information. From a socioeconomic point of view, the expected impact is: (1) The computational solutions can be exploited for the managing of EVs in the electric infrastructure. Large-amounts of information streamed by smart meters can be required to be processed in real-time when EVs re-charge their batteries. (2) As a result, there will be a substantial reduction of computational resources and, as a consequence, of energy in their management as well. Additionally, a substantial reduction of human intervention is expected, saving costs to companies. (3) The project falls within the social challenge 4 of the Horizon 2020 program of the European Union: "Smart, Green and Integrated Transport." (4) The results of the research shall also apply to decision-making systems of healthcare systems. (5) The results of the research may also apply to different data processing applications with real-time needs, that can be of interest for highly qualified SMEs, enabling them to become a media company to data process.

3 Collaboration

In this endeavour, we are looking for partners that can help us to construct solid collaboration structures in order to apply for an European project proposal within the EU Horizon 2020 framework.

Modular Modeling of SMIL Documents with Complex Termination Events

Djaouida Dahmani[1], Samia Mazouz[2], and Malika Boukala[1]

[1] MOVEP, USTHB, Algiers.
dzaouche,mboukala@usthb.dz,
[2] LSI, USTHB, Algiers.
smazouz@usthb.dz

Abstract. In order to design and analyse complex real time systems, we improve the communication mechanism of Time Recursive Petri Net model that we have proposed in previous works. We have used our extended model, named Time $ERPN^+$, to check the temporal coherence of SMIL documents. This paper presents on-going work where special attention is given to the **termination events** between objects having distinct time references.

This paper summarises and extends some previous results of our work aiming at controlling the temporal coherence of SMIL documents by using models based on Time Recursive Petri net (Time RPN) [1]. This latter is an extension of Recursive Petri net (RPN) with time by proposing a formal methodology for the design and validation of component-based real-time systems with dynamic structure, namely Time RPN. The modularity is inherent in Time RPN without any extended notation. Afterwards, we have refined Time RPN model by improving the communication mechanism between threads; the proposed version is called Time $ERPN^+$. In Time RPN, each thread has its own local execution context (local place marking). No communication between threads is possible, except birth and death relationship. When a thread T terminates, it aborts its whole descent of threads. Only T may return results to its father-thread (which gave birth to T). Therefore, all aborted threads have a silent death, except T. Whereas in Time $ERPN^+$, communications between threads are enriched by means of shared context (global places). Furthermore, any thread has the ability to report its death, contrary to Time RPN, where some threads have silent death. Moreover, finitude, accessibility and boundness of Time $ERPN^+$ are decidable.

A multimedia document SMIL is a collection of media. A media may be (i) basic as an image, a video, a text or an hyper-link, or (ii) composite as a *par* (resp. *seq*) object which plays in parallel (resp. seq) a set of media, considered as its children. The temporal behaviour of a media is described by a set of temporal attributes. For instance, *begin* and *end* attributes define respectively the beginning and ending times of a media; it can be a known value or a synchronisation

event (e.g. $end(media1) = end(media2)$ means that $media1$ must terminate when $media2$ terminates). Author, who is in charge of creating such document, can incrementally add, modify or remove any temporal relation. Therefore, his document may become incoherent. Research in the field of temporal consistency verification of multimedia documents covers several aspects : verification at authoring stage, designing schedulers which handle synchronisation at player level and so on. Our approach focuses on temporal consistency verification of multimedia documents at authoring stage [2].

A SMIL document is translated into a Time $ERPN^+$, used to check its temporal coherence. To each SMIL media (basic or composite), a time *sub-*$ERPN^+$ is associated as a simulation support. Furthermore, the transitions of the sub-net are constrained by temporal intervals which are deduced from the temporal attributes of the media. Also an abstract transition is attached to the media and viewed as its interface. A firing of the media interface allows to launch one media simulation occurrence by creating a thread which will play such a simulation. In a previous work [2], we have modeled *simple* synchronisation **event** between children of a same parent media. For this purpose, local places are used within the sub-net associated with the parent media. Furthermore a simple synchronisation **event** is played by the parent thread and no need to global places. In second time, we have tackled media having distinct parents but we have only considered relations such as $begin(media1) = begin(media2) + t$ or $begin(media1) = end(media2) + t$. We have shown that the time $ERPN^+$ model is more suitable for such a synchronisation, characterized as complex, in fact it allows to maintain the compactness of our modeling [3]. In this paper, we consider terminate events such as $end(media1) = begin(media2) + t$ or (ii) $end(media1) = end(media2) + t$ and explain briefly its modeling. It consists in adding a time *sub-*$ERPN^+$ which is played by the initial thread. The parent thread of $media2$ puts a token in a global place as soon as $media2$ (i) starts or (ii) terminates. After t time units, the initial thread marks a second global place which will force the end of the thread playing $media1$. Thanks to global places and the new communication concept of time $ERPN^+$, our modeling is modular and reflects the structure of a SMIL document.

References

1. D. Dahmani, JM. Ilié, and M. Boukala. Time recursive PetriNets. In *Transactions on Petri Nets and Other Models of Concurrency*, volume 1, pages 104–118. Springer Verlag, 2008.
2. D. Dahmani, S. Mazouz, and M. Boukala. Modular Modeling and Analyzing of Multimedia Documents with Repetitive Objects. In *5th International Conference on Computer Science and Information Technology (CSIT'13)*, pages 308–316, Amman, Jordan, March 2013. IEEEXplore digital library.
3. D. Dahmani, S. Mazouz, and M. Boukala. Efficient and Modular Modeling of Hierarchical Multimedia Objects by Using Time ERPN+. In *4th International Conference on Multimedia Computing and Systems (ICMCS'14)*, Marrakesh, Morocco, April 2014. IEEE Catalog Number : CFP-14050-CDR.

D&A4WSC as a Design and Analysis Framework of Web Services Composition

Rawand Guerfel and Zohra Sbaï

Université de Tunis El Manar, Ecole Nationale d'Ingénieurs de Tunis,
BP. 37 Le Belvédère, 1002 Tunis, Tunisia
guerfel.rawand@gmail.com, zohra.sbai@enit.rnu.tn

Abstract. The Web services composition (WSC) has an enormous potential for the organizations in the B2B area. In fact, different services collaborate through the exchange of messages to implement complex business processes. BPEL is one of the most used languages to develop such cooperation. However, a composition is not with added value if it is not compatible. This property garantees a placement of a correct composite web service. In this context, we develop a verification approach of the WSC compatibility. We propose, hence, a framework named D&A4WSC which allows to model the WSC by oWF-nets, to check their compatibility with the model checker NuSMV and to translate them if they are compatible in BPEL processes using the oWFN2BPEL compiler.

Keywords: Open workflow nets, Model checking, Compatibility, BPEL

Web services are exposed to consumers through standard interfaces. In general, because of the complexity of the demand of the consumer, a single service cannot reach this request and it therefore needs to contact one or more other services. This corresponds to Web services composition which allows multiple services to communicate and collaborate by exchanging messages and thus to implement a composite service that can perform complex tasks for consumers.

Several languages have been proposed to ensure this composition namely BPEL which is based on XML. Indeed, BPEL is used to define the abstract and executable business processes as a set of Web services coordinated recursively. Yet, it is necessary to make sure that these services can interact properly.

In this context, we propose D&A4WSC as a framework allowing the user to model the composition of Web services by the composition of open workflow nets (oWF-nets) [2]. It allows in special, the compatibility checking [3] and the generation of a BPEL process for the composite service. D&A4WSC is composed of four modules (as drawn in figure 1): a module for WSC modelling, an SMV translator, an analysis module and a BPEL generator of WSC.

WSC modelling: Web services modelling is performed using oWF-nets as a subclass of Petri nets [4]. Each service is modeled by an oWF-net in which transitions represent the different tasks performed by the service, places define the conditions and output (resp. input) interface places send (resp. receive) messages to (resp. from) partners.

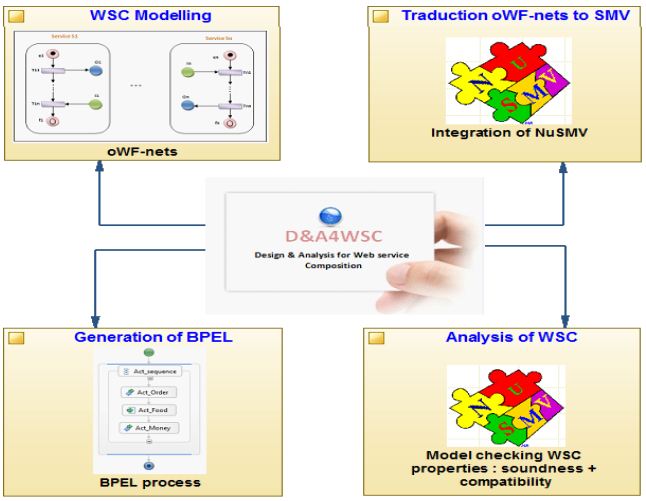


Fig. 1. The proposed approach

SMV generation of WSC: In order to be checked by NuSMV, the composition of oWF-nets is converted to SMV code. In this code, we mapped the net evolution and the firing history in integer arrays in which this evolution will be saved.
Analysis of WSC: D&A4WSC checks first for a syntactical compatibility, which consists of a test of conformance in the number and types of interfaces. Then, a semantic compatibility is enhanced, testing thus if the composition suffers from any problem like deadlocks. We characterized 3 classes of compatibility and showed how to model check them using NuSMV model checker [1].
Generation of BPEL process: BPEL allows the integration and orchestration of a large number of applications published in the form of services. Thus, we integrated in our framework a generator of a compatible composition into a BPEL process. This generator consists on translating oWF-nets to owfn files and then invoking the oWFN2BPEL to convert these files to BPEL.

References

1. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: Nusmv: A new symbolic model verifier. In: Proceedings of the 11th International Conference on Computer Aided Verification. pp. 495–499. Springer-Verlag (1999)
2. Lohmann, N., Kleine, J.: Fully-automatic translation of open workflow net models into simple abstract bpel processes. In: Modellierung 2008. Volume 127 of LNI., GI. pp. 57–72 (2008)
3. Martens, A.: On compatibility of web services. In: Petri Net Newsletter. pp. 12–20 (2003)
4. Sbaï, Z., Barkaoui, K.: Vérification formelle des processus workflow - extension aux workflows inter-organisationnels. Revue Ingénierie des Systèmes d'Information: Ingénierie des systèmes collaboratifs. 18(5), 33–57 (2013)

Constructing Petri Net Transducers with PNT$_{\varepsilon}^{\text{ooL}}$

Markus Huber and Robert Lorenz

Department of Computer Science
University of Augsburg, Germany
firstname.lastname@informatik.uni-augsburg.de

Abstract This poster presents a tool for the modular construction of Petri net transducers by means of several composition operations as for example union, concatenation, closure, parallel and synchronous product and language composition. There are exports to PNML and several graphical output formats.
For the implementation, we use the SNAKES framework, which is a Python library supporting the rapid prototyping of new Petri net formalisms and provides many basic Petri net components and functionality.
In the context of our research activities, PNT$_{\varepsilon}^{\text{ooL}}$ serves as a scientific prototype for the development of an open library openPNT of efficient algorithms for the construction, composition, simulation and optimisation of PNTs which can be used in real world examples.

In [1] we introduced Petri net transducers (PNTs) and showed in [2] how they can be successfully applied in the field of semantic dialogue modelling for translating utterances into meanings. In [3] we presented a basic theoretical framework of PNTs.

In short PNTs are a formalism for the weighted translation of labelled partial orders (LPOs), which are words consisting not of a total order between their symbols but of a partial order. In this sense they are a generalisation of weighted finite state transducers (FSTs) translating words.

(a) A transducer N_1 and a generator N_2. (b) Parallel product of N_1 and N_2 after t_1 has fired.

Figure 1. Some simple PNTs.

In Figure 1 one can see on the left side two simple PNTs called N_1 and N_2. Like for weighted FSTs input symbols, output symbols and weights are annotated to the transitions. So the transition of N_1 reads the symbol a and writes the symbol x (with weight 0.5) – thus N_1 translates the word a into the word x (with weight 0.5). The symbol ε is used to denote empty input or output. So the PNT N_2 is a generator which produces the word y. The weights are elements of an algebraic structure called bisemiring [3] which extends semirings by an additional operation. Thus, a bisemiring a set equipped

with three operations, namely addition, sequential multiplication and parallel multiplication, satisfying certain consistency properties (for example the existence of neutral elements). Addition is used to combine the weights of alternative LPO-runs of a PNT, sequential multiplication of weights is used for sequentially composed LPOs and the weights of parallel composed LPOs are parallel multiplied. In the examples we use the extended Viterbi semiring $([0,1], max, \cdot, min)$.

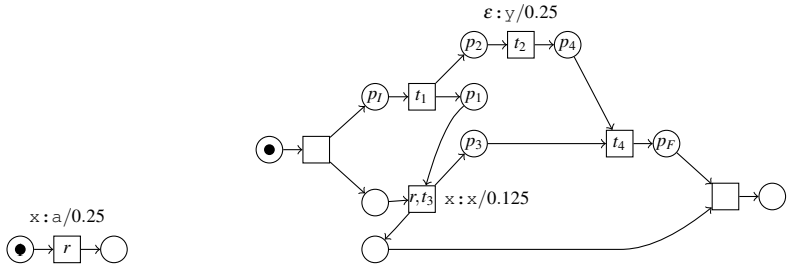
As for FSTs there exist composition operations for combining simple transducers to more complex ones. For example, Figure 1 shows on the right side the parallel product of the PNTs – an operation which does not exist for FSTs. In figures, we omit annotations of the form $\varepsilon{:}\varepsilon/1$ where $\overline{1}$ is the neutral weight w.r.t. sequential and parallel multiplication. The PNT realises the translation from the word `a` into the LPO `w=x`$\|$`y` where $\|$ denotes the parallel composition of LPOs. There are more operators defined for PNTs namely concatenation, union, closure, synchronous product and language composition of PNTs [3,2]. In Figure 2 on the right side the language composition of the PNT N_3 on the left side with the PNT shown on the right side of Figure 1 is illustrated. Here, a transition from the first PNT is merged with a transition from the second PNT, if the output of the first transition equals the input of the second one. The weight of the merged transition labelled r, t_3 is computed from the weights of the transitions labelled r and t_3 using sequential multiplication.

Note that PNTs are defined to always have a single source place which holds exactly one token at the initial state. Furthermore a PNT has a single sink place and per definition only such LPO-runs are considered, which lead to a state where exactly the sink place is marked by one token (the final state). Each LPO-run translates an LPO over input symbols into an LPO over output symbols via a projection onto input symbols resp. output symbols [3].

For such a formalism to be useful one needs a tool where PNTs can be implemented, analysed, combined, simulated, drawn and the like. Since the PNT-formalism is new we decided to start PNT$_\varepsilon^{\text{ooL}}$. We use the SNAKES framework [4] which is a Python library supporting the rapid prototyping of new Petri net formalisms and provides many basic Petri net components and functionality. Therefore we implemented PNT$_\varepsilon^{\text{ooL}}$ as a Python library extending SNAKES such that we essentially can use all the functionality already provided by SNAKES. All graphics in this paper were generated by PNT$_\varepsilon^{\text{ooL}}$.

The support of graphical output serves as a possibility to check the implementation and as a handy utility in the process of writing scientific papers. PNT$_\varepsilon^{\text{ooL}}$'s functionality supports fast construction of concrete example PNTs for case studies. PNML export can be used to analyse constructed example PNTs with other Petri net tools. In the context of our research activities, PNT$_\varepsilon^{\text{ooL}}$ serves as a scientific prototype for the development of an open library openPNT of efficient algorithms for the construction, composition, simulation and optimisation of PNTs which can be used in real world examples.

PNT$_\varepsilon^{\text{ooL}}$ can be downloaded as a ZIP-archive from our website `www.informatik.uni-augsburg.de/EduCoSci/PNTooL`. Assumed you have a working installation of Python, SNAKES, Graphviz, and dot2tex you only need to copy the `py`-files into the `plugins` sub-directory of your SNAKES installation.



(a) A simple PNT N_3. (b) Language composition $N_3 \circ (N_1 \parallel N_2)$.

Figure 2. Some more PNTs.

References

1. R. Lorenz and M. Huber. Petri net transducers in semantic dialogue modelling. In M. Wolff, editor, *Proceedings of "Elektronische Sprachsignalverarbeitung (ESSV)"*, volume 64 of *Studientexte zur Sprachkommunikation*, pages 286 – 297, 2012.
2. R. Lorenz and M. Huber. Realizing the Translation of Utterances into Meanings by Petri Net Transducers. In P. Wagner, editor, *Proceedings of "Elektronische Sprachsignalverarbeitung (ESSV)"*, volume 65 of *Studientexte zur Sprachkommunikation*, pages 103 – 110, 2013.
3. R. Lorenz, M. Huber, and G. Wirsching. On weighted Petri Net Transducers. In *Proccedings of "35th International Conference on Application and Theory of Petri Nets and Concurrency"*, Lecture Notes in Computer Science. Springer, 2014.
4. F. Pommereau. The SNAKES toolkit. `https://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/`, 11 2013.

SLAP$_N$: A Tool for Slicing Algebraic Petri Nets

Yasir Imtiaz Khan and Nicolas Guelfi

University of Luxembourg, Laboratory of Advanced Software Systems
6, rue R. Coudenhove-Kalergi, Luxembourg
{yasir.khan,nicolas.guelfi}@uni.lu

Abstract. Algebraic Petri nets is a well suited formalism to represent the behavior of concurrent and distributed systems by handling complex data. For the analysis of systems modelled in Algebraic Petri nets, model checking and testing are used commonly. Petri nets slicing is getting an attention recently to improve the analysis of systems modelled in Petri nets or Algebraic Petri nets. This work is oriented to define Algebraic Petri nets slicing and implement it in a verification tool.

1 Introduction

Among several dedicated analysis techniques for Petri nets (PNs) and Algebraic Petri nets (APNs) (i.e., an evolution to PNs), model checking and testing are used more commonly. A typical drawback of model checking is its limits with respect to the state space explosion problem. Similarly testing suffers with the problems such as large input amount of test data, test case selection etc.

Petri nets slicing is a technique that aims to improve the verification of systems modelled in Petri nets. *PNs slicing* is used to syntactically reduce Petri net model based on the given *criteria*. A criteria is a property for which Petri net model is analysed. The sliced part constitutes only that part of the PN model that may affect the criteria. Roughly, we can divide *PN Slicing* into two major classes, which are

Static Slicing: If the initial markings of places are not considered for generating sliced net.

Dynamic Slicing: If the initial markings of places are considered for generating sliced net.

One characteristic of APNs that makes them complex to slice is the use of multisets of algebraic terms over the arcs. In principle, algebraic terms may contain variables. Even though, we want to reach a syntactically reduced net (to be semantically valid), its reduction by slicing, needs to determine the possible ground substitutions of these algebraic terms. We use partial unfolding proposed in [1] to determine ground substitutions of the algebraic terms over the arcs of an APN. In the first column of Table1, our proposed APN slicing algorithms are shown [2, 3]. The second column represents properties that are preserved by algorithm whereas in the last column slicing type is mentioned.

Table 1. Different APNs Slicing Algorithms

Algorithm	*Preserved Prop*	*Type Slicing*
Abstract Slicing	CTL*$_{-X}$	*Static*
APN Slicing	LTL$_{-X}$	*Static*
Liveness Slicing	*Livenss*	*Static*
Concerned Slicing	*Particular*	*Dynamic*

1.1 SLAPN$_N$ Overview

First of all, an APN is partially unfolded and from the temporal description of properties places are extracted (shown in Fig.1). Different slicing algorithms such as *abstract slicing, concerned slicing, APN slicing, safety slicing, liveness slicing* can be used to generate the slice (can be observed in the meta model of SLAP$_N$ (shown in Fig.2)). Following steps an APN slice can be generate by the tool.

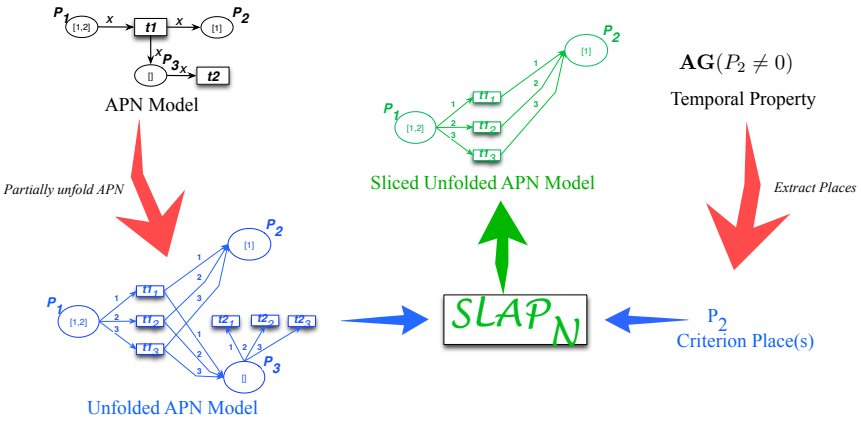


Fig. 1. SLAP$_N$ Overview

Step 0: Create an APN model and interesting property (i.e., writing a property in the form of temporal formula).

Step 1: Choose a slicing algorithm.

Step 3: Sliced APN model is generated.

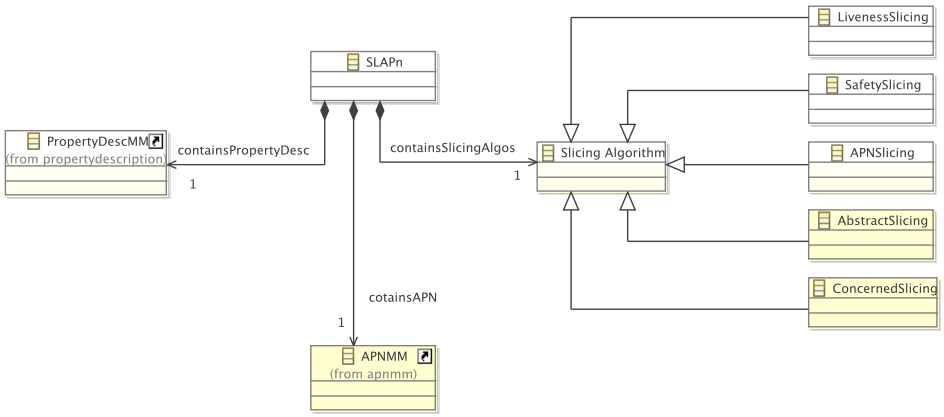


Fig. 2. SLAP$_N$ Meta Model

2 Conclusion and Future Work

- APN slicing can be used as a pre-processing step towards the verification of systems modelled in APNs. The sliced APN model can then be used to generate state space.
- Our work is the first effort to define and implement the proposed slicing algorithms.
- As a future work, we consider to integrate SLAP$_N$ with the existing model checkers such as AlPiNA [3].
- We intend to develop SLAP$_N$ as a generic tool over the PN classes such as timed PN, colored PN.

References

1. D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. Alpina: A symbolic model checker. In J. Lilius and W. Penczek, editors, *Applications and Theory of Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*, pages 287–296. Springer Berlin Heidelberg, 2010.
2. Y. I. Khan. Slicing high-level petri nets. Technical Report TR-LASSY-14-03, University of Luxembourg, 2014.
3. Y. I. Khan and M. Risoldi. Optimizing algebraic petri net model checking by slicing. *International Workshop on Modeling and Business Environments (ModBE'13, associated with Petri Nets'13)*, 2013.

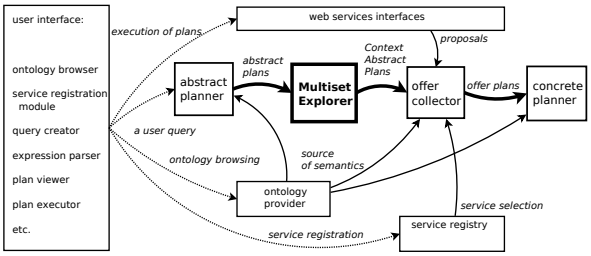
This work has been supported by the National Research Fund, Luxembourg, Project RESIsTANT, ref.PHD-MARP-10.

Generating CA-Plans from Multisets of Services

Łukasz Mikulski[1], Artur Niewiadomski[2], Marcin Piątkowski[1], Sebastian Smyczyński[3]

[1] Nicolaus Copernicus University,
{lukasz.mikulski, marcin.piatkowski}@mat.umk.pl
[2] Siedlce University, artur.niewiadomski@uph.edu.pl
[3] Simplito Computer Science Lab, s.smyczynski@simplito.com

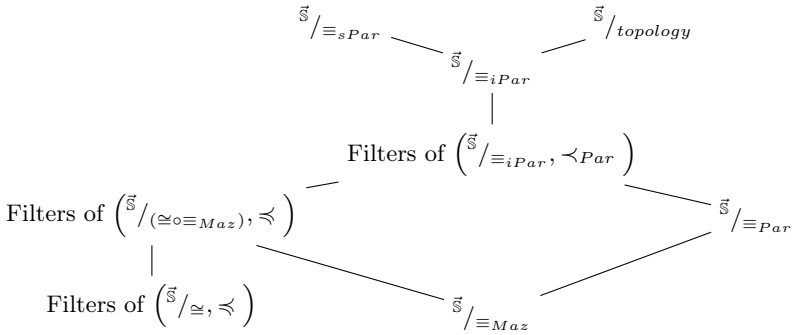
The main idea of solving WSCP utilised by PlanICS(see [3]) is to divide the composition process into several stages. The first phase, called *abstract planning*, deals with an *ontology* which contains a hierarchy of *classes* describing sets of real-world services and processed object types. Our abstract planners find multisets of service types that potentially satisfy a *user query*. Still, each equivalence class defined by a multiset can be viewed as the union of finer equivalence classes defined by partial orders, in which the plans differ only in the ordering of context independent services. Finding all of such classes is the task of Multiset Explorer - a module of PlanICS presented here.



The realizations of web service composition is a transformation sequence of services together with sets of affected objects (the arguments of those services). In contrast to concrete plan, we abstract from objects attributes. We also abstract from concrete object names (defining an equivalence relation $\cong$). We treat two sequences as indistinguishable if they differ only in types of arguments which are in inheritance relation (we build a partial order $\preccurlyeq$ based on inheritance relation and utilize the filters over $\preccurlyeq$) or are equivalent in Mazurkiewicz sense (see [2]), which we denote by $\equiv_{Maz}$. However, we distinguish between two sequences that match produced objects with the expected ones (specified by user query) differently.

The diagram presented below shows relationships between classes of transformation sequences obtained by dividing the set of all potential ones that starts with the set of initial objects specified in the user query. We denote this set by $\vec{\mathbb{S}}$. At the bottom of this diagram individual transformation sequences ($\vec{\mathbb{S}}/_{\mathbb{I}}$), can be seen. Looking at the top this diagram, we define three equivalence relations based on Parikh equivalence of services utilized in the transformation sequence. Namely, they are $\equiv_{sPar}$ which looks only on names of services (as in abstract plan), $\equiv_{Par}$ which takes into account names of the attributes and lying in be-

tween $\equiv_{iPar}$ which abstracts from the names and types of objects in favor of the inheritance relation. In our solution we cut classes of $\equiv_{sPar}$ into classes of $\equiv_{iPar}$ using the notion of relational structures (see [5]) and based on them equivalence relation $\equiv_{topology}$.



The main goal of the presented procedure is to browse all transformation sequences satisfying a given user query with the same Parikh vector of service specifications without duplicating indistinguishable ones. As an input we take the ontology, the user query in the form of two sets of objects, and an arbitrary multiset of service names that identifies single equivalence class $\equiv_{sPar}$. We start from fixing the names of objects originating form the user query initial world or produced by the considered services. After that, we distribute them between inputs of the services to obtain all possible topologies and compute maximal (in the sense of $\preccurlyeq$) possible types of utilized objects. In the next step, we match the obtained possibilities with the user query expected world, considering all valid matchings. The last step is browsing all traces (in Mazurkiewicz sense) based on the multisets of context services from $\vec{\mathbb{S}}/_{\equiv_{iPar}}$. This phase of the algorithm is based on the approach presented in [4] adapted to the specification in the form of a relational structure.

The preliminary experimental results are very promising. We used Z3 SMT-solver together with Abstract Planner (AP). We browse all solutions equivalent in the sense of $\equiv_{sPar}$ with the one reported by AP. In the case of the shortest plans we are able to validate their uniqueness significantly faster than the procedure of their generation. For longer plans, we are as fast as Z3+AP reporting from 1.5 to 5 times more plans.

Acknowledgements This research was supported by the National Science Center under the grants No.2011/01/B/ST6/01477 and No.2013/09/D/ST6/03928.

References

1. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. LNCS 4963:337-340, Springer, 2008.
2. V. Diekert and G. Rozenberg, editors. *The Book of Traces.* World Scientific, 1995.
3. D. Doliwa et al. PlanICS - a web service composition toolset. *Fund. Inf.*, 112(1):47-71, 2011.
4. Ł. Mikulski et al. Algorithmics of posets generated by words over partially commutative alphabets (extended). *Scientific Annals of Comp. Sci.*, 23(2):229-249, 2013.
5. R. Janicki et al. Causal structures for general concurrent behaviours. In *CS&P'13*, pp. 193-205.

LoLA as Abstract Planning Engine of PlanICS*

Artur Niewiadomski[1] and Karsten Wolf[2]

[1] ICS, Siedlce University, 3-Maja 54, 08-110 Siedlce, Poland
artur.niewiadomski@uph.edu.pl
[2] ICS, University of Rostock, 18051 Rostock, Germany
karsten.wolf@uni-rostock.de

Abstract. An abstract planning (AP) is the first phase of the web service composition in the PlanICS framework. We propose an automatic translation of AP to reachability problem in high-level Petri nets, and exploiting the LoLA tool to solve it. We present our approach together with a prototype implementation, and preliminary experimental results.

Keywords: Web Service Composition, Abstract Planning, PlanICS, LoLA

1 Introduction

PlanICS [1] is a system that solves the Web service composition problem by dividing it into several stages. The first phase, called the *abstract planning*, deals with an *ontology* which contains a hierarchy of *classes* describing sets of real-world services and processed object types.

This paper reports the use of LoLA [3] tool as an abstract planning engine. To this aim, we developed a translator which takes an ontology and a user query as the input, and builds a high-level Petri net augmented with a state-predicate formula P. The translation is performed in such a way that some marking satisfying P is reachable only if the planning problem has a solution. Moreover, the comparison of very first results with those obtained with SMT-based abstract planner [2] shows that our new approach is very promising.

2 Translation

At this early stage of our work we assume that service descriptions do not contain alternatives, and we do not take the type inheritance into account. According to the restrictions above, the main ideas of our translation are as follows. The object types of PlanICS become LoLA sorts, i.e., domains for tokens on places. These sorts are records consisting of components corresponding to object attributes. Moreover, for each object type we put a single place in a high-level output net,

* Partially supported by the European Union from resources of the European Social Fund. Project PO KL "Information technologies: Research and their interdisciplinary applications", Agreement UDA-POKL.04.01.01-00-051/10-00.

Table 1. Experimental results

		avg				LoLA		PlanICS	
types	**attrs**	**objs**	**conds**	**services**	**length**	**time[s]**	**mem[MB]**	**time[s]**	**mem[MB]**
78	58	3,14	1,58		6	**0,05**	**6,71**	3,88	12,17
92	62	3,98	2,06	64	12	**0,07**	**7,26**	155	34,76
106	74	3,88	2,17		15	**0,08**	**7,59**	593	63,19
137	136	3,53	1,77		6	**0,33**	30,32	16,57	**16,67**
143	113	3,28	1,55	128	12	**0,06**	**9,92**	242	44,16
142	112	3,34	1,71		15	**0,06**	**9,85**	1007	69,8
151	209	3,74	1,89		6	**4,95**	434	37,26	**21,11**
151	173	3,6	1,76	256	12	**12,1**	934	659	**69,01**
151	166	3,51	1,75		15	**22,5**	1639	1038	**85,85**
101	141	5,16	5,06			382	6643	**13,18**	**17,12**
121	145	6,92	10,4		5	-	>7000	**15,43**	**19,28**
127	132	7	7,2	50		67,3	1045	**12,63**	**17,97**
127	181	6,1	4,62		10	-	>7000	**92,66**	**90,92**
119	193	5,96	4,9			-	>7000	**155**	**230,7**

and thus *worlds* (sets of objects in some states) are represented by markings. The PlanICS services, which transform worlds by producing new objects and changing states of existing ones, become transitions consuming and producing tokens from/to world places w.r.t. service descriptions. Finally, the part of user query specifying the initial worlds is translated to a set of (initially marked) places and transitions which put appropriate tokens to world places. Similarly, a set of special places and transitions, together with a state-predicate formula, correspond to the query fragment concerning the goal of the plan. Moreover, some reductions of an input ontology have been implemented. Their aim is to remove all object types and attributes which are neither used by services, nor query, what leads to a significant improvement of LoLA performance.

3 Experiments and Conclusion

In order to evaluate the efficiency of our approach, we generated a number of parametrized random benchmarks using PlanICS Ontology Generator. Then, we tried to solve them exploiting a prototype PlanICS2LoLA implementation and LoLA tool, as well as an SMT-based planner. The obtained results are summarized in Table 1. We marked in bold the better ones. The overall conclusion is that due to ontology reductions applied during the translation, LoLA in many cases outperforms the SMT-based abstract planner.

References

1. D. Doliwa et al. PlanICS - a Web Service Compositon Toolset. *Fundam. Inform.*, 112(1):47–71, 2011.
2. A. Niewiadomski and W. Penczek. Towards SMT-based Abstract Planning in PlanICS Ontology. In *Proc. of KEOD 2013*, pages 123–131, 2013.
3. K. Schmidt. LoLA: A Low Level Analyser. In *Application and Theory of Petri Nets 2000*, volume 1825 of *LNCS*, pages 465–474. Springer Berlin Heidelberg, 2000.

PlanICS 2.0 - A Tool for Composing Services*

Artur Niewiadomski[1] and Wojciech Penczek[1,2]

[1] ICS, UPH, Siedlce, Poland, artur.niewiadomski@uph.edu.pl
[2] ICS PAS, Warsaw, Poland, wpenczek@gmail.com

Abstract. This poster reports on the current state of the PlanICS toolset, which aims at solving the Web service composition problem by dividing it into several stages. These include an abstract planning, an offer collecting, and a concrete planning.

Keywords: Web Service Composition, multi-phase Planning, SMT, GA

1 Introduction

A Web Service Composition is a hot topic of many theoretical and practical approaches. It is so deeply investigated since typically a simple Web service does not need to satisfy a user objective. Moreover, due to a support of automatic tools the user is exempted from a manual preparation of execution plans, matching services to each other, and choosing optimal providers for all components. In this poster, we report on the current state of the Web service composition system PlanICS [1]. We describe the general idea behind the system and its modules as well as the work in progress together with some future work directions.

2 PlanICS

PlanICS makes use of a uniform semantic description of services and service types as a part of the *ontology*, which contains also the objects processed by the services. The user query is expressed in a fully declarative language defined over terms from the ontology. The user describes two object sets, called the *initial* and the *expected world*. The task of PlanICS consists in finding a way of transforming the initial world into a superset of the expected one using service types available in the ontology and matching them later with real-world services.

The general system architecture is shown in Figure 1. PlanICS divides the composition process into several stages. The first phase, called the *abstract planning*, deals only with the service types of the ontology. So far, we have implemented two abstract planners: the SMT-based one [3] and the other based on Genetic Algorithms (GA) [8]. Currently, we investigate hybrid algorithms combining SMT with GA, and we work on a translation of the abstract planning to a task for tools dealing with Petri nets, like LoLA [7]. Moreover, we work

* This work has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.

on extending abstract planning to its temporal [4] and parametric version. The abstract planners find multisets of service types that potentially satisfy a *user query*. Still, such a multiset can be viewed as the union of finer equivalence classes defined by partial orders that are identified by the Multiset Explorer module [2].

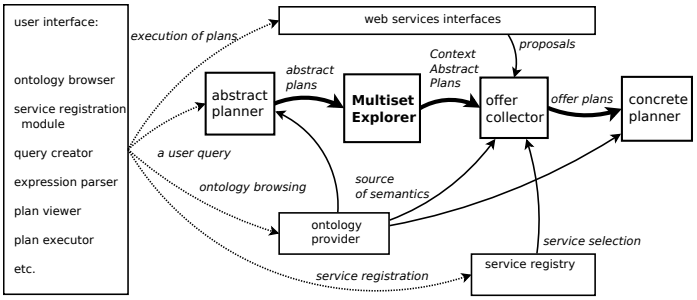


Fig. 1. PlanICS architecture overview.

The second planning stage is performed by the Offer Collector (OC) module, which, in cooperation with service registry, communicates with Web services collecting data to replace the abstract attribute values computed in the first planning phase. Moreover, OC is also to build a set of constraints over offers corresponding to the dependencies from the abstract plan, and resulting from the user query. Then, concrete planners (CPs) get into action. Their task is to prepare a concrete plan by choosing one offer from each set in such a way that all the constraints are satisfied, and the quality function (a part of the user query) is maximized. We provide implementations of CPs based on SMT and GA [5], as well as the hybrid one [6] combining the power of both the methods.

References

1. D. Doliwa et al. PlanICS - a Web Service Compositon Toolset. *Fundam. Inform.*, 112(1):47–71, 2011.
2. L. Mikulski et al. Generating CA-Plans from Multisets of Services. In *PNSE, this volume*, 2014.
3. A. Niewiadomski and W. Penczek. Towards SMT-based Abstract Planning in PlanICS Ontology. In *KEOD*, pages 123–131, 2013.
4. A. Niewiadomski and W. Penczek. SMT-based Abstract Temporal Planning. In *PNSE, this volume*, 2014.
5. A. Niewiadomski, W. Penczek, and J. Skaruz. SMT vs Genetic Algorithms: Concrete Planning in PlanICS Framework. In *CS&P*, pages 309–321, 2013.
6. A. Niewiadomski, W. Penczek, and J. Skaruz. Genetic Algorithm to the Power of SMT: a Hybrid Approach to Web Service Composition Problem. In *Service Computation*, pages 44–48, 2014.
7. A. Niewiadomski and K. Wolf. LoLA as Abstract Planning Engine of PlanICS. In *PNSE, this volume*, 2014.
8. J. Skaruz, A. Niewiadomski, and W. Penczek. Evolutionary Algorithms for Abstract Planning. In *PPAM (1)*, volume 8384 of *LNCS*, pages 392–401. Springer, 2013.

Petri Net Simulation as a Service

Petr Polasek, Vladimir Janousek, and Milan Ceska

Faculty of Information Technology, BUT,
IT4Innovations Centre of Excellence,
Bozetechova 1/2, 612 66 Brno, Czech Republic
{polasek,janousek}@fit.vutbr.cz
http://www.fit.vutbr.cz

Abstract. *This paper presents an approach to integrating a Petri net simulator into a service-oriented simulation architecture in order to provide on demand simulation as a service. As a concrete example the simulation tool Renew is wrapped as a service and used to simulate a sample traffic control system represented as a Petri net model that is able to adjust its parameters via reflective simulation in a distributed simulation environment. The simulation architecture where modeling and simulation is treated as a service (MSaaS) is presented. The main components, their roles and attributes are briefly discussed.*

Keywords: simulation as a service, Petri nets, integration of tools, simulation architecture, web services, system design, simulation model, model specification, reflective simulation

1 Introduction, motivation and context

Modeling and Simulation (M&S) plays an unsubstitutable role in successful design of systems. A need for timely, cost-effective and resource-effective development requires to have a model of a system that can be analyzed, simulated, verified and even validated. As one may expect, modeling and simulation of complex and heterogeneous systems requires the application of more than one approach. Although developers have dozens of tools available that support various formalisms and allow them to use different M&S practices and methodologies, their integration into a simulation environment is rather a challenging task. Proper interaction and communication is a non-trivial task. Enabling access to functionality of an integrated application to a group of people is a challenge. The requirement for scalability and on demand computational power in simulation environment may represent another complication.

As a small and motivated team with limited resources, we have relied on properly chosen software for modeling and simulation tasks and inclined to integrate existing software and used a set of tools as a whole to achieve our goals. This was always rather a challenging task as the integration of different modeling and simulation tools was impossible without additional work. Our efforts devoted to standardize communication processes, simplifying the integration, deployment

and additional requirements like remote simulation and model manipulation lead to the creation of a *service oriented architecture* where modeling and simulation is treated as a service (MSaaS). The goal was to have extensible architecture that allows integration of various modeling and simulation tools with different levels of functionality that can be used as a multiparadigm platform for modeling and simulation.

In this paper, we present the architecture and show how the Petri net simulation tool *Renew* [2], [3] can be integrated as a modeling and simulation service and used to simulate a sample traffic control system represented by a Petri net model that is able to adjust its parameters via reflective simulation in a distributed simulation environment.

2 Service Oriented Architecture for Modeling and Simulation

Service Oriented Architecture (SOA) as a software architecture with loosely coupled services is not tied to specific technology and independent services can be accessed without knowledge of their underlying implementation. It can be applied in many different scenarios including web-based services running on different network nodes (servers) or in more challenging cloud environment. The comparison with existing architectures is in [4]. We employ the service-oriented approach mainly to:

1. Enable integration of existing software to supplement our tools that support dynamic and interactive development of systems with unclear specification.
2. Allow reusability of existing tools/applications/services so that they can be widely used.
3. Enable remote simulation and model management.
4. Simplify the installation, deployment and administration of services
5. Allow better scalability and on demand computing power in a distributed simulation environment

2.1 Architecture Details

The simulation architecture is based on web services. Web services stand for discrete web-based applications that provide a software function over a network. Web service has an interface described in *Web Service Description Language (WSDL)* and interacts with other systems in a manner prescribed by its description using a *Simple Object Access Protocol (SOAP)* that defines a messaging framework.

The advantages of web services allowed us to create a simulation framework with well-defined API that can be used by developers. On one side, there is a set of possibly interconnected services and on the other side, there is a client using this interface to perform various tasks, all depending on abilities of the service.

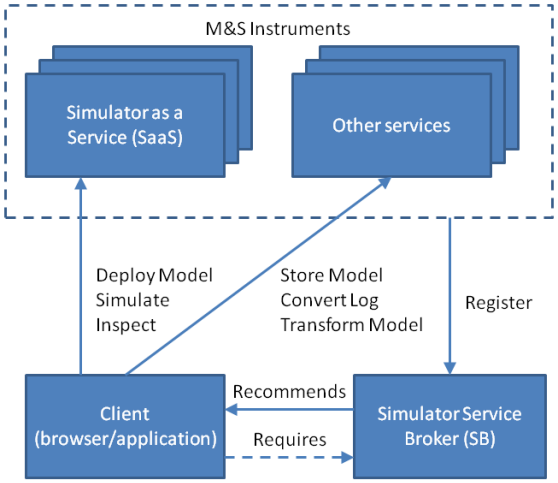


Fig. 1. Service oriented architecture

The architecture is shown in Fig. 1 and its main components are: *simulator as a service (SaaS)*, *service broker (SB)*, *other services* and *client*.

Simulation service allows clients to deploy and simulate models and can be used to inspect running simulations. Every service has a defined interface and allows the client to perform a set of operations. A service doesn't need to support every possible operation and may publish just a few of them - in practice there are a variety of services with different features and clients can choose the proper one that fits their needs. Operation categories are shown in Table 1.

Services are registered in and located by *Service broker* that acts as *UDDI registry (Universal Description, Discovery and Integration)*. It is designed to be interrogated by SOAP messages and to provide access to WSDL documents describing the protocol bindings and message formats required to interact with services listed in its directory. Service broker is used by clients also for a service recommendation and may have implemented a set of rules to decide what service to retrieve. A simple recommendation mechanism may be based on model format or preferred formalisms. Non-simulation services are also registered via Service broker and a client may get its description documents and start using them.

Client is a service consumer. It can use Service broker's registry to locate requested or recommended service and then it communicates directly with simulation or other services. The communication between the client and a service is done via SOAP and it may be a simple request to start a simulation of a specific model: if the model is accepted and the simulation is started, the simulation service responds with the simulation identifier for further reference. Additional configuration parameters may be provided by the client in the request and the model payload can be sent either directly in the XML document, via *SOAP*

attachments or using SOAP *Message Transmission Optimization Mechanism (MTOM)* [9].

Table 1. Service operations

Operation category	Description/Purpose
Administration and configuration	Simulation control Simulator/Service configuration
Inspection and monitoring	Model inspection Simulation inspection Gathering log information Simulator or service monitoring Event Notifications
Model/Simulation Manipulation	Model design Editing actions over models Actions over simulation objects
Storage access	Model repository Simulation repository

Other services provide non-simulation but not less important functions. One of them is the *log presentation service* that can be used to convert raw outputs from simulators to more comprehensive and human readable graphical reports. Our implementation of a log presentation service exists in a Squeak environment [20] empowered with the SoapOpera and was integrated into the simulation environment as a web service. *Model transformation service* is another useful service for developers that is designed to translate models between different formalisms so that a developer can use one formalism for the design and another for the simulation. We have experimented with simple Petri net models that were translated to models in classic DEVS formalism (Discrete Event Systems Formalism) [1] according to the approach presented in [11]. *Model library* service provides very basic functionality. It is a common place where models together with other related documents can be stored as project resources and shared with other clients. It may also offer more sophisticated access control.

2.2 Integration of Existing Tools

To integrate and deploy existing applications or tools and to transform them into simulation services, we use *Apache Axis2* [12] as a core engine for web services. It is a re-designed and re-written successor to the widely used *Apache Axis* [13] SOAP stack with implementations available in Java and C languages. Axis2 provides us with a capability to add a web service interface to the existing application and can function as a standalone server where a service can be deployed without restart. It supports WSDL 2.0 and SOAP 1.2 which allows us to easily build stubs to access remote services.

Renew as a Service Petri net simulation tool Renew is a Java-based multi-formalism editor and simulator that provides a flexible modeling approach based on *reference nets* [15]. We have extended the Renew tool so that it can be deployed in Axis2 server and may act as a service that clients can use for simulation purposes - moreover, the client can be another Renew service and this will allow us to execute nested simulations remotely in the case study below.

To develop a web service, we use a *bottom-up approach*, where implementation of a service has to be coded first. A service operation logic is in Java classes where methods have simple contracts that allow a caller to start or stop simulations, to configure the service, to get logs, etc. This implementation requires the Renew application to be installed on a target system, where it is executed as an external program. Service classes are used to create a deployable service artifact - it is a single archive file and we deploy it in the Axis2 server where it is immediately accessible without reload. We use scripts that automate the service deployment and the installation of the Renew application. Java libraries that may be needed for a model execution can be either pre-installed together with Renew or can be provided by the service later as well.

To be able to invoke service methods over a network, a client that understands the service API has to be implemented. For this purpose, we can use a service description file, i.e. a WSDL file that Axis2 server generates on demand for every deployed service. Then Apache Axis2 tools can be used to create stub classes that may be called from another Java program to invoke the remote Renew service.

Although Renew already supports remote simulation with RMI (Remote Method Invocation), we use SOAP as a messaging framework for communication between services. SOAP has significant advantages over RMI. It uses XML, it is language independent and allows platform independent services whereas RMI is Java-centric. SOAP as a more robust technology allows functionality to be accessible to a variety of clients and it allows a loosely coupled architecture. Although RMI has smaller latency, it is more suited to smaller applications where objects must stay in sync in all applications.

2.3 Simulation as a Service in Cloud Environment

The deployment of simulation services in cloud environments boosts efficiency, allows scalability, simplifies deployment and administration tasks and allows better control over the simulation environment. Especially in cases when the tool integration, installation and service configuration is rather a complicated task, it is worth having a virtual machine with a service or a set of services preinstalled and available in the cloud environment. By maintaining a preconfigured image you can avoid complicated service setup. An easy deployment of virtual machines from captured image with pre-configured services increases architecture scalability for demanding scenarios in multi-simulation environments. We have used Windows Azure [14] to successfully deploy and manage simulator services in the cloud environment.

3 Case study

To demonstrate a Petri Net simulator as a service, we have created a sample model of a traffic control system that controls lights on a simple crossroad. The model is able to adjust its configuration via reflective simulation. Renew was effectively used with all its advantages to design and build the model and when the tool was deployed as a service, it could operate as a remote simulator and as a client at the same time allowing nested simulations to be executed remotely.

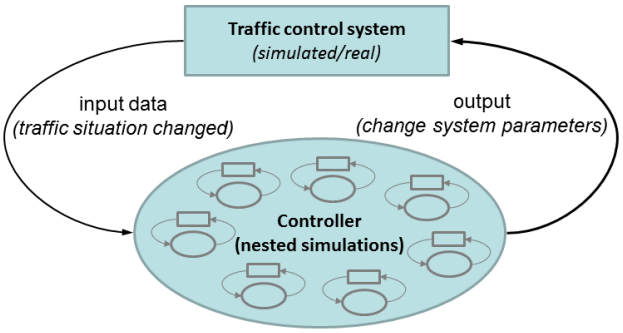


Fig. 2. Concept of reflective simulation with main components

3.1 Reflective simulation

The main concept of reflective simulation is depicted in Fig. 2 and Fig. 3. In Fig. 2 we can see a traffic model with its control system and the way they influence each other. The control system reacts to changes in the traffic (e.g. increased volume of vehicles) which may trigger a change of traffic control system settings (e.g. signal timing plans) and this further influences the traffic (e.g. improves throughput) which may again require a change to control system settings and so on. In this scenario an undesired oscillation may happen - imagine a situation when increasing the green interval in one direction may cause excessive queuing of vehicles waiting in the other direction. In this case, it would be better if the control system could anticipate the effect of the change, infer on its own future behavior and adjust the signal timing appropriately. The control system is able to trigger nested simulations, to simulate its own behavior and to perform decisions based on collected results. We call this a reflective simulation. You can compare it with the approach to nested simulations presented in [17] or [18].

3.2 Model

The model comprises of three main components, each of them being a Petri net: the *traffic model*, the *traffic control system* and the *simulation control model*. The

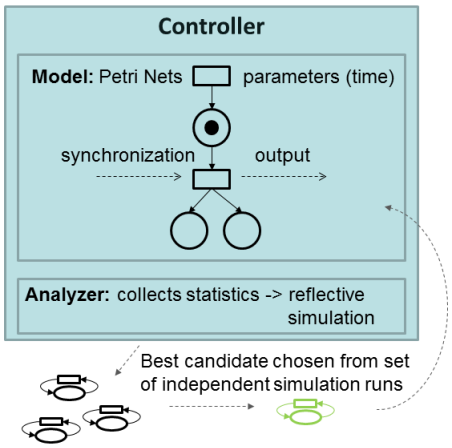


Fig. 3. Simulation control model and its role in reflective simulation

crossroad and its traffic control model were inspired by the example presented in [16] and further modified to our purpose. Other components - the odel of traffic and the simulation control model (in Fig. 4) were created from scratch.

Traffic Model The traffic flow at the crossroad is divided into three phases with blocking and non-blocking directions. In the traffic model net, tokens represent vehicles that travel through the intersection. Each lane and each direction has assigned a time it takes every vehicle to travel through the crossroad. The model contains a statistics part that is dedicated to gathering statistics about the traffic. This comprises the number of vehicles waiting to cross the intersection in each lane and the number of vehicles that successfully travelled through the crossroad in each phase. A *Synchronous channel* concept (uplinks and downlinks) is used to trigger statistics gathering and to deliver results to the simulation control model. The traffic model net has a part that allows its initialization when running in a nested simulation - e.g. it sets a number of waiting vehicles before the simulation starts.

Traffic Control System The Petri net representing the traffic control system is presented in [16]. It reflects three phases of a crossroad, each with signal lights for pedestrians and vehicles. In every phase, only vehicles from the corresponding phase in the traffic model net are allowed to move through the intersection. The actual behavior of the traffic control system is determined by its main parameters that contain time values for signal timing plans. In comparison with [16] the control system net was enhanced to allow changes to its configuration (signal timing plans) and to communicate the active phase to the traffic model net.

Simulation Control The Petri net in Fig. 4 controls the simulation and acts as a controller as shown in Fig. 3. It configures the traffic control system

according to collected data from the traffic model and based on results from nested simulations. More specifically, it:

1. Initializes traffic model and traffic control system nets
2. Starts the simulation
3. Gathers and analyzes data from the traffic model
4. Communicates with other simulation services
5. Triggers nested simulations of its own model with different configurations
6. Collects and interprets logs from nested simulations
7. Changes traffic control system settings according to results from nested simulations

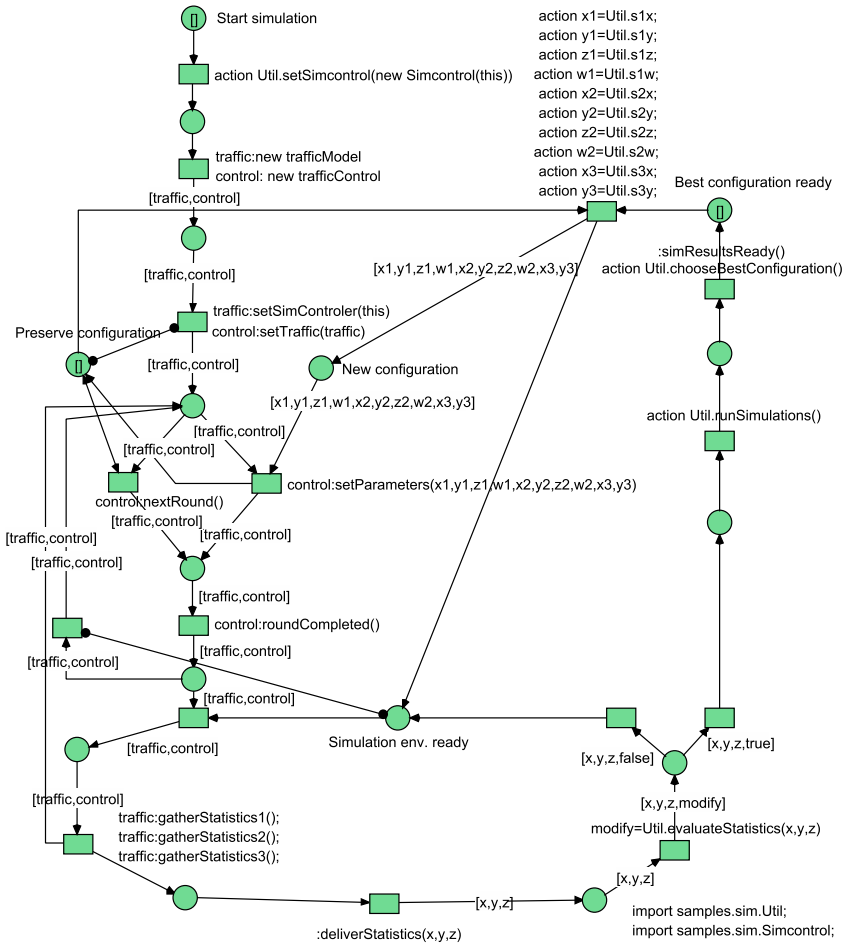


Fig. 4. Simulation control

As reference nets work seamlessly with Java programs, part of the model is implemented in Java language and it is called via action inscriptions or method invocations. This enables access to remote services directly from the Petri net model. For this purpose two Java classes are imported in the bottom right corner in Fig. 4. The *Util* class is used to analyze traffic data and to spawn nested simulations. Under the hood, a small framework with a set of Java libraries is used for communication with remote simulators. The *Simcontrol* is a stub class that wraps the simulation control net. It forwards its method calls to synchronous channels of the wrapped net instance and enables them to be used from Java.

The simulation control net instantiates the traffic net and the traffic control system net and keeps their references, named as *traffic* and *control*.

After the initial configuration the system starts to control the traffic. According to statistics gathered from the traffic model (*traffic: gatherStatistics()*) that are evaluated (*modify=Util.evaluateStatistics()*), the system may decide to reconfigure signal timing plans in traffic control net to increase the throughput. Then a set of configuration candidates is suggested and a nested simulation is executed for each of them to see if it brings some improvement (*action Util.runSimulations()*). The master simulation control model acts as a client. Each simulation is controlled by a single Java thread that communicates with either a local or a remote simulation service(s). Although we can configure the level of nesting and a model in a nested simulation may further trigger other nested simulations of itself, our tests show that there is no significant benefit, especially when compared to increased demand for resources.

After some time, remote simulations are stopped and logs are acquired and interpreted. The simulation that improved the throughput more than others is chosen as the best candidate and its configuration is applied to the traffic control system when possible. A brute force computing is used to achieve continuous improvement and more than one round of nested simulations may be executed. However, the model quickly converge to the optimal configuration, especially when changes in the traffic flow are less extreme.

A model containing all three nets (simulation control net, traffic control system net and traffic model net) is sent to the simulator service along with all necessary libraries, configuration and startup parameters. As we use Renew's non-graphical simulator for nested simulations, the model must be sent as a single merged file in shadow net format. As a result the transmitted model contains only the semantic information and not the visual appearance and it can be used only by a non-graphical simulator. To preserve the graphical information the PNML (Petri Net Markup Language) may be used instead when transferring the model.

Acknowledgement. *The work was supported by the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070 as well as the internal BUT projects FIT-S-12-1 and FIT-S-14-2486.*

References

1. Zeigler, B.P., Kim, T.G., Praehofer, H.: Theory of Modeling and Simulation, Second Edition. Academic Press (2000)
2. Renew - The Reference Net Workshop, http://www.renew.de
3. Kummer, O., Wienberg, F., Duvigneau, M., Kohler, M., Moldt, D., Rolke, H.: Renew - the Reference Net Workshop. In: Tool Demonstrations, pp. 99–102. 24th International Conference on Application and Theory of Petri Nets 2003. Department of Technology Management, Technische Universiteit Eindhoven, Beta Research School for Operations Management and Logistics (2003)
4. Janousek, V., Polasek, P.: Modeling and Simulation Management in Distibuted Environment Using Web Services, In: WOSC 2008 - 14TH International Congress of Cybernetics and Systems. Wroclaw (2008)
5. Johns, K., Taylor, T.: Professional Microsoft Robotics Developer Studio, Paperback, Wiley (2008)
6. Coen-Porsini, A., Gallo, I., Zanzi, A.: Integration of web based simulators in the SINPL platform. In: Proceedings of ESM 2006. Ghent, BE, pp. 259–263 (2006)
7. Dahmann, J.S., Fujimoto, R.M., Weatherly R.M.: The Department of Defense High Level Architecture. In: Proceedings of the 1997 Winter Simulation Konference (1997)
8. Multi-Simulation Interface (MSI) Brochure, http://msi.sourceforge.net
9. W3C - SOAP Message Transmission Optimization Mechanism http://www.w3.org/TR/2005/REC-soap12-mtom-20050125
10. SmallDEVS, www.fit.vutbr.cz/~janousek/smalldevs
11. Jacques, C.J.D., Wainer, G.A.: Using the CD++ DEVS Tookit to Develop Petri Nets. In: Proc. of the 2002 Summer Computer Simulation Conference, San Diego, CA, USA (2002)
12. Apache Axis2 - Apache Axis2/Java - Next Generation Web Services http://axis.apache.org/axis2/java/core
13. Web Services - Axis http://axis.apache.org/axis
14. Windows Azure cloud platform http://www.windowsazure.com
15. Cabac, L., Duvigneau, M., Moldt, D., Rölke H.: Modeling dynamic architectures using nets-within-nets. In: Proceedings, volume 3536 of Lecture Notes in Computer Science, pp. 148–167. 26th International Conference, Applications and Theory of Petri Nets 2005, Miami, USA (2005)
16. Turek, R.: Modelovani vybranych dopravnich problemu s vyuzitim Petriho siti (in Czech) [Modeling of Chosen Traffic Problems with Petri Nets]. In: Posterus.sk, Portal pre odborne publikovanie [Portal for Scientific Publications], vol. 3, nr. 12 (2010)
17. Sklenar, J. - Introduction to OOP in Simula (Nested Simulation) http://staff.um.edu.mt/jskl1/talk.html
18. Kindler, E. - Reflective Simulation - Simulation of Systems That Simulate. In: Proc. of ESM - European Simulation and Modeling, Porto, Portugal (2005)
19. Chandrasekaran, S., Cardoso, J., Silver, G., Miller, A.J., Sheth, A.P.: Web service technologies and their synergy with simulation. In: Proceedings of the 2002 Winter Simulation Conference, pp. 606–615. San Diego, California (2002)
20. Squeak http://www.squeak.org