

Proceedings

»PNSE'15«

International Workshop on
Petri Nets and Software Engineering

Satellite event of the
36th International Conference on
Application and Theory of Petri Nets
and Concurrency

15th International Conference on
Application of Concurrency to
System Design

Brussels, Belgium, June, 2015

including papers of

»ADECS'15«

International Workshop on
Petri Nets for Adaptive Discrete Event Control Systems

Compilation Editor:
Daniel Moldt
University of Hamburg
Department of Informatics
Theoretical Foundations of Informatics
Vogt-Kölln-Str. 30
D-22527 Hamburg
Germany
<http://www.informatik.uni-hamburg.de/TGI/>

These proceedings are published online by the editors as Volume 1372 at

CEUR Workshop Proceedings
ISSN 1613-0073
<http://ceur-ws.org/Vol-1372>

Copyright © 2015 for the individual papers is held by the papers' authors. Copying is permitted only for private and academic purposes. This volume is published and copyrighted by its editors.

PNSE'15 Editors: Daniel Moldt and
Heiko Rölke and
Harald Störrle

Proceedings of the
International Workshop on

Petri
Nets and
Software
Engineering

PNSE'15

University of Hamburg
Department of Informatics

Preface

These are the proceedings of the International Workshop on *Petri Nets and Software Engineering* (PNSE'15), which also includes the papers of the International Workshop on *Petri Nets for Adaptive Discrete Event Control Systems* (ADECS'15) in Brussels, Belgium, June 22–23, 2015.

They are co-located events of *Petri Nets 2015*, the 36th International Conference on Applications and Theory of Petri Nets and Concurrency and *ACSD 2015*, the 16th International Conference on Application of Concurrency to System Design. More information about the workshops can be found at:

For PNSE'15:

<http://www.informatik.uni-hamburg.de/TGI/events/pnse15/>

For ADECS'15:

<http://adecs2015.cnam.fr/>

PNSE'15 preface:

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.

ADECS'15 preface:

The new generation of Discrete Event Control Systems (DECS) is addressing new important criteria as flexibility and agility. This year concentrated on Dynamic Software Architectures and Adaptable Systems.

We included the papers of ADECS'15 in PNSE'15 and received more than 30 high-quality contributions. For each paper at least three reviews were provided. The program committees have accepted seven of them for full presentation. Furthermore the committee accepted eight papers as short presentations. Several more contributions were submitted and accepted as posters.

The two invited talks are presented by

Fabio Gadducci (UNIVERSITY OF PISA, ITALY).

Nicolas Guelfi (UNIVERSITY OF LUXEMBOURG, LUXEMBOURG)

The international program committee of PNSE'15 was supported by the valued work of following sub reviewers:

Thomas Brand,
Cesar Rodriguez,
Marisa Llorens,
Benjamin Meis,
Pedro Alvarez,
Marcin Hewelt and
Dimitri Plotnikov

Their work is highly appreciated.

Furthermore, we would like to thank our colleagues in Hamburg, Germany, for their support in the compilation of the proceedings, in Hagen, Germany, for the support with the CEUR handling and in Brussels, Belgium, for their excellent and responsive organizational 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, Heiko Rölke and Harald Störrle (Chairs for PNSE'15) and
Kamel Barkaoui and Chadlia Jerad (Chairs for ADECS'15)

The PNSE'15 program committee consists of:

Bernhard Bauer (University of Augsburg, Germany)
 Robin Bergenthum (University of Hagen, Germany)
 Didier Buchs (University of Geneva, Switzerland)
 Lawrence Cabac (University of Hamburg, Germany)
 Piotr Chrzastowski-Wachtel (University of Warsaw, Poland)
 Gianfranco Ciardo (Iowa State University, USA)
 José-Manuel Colom (University of Zaragoza, Spain)
 Ernesto Damiani (University of Milan, Italy)
 Patrick Delfmann (University of Münster, Germany)
 Raymond Devillers (Université Libre de Bruxelles, Belgium)
 Susanna Donatelli (University of Turin, Italy)
 Gregor Engels (University of Paderborn, Germany)
 Joaquín Ezpeleta Mateo (University of Zaragoza, Spain)
 Jorge C. A. de Figueiredo (Federal University of Campina Grande, Brazil)
 Ulrich Frank (University of Duisburg-Essen, Germany)
 Holger Giese (University of Potsdam, HPI, Germany)
 Paolo Giogini (University of Trento, Italy)
 Luís Gomes (Universidade Nova de Lisboa, Portugal)
 Nicolas Guelfi (University of Luxembourg)
 Stefan Haar (ENS Cachan, France)
 Serge Haddad (ENS Cachan, France)
 Nabil Hameurlain (Université de Pau et des Pays de l'Adour, France)
 Xudong He (Florida International University, USA)
 Vincent Hilaire (Université de Technologie de Belfort-Montbéliard, France)
 Thomas Hildebrandt (IT University of Copenhagen, Denmark)
 Lom-Messan Hillah (Université Paris Ouest and LIP6 (UPMC), France)
 Kunihiro Hiraishi (Japan Advanced Institute of Science and Technology, Japan)
 Vladimír Janoušek (Brno University of Technology, Czech Republic)
 Peter Kemper (College of William and Mary, USA)
 Astrid Kiehn (IIIT Delhi, India)
 Ekkart Kindler (Technical University of Denmark, Denmark)
 Hanna Klaudel (Université d'Evry-Val d'Essonne, France)
 Radek Kočí (Brno University of Technology, Czech Republic)
 Fabrice Kordon (Université Paris Ouest and LIP6 (UPMC), France)
 Maciej Koutny (University of Newcastle, United Kingdom)
 Lars Kristensen (Bergen University College, Norway)
 Michael Köhler-Bußmeier (University of Applied Science Hamburg, Germany)
 Niels Lohmann (Carpeq GmbH, Germany)
 Robert Lorenz (University of Augsburg, Germany)
 Heinrich Mayr (Alpen-Adria-Universität Klagenfurt, Austria)
 Jan Mendling (Vienna University of Economics and Business, Austria)

Daniel Moldt (University of Hamburg, Germany) (Co-Chair)
Berndt Müller (University of South Wales, United Kingdom)
Andreas Oberweis (Karlsruhe Institute of Technology, Germany)
Andrea Omicini (University of Bologna, Italy)
Chun Ouyang (Queensland University of Technology, Australia)
Wojciech Penczek (UPH Siedlce and IPI PAN Warsaw, Poland)
Laure Petrucci (University Paris 13, France)
Lucia Pomello (Università degli Studi di Milano-Bicocca, Italy)
Heiko Rölke (DIPF, Germany) (Co-Chair)
Bernhard Rumpe (RWTH Aachen, Germany)
Christophe Sibertin-Blanc (Université Toulouse 1, France)
Mark-Oliver Stehr (SRI International, USA)
Harald Störrle (Technical University of Denmark, Denmark) (Co-Chair)
Ingo Timm (University of Trier, Germany)
Adeline Uhrmacher (University of Rostock, Germany)
Eric Verbeek (Eindhoven University of Technology, Netherlands)
Jan Martijn van der Werf (Utrecht University, Netherlands)
Mathias Weske (University of Potsdam, HPI, Germany)
Manuel Wimmer (Vienna University of Technology, Austria)
Karsten Wolf (University of Rostock, Germany)

The ADECS'15 program committee consists of:

- Hassane Alla, France,
- Farhad Arbab, The Netherlands
- Mohamed Bakhouya, Morocco
- Faiza Belala, Algeria
- Beatrice Bérard, France
- Luca Bernardinello, Italy
- Hanifa Boucheneb, Canada
- Roberto Bruni, Italy
- Javier Campos, Spain,
- Piotr Chrzastowski-Wachtel, Poland,
- Vincenzo De Florio, Belgium
- Raymond Devillers, Belgium
- Maria Pia Fanti, Italy
- Georg Frey, Germany,
- Abdoulaye Gamatie, France
- Alessandro Giua, Italy,
- Luis Gomes, Portugal,
- Serge Haddad, France
- Hans-Michael Hanisch, Germany,
- Vladimír Janoušek, Czech Republic,
- Jorge Júlvez, Spain
- Laid Kahloul, Algeria
- Peter Kemper, USA,
- Mohamed Khargui, Tunisia
- Hanna Klaudel, France,
- Michael Köhler-Bußmeier, Germany,
- Radek Koci, Czech Republic,
- Ouajdi Korbaa, Tunisia
- Lars Kristensen, Norway,
- Alberto Lluch Lafuente, Italy
- Petrucci Laure, France,
- Zhiwi Li, China
- Gaiyun Liu, China
- Robert Lorenz, Germany,
- Cristian Mahulea, Spain
- Ramon Piedrafita Moreno, Spain
- Andrey Mokhov, UK
- Daniel Moldt, Germany,
- Javier Oliver, Spain
- Wojciech Penczek, Poland
- Riadh Robbana, Tunisia
- Éric Rutten, France

- José Luis Villarroel Salcedo, Spain
- Heiko Rölke, Germany,
- Kleanthis Thramboulidis, Greece,
- Yamen El Touati, Tunisia
- Murat Uzam, Turkey,
- Valeriy Viyatkina, New Zealand,
- Habib Youssef, Tunisia
- Weimin Wu, China,
- Mengchu Zhou, USA

Contents

Part I Invited Talks

Software Engineering and Modeling Education: Problems and Solutions

Nicolas Guelfi 17

Awareness and Control in Adaptable Transition Systems

Roberto Bruni, Andrea Corradini, Fabio Gadducci, Alberto Lluch Lafuente and Andrea Vandin 19

Part II Long Presentations

Unifying Patterns for Modelling Timed Relationships in Systems and Properties	
<i>Étienne André and Laure Petrucci</i>	25
Negotiations and Petri Nets	
<i>Jörg Desel and Javier Esparza</i>	41
Non-Interference Notions Based on Reveals and Excludes Relations for Petri Nets	
<i>Luca Bernardinello, Görkem Kılınç and Lucia Pomello</i>	59
Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification	
<i>Kent Inge Fagerland Simonsen, Lars M. Kristensen and Ekkart Kindler</i>	79
Providing Petri Net-Based Semantics in Model Driven-Development for the Renew Meta-Modeling Framework	
<i>David Mosteller, Lawrence Cabac and Michael Haustermann</i>	99
Validating DCCP Simultaneous Feature Negotiation Procedure	
<i>Somsak Vanit-Anunchai</i>	115
Dynamic Software Architecture for Distributed Embedded Control Systems	
<i>Tomáš Richta, Vladimír Janoušek and Radek Kočí</i>	133

Part III Short Presentations

Reengineering the Editor of the GreatSPN Framework <i>Elvio Gilberto Amparore</i>	153
Improving Performance of Complex Workflows: Investigating Moving Net Execution to the Cloud <i>Sofiane Bendoukha and Thomas Wagner</i>	171
Modelling the Behaviour of Management Operations in Cloud-based Applications <i>Antonio Brogi, Andrea Canciani, Jacopo Soldani and Pengwei Wang</i> ...	191
Unfolding CSPT-nets <i>Bowen Li and Maciej Koutny</i>	207
Discovery of Functional Architectures From Event Logs <i>Jan Martijn van der Werf and Erwin Kaats</i>	227
Interval-Timed Petri Nets with Auto-concurrent Semantics and their State Equation <i>Elisabeth Pelz, Abderraouf Kabouche and Louchka Popova-Zeugmann</i> ...	245
Lookahead Consistency Models for Dynamic Migration of Workflow Processes <i>Ahana Pradhan and Rushikesh K. Joshi</i>	267
Catalog-based Token Transportation in Acyclic Block-Structured WF-nets <i>Ahana Pradhan and Rushikesh K. Joshi</i>	287

Part IV Poster Abstracts

De-Materializing Local Public Administration Processes

*Giancarlo Ballauco, Paolo Ceravolo, Ernesto Damiani, Fulvio Frati
and Francesco Zavatarelli* 311

Renew – The Reference Net Workshop

Lawrence Cabac, Michael Haustermann and David Mosteller 313

**Queue-less, Uncentralized Resource Discovery: Formal
Specification and Verification**

Camille Coti, Sami Evangelista and Kais Klai 315

Introducing the *Quick Fix* for the Petri Net Modeling Tool**RENEW**

Jan Hicken, Lawrence Cabac and Michael Haustermann 317

**Process-oriented Worksheets for the Support of Teaching
Projects**

Dennis Schmitz and Lawrence Cabac 319

**Integrating Network Technique into Distributed Agent-
Oriented Software Development Projects**

Christian Röder and Lawrence Cabac 321

**Applying Petri Nets to Approximation of the Euclidean
Distance with the Example of SIFT**

Jan Henrik Röwekamp and Michael Haustermann 323

**Coordination Rules Generation from Coloured Petri Net
Models**

Adja Ndeye Sylla, Maxime Louvel and François Pacull 325

Part I

Invited Talks

Software Engineering and Modeling Education: Problems and Solutions

Nicolas Guelfi

Faculté des Sciences, de la Technologie et de la Communication,
Université du Luxembourg, nicolas.guelfi@uni.lu

Abstract:

Mastering the development of software having the required quality level is a complex task. Since 1968, the software engineering discipline has grown in order to offer theories, methods and tools to software engineers to tackle this complex task. The role of software engineering educators is to help the learners to acquire competencies in those theories, methods and tools to better master the production of quality products.

- After near than 50 years of development what is the status of software engineering and modeling education?
- What are the attributes, threats and means for quality software engineering education?

This talk will present the outcomes of an individual experience of 25 years of teaching software engineering and modeling in computer science programs at bachelor and master levels. A concrete educational software engineering and modeling environment will be presented as one of the means to better educate our engineers to our discipline and be prepared for facing their future professional challenges.

Awareness and Control in Adaptable Transition Systems^{*}

Roberto Bruni¹, Andrea Corradini¹, Fabio Gadducci¹,
Alberto Lluch Lafuente², and Andrea Vandin³

¹ Department of Informatics, University of Pisa, IT

² DTU Compute, Technical University of Denmark, DK

³ Electronics and Computer Science, University of Southampton, UK

The CoDa approach. Self-adaptive systems are advocated as a solution to the problem of mastering the complexity of modern software systems and the continuous evolution of the environment where they operate. Self-adaptation is considered a fundamental feature of autonomic systems, one that can specialise to several other self-* properties, like self-configuration and self-optimisation.

Should the analysis favour a *black-box* perspective, a software system is called “self-adaptive” if it can modify its behaviour in response to a change in its context. On the contrary, *white-box* adaptation focuses on how adaptation is realised in terms of architectural and linguistic mechanisms and usually promotes a clear separation of adaptation and application logics. Our own approach [2, 5] characterizes adaptivity on the basis of a precisely identified collection of *control data* (CoDa), deemed to be interpreted as those data whose manipulation triggers an adaptation. This view is agnostic with respect to the form of interaction with the environment, the level of context-awareness, the use of reflection for self-awareness. In fact, our definition applies equally well to most of the existing approaches for designing adaptive systems. Overall, it provides a satisfactory answer to the question “what is adaptation *conceptually*?”.

But “what is adaptation *formally*?” and “which is the right way to reason about adaptation, *formally*?”. We are aware of only a few works (e.g. [8]) that address the foundational aspects of adaptive systems, including their semantics and the use of formal reasoning methods, and often only generic analysis techniques are applied. An example of the possibilities of such technique is our approach [4] to adaptive self-assembly strategies using Maude (and following precisely both [8] and [2]), where we applied standard simulation and statistical model checking.

Adaptable Transition Systems. Building on the intuitions briefly discussed above and on some foundational models of component based systems (like *I/O automata* [7] and *interface automata* [1]), we proposed a simple formal model based on a new class of transition systems [3], and we sketched how this definition can be used to specify properties related to the adaptive behaviour of a system. A central role is again played by control data, as well as by the interaction among components and with the environment (not addressed explicitly in [2]).

^{*} Research partially supported by the MIUR PRIN 2010LHT4KM CINA.

Let us recall that the steps of I/O and interface automata are labeled over three disjoint sets of actions, namely *input*, *output* and *internal* actions. The composition of two automata is defined only if certain disjointness constraints over the sets of actions are satisfied, and it is obtained conceptually as a synchronous composition on shared actions and asynchronous on the others, the differences between the two models not being relevant at this level of abstraction.

Adaptable Transition Systems (ATSs) combine these features on actions within an extended Kripke frame presentation, in order to capture the essence of adaptativity. An ATS is a tuple $\mathcal{A} = \langle S, A, T, \Phi, l, \Phi^c \rangle$ where S are the states, $A = \langle I, O, H \rangle$ is a triple of three disjoint sets of input, output and internal actions, and $T \subseteq S \times A \times S$ is a transition relation, where by A here we denote the union $I \uplus O \uplus H$. Furthermore, Φ is a set of atomic propositions, and $l : S \rightarrow 2^\Phi$ is a labeling function mapping states to sets of propositions. Finally, $\Phi^c \subseteq \Phi$ is a subset of *control propositions*, which play the role of the control data [2].

A transition $s \xrightarrow{a} s' \in T$ is called an *adaptation* if it changes the control data, i.e., if there exists a $\phi \in \Phi^c$ such that $\phi \in l(s) \iff \phi \notin l(s')$. Otherwise, it is called a *basic* transition. An action $a \in A$ is called a *control action* if it labels at least one adaptation, and the set of all control actions is denoted by C .

The relationship between the action set C and the alphabets I , O and H is arbitrary in general, but it could satisfy some pretty obvious constraints for specific classes of systems. For example, an ATS \mathcal{A} is *self-adaptive* if $C \cap I = \emptyset$, i.e., if all adaptations are under the control of the system. If instead $C \subseteq I$ the system is *adaptable*; intuitively, adaptations cannot be executed locally but should be triggered by an external manager. Hybrid situations are possible as well, when a system has both input and local control actions.

The composition operations on I/O automata can be extended seamlessly to ATSs. They have been exploited to model the composition of an adaptable basic component \mathcal{A}_B and an adaptation manager \mathcal{A}_M that realizes the adaptation logics, for example a control loop in the style of the MAPE-K architecture [6]. In this case, natural well-formedness constraints could be expressed as relations among sets of actions. For example, the manager controls *completely* the adaptivity features of the basic component if $C_B \subseteq O_M$; and if the manager itself is at least partly adaptable (i.e., $C_M \cap I_M \neq \emptyset$), a natural requirement to avoid circularities would be that $O_B \cap C_M = \emptyset$, i.e. that the basic component cannot govern the adaptivity of the manager. Composition of ATSs will also be used to model different kinds of aggregation of adaptive systems, like *ensembles* and *swarms*.

Summing up the talk. ATSs are a concrete instance of a methodological approach to white-box adaptation for software systems. More precisely, the CoDa approach we sketched in the first section provides the designer with a criterion to specify where adaptation is located and, as a consequence, which parts of a system have to be adapted. It assumes the possibility to inspect, to some extent, the internal structure of a system, and requires to identify a set of control data, which can be changed to adapt the component's behaviour. Adaptation is the run-time modification of such data.

As described in the second section, ATSS extend interface automata by equipping them with a set of control propositions evaluated on states, which represent the formal counterpart of control data. As for control data, the choice of control propositions is arbitrary but it imposes a clear separation between the ordinary, functional behaviours and the adaptive ones. Control propositions can then be exploited in the specification and analysis of adaptive systems, formally recovering various notions proposed in the literature, such as adaptability, feedback control loops, and control synthesis.

The talk presents ATSS and some applications, and it introduces an explicit representation of *awareness data*, ideally intended as those “sensor” data that are exploited at the control level in order to possibly enforce an adaptation. Awareness and control data complement each other in answering the question regarding *where* and *when* adaptation takes places: A clear identification of awareness data helps selecting which artifacts indicate that it may be necessary to perform an adaptation, and precisely stating when that may occur.

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: ESEC/SIGSOFT FSE 2001. ACM SIGSOFT Software Engineering Notes, vol. 26(5), pp. 109–120. ACM (2001)
2. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A conceptual framework for adaptation. In: de Lara, J., Zisman, A. (eds.) FASE. LNCS, vol. 7212, pp. 240–254. Springer (2012)
3. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Adaptable transition systems. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 95–110. Springer (2013)
4. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: Modelling and analyzing adaptive self-assembly strategies with maude. Science of Computer Programming 99, 75–94 (2015)
5. Bruni, R., Corradini, A., Gadducci, F., Lluch-Lafuente, A., Vandin, A.: A white box perspective on behavioural adaptation. In: Nicola, R.D., Hennicker, R. (eds.) Software, Services, and Systems. LNCS, vol. 8950, pp. 552–581. Springer (2015)
6. Horn, P.: Autonomic Computing: IBM’s perspective on the State of Information Technology (2001)
7. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: PODC 1987. pp. 137–151. ACM (1987)
8. Meseguer, J., Talcott, C.L.: Semantic models for distributed object reflection. In: Magnusson, B. (ed.) ECOOP. LNCS, vol. 2374, pp. 1–36. Springer (2002)

Part II

Long Presentations

Unifying Patterns for Modelling Timed Relationships in Systems and Properties

Étienne André and Laure Petrucci

LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, France

Abstract. Specifying the correctness of complex concurrent and real-time systems is a crucial problem. Many property languages have been proposed to do so; however, these techniques often involve formalisms not easily handled by engineers, and furthermore require dedicated tools. We propose here a set of patterns that encode common specification or verification components when dealing with concurrent real-time systems. We provide a formal semantics for these patterns, as time Petri nets, and show that they can encode previous approaches.

1 Introduction

In the past few decades, many formal languages for specifying and verifying complex concurrent and real-time systems have been proposed. However, these formalisms are not always easy to handle by industry engineers. Temporal logics (*e.g.* [14,6]) offer a very powerful way of expressing correctness properties for concurrent systems but they are often considered too complicated (and maybe too rich as well) to be widely adopted by engineers. Furthermore, they generally need advanced tools dedicated to model checking.

To overcome these difficulties, several pattern-based solutions have been proposed. Patterns allow to identify frequent components of systems or properties in a standardised manner and (sometimes) to compose them so as to build more complex components. Here, we unify three sets of patterns proposed in the past.

In [11], patterns are proposed for modelling scheduling problems; they are then translated into timed automata [4]. In [7], tasks scheduling for operational planning is tackled. A coloured Petri net [10] model is then derived. Both works address the specification of systems [11,7] whereas [5] proposes real-time patterns for verification. These patterns are non-compositional, and do not aim at exhaustiveness; on the contrary, they correspond to common correctness issues met in the literature and in industrial case studies. They are translated to both timed automata [4] and Stateful timed CSP [15], and their verification reduces to simple reachability checking.

Contribution In this paper, we unify previous approaches to propose a pattern-based language for the specification of real-time systems and/or properties for their verification. Each pattern has a syntax as human-readable as possible, so that engineers non-experts in formal methods can use them. Furthermore,

we propose a Time Petri Net [13] semantics of these patterns for both system models and properties. For verification, the patterns are thus translated into pure reachability properties using simple observers, *i.e.* additional subsystems that observe some system actions using synchronisation and may also use time. Hence, their verification in practice avoids the use of complex verification algorithms or dedicated tools, and tool developers can implement them at little cost.

Our patterns can be used for two distinct purposes:

1. specify a system, by means of simple English-like constructs, rather than using complex formalisms. Nevertheless the translation of our patterns into time Petri nets provides a formal model of the system.
2. verify a system (not necessarily specified by our patterns), by means of the same syntax. In this situation, our patterns are again translated into time Petri nets, and can be used to verify the system model by synchronisation on transitions, and using the sole reachability of some “bad” place. This avoids the use of complex model checking algorithms.

Even though the syntax is identical for both purposes, the translation into time Petri nets for verification contains a few more places and transitions.

Related Work Concerning the specification of properties for verifying real-time systems, temporal logics (*e.g.* [14,6]) and their timed extensions (*e.g.* [3] among others) are by far the most commonly used, although many other formalisms have been proposed. Much more expressive than our patterns, temporal logics are more difficult to handle by non-experts. Furthermore, many tools do not actually support their full expressiveness, but only some fragments.

The idea of reducing (some) properties to reachability checking is not new: in [2], safety and bounded-liveness properties are translated into test automata, equivalent to our notion of observers. Among the differences are *i*) the fact that we do not only verify but also specify systems using our patterns, and *ii*) the fact that (as in [5]) we exhibit commonly used patterns, whereas [2] aims at completeness (the expressiveness of such reachability checking has been characterised in [1]).

In [11], typical temporal constraints dedicated to modelling scheduling problems are identified, and then translated into timed automata. In [12], patterns for specifying the system correctness are defined using UML statecharts, and then translated into timed automata. As in our approach, their correctness reduces to reachability checking. Differences include the choice of the target formalism (time Petri nets for us) and the fact that our patterns can encode either the system or its correctness property.

Outline First, Section 2 recalls the earlier definitions of patterns we base on. We then propose our new set of patterns in Section 3, and formalise them using time Petri nets in Section 4. The patterns of [11,7,5] are encoded using our unified patterns in Section 5. Finally, Section 6 concludes and gives perspectives for future work.

2 Earlier Works

Earlier works we base on ([11,7,5]) are concerned on the one hand with causality or timing relations between events, and on the other hand with patterns to encode behavioural properties. Hence they address altogether different aspects (model or property) that are nevertheless very intertwined.

2.1 Scheduling Patterns [11]

In [11], planning constraints are mapped into timed automata, using elementary rules. A set of 17 interval-based temporal relations is defined. As in [11], we group these 17 patterns in 6 categories.

The temporal relations [11] allow for stating timing constraints between tasks. These tasks are composed of two events, that are the start and the end of the task. Time is often specified as an interval (d, D) , to express that an event happens between d and D units of time w.r.t. its reference.

The temporal relations from [11] are the following ones (the words in brackets are added in order to improve readability). The major ones (one per category) are represented in Fig. 1.

1. “ A [ends] before (d, D) B [starts]” (Fig. 1a)
2. “ A meets B ” [*i.e.* A ends exactly when B starts] (degenerated case of rule 1 with $d = D = 0$)
3. “ B [starts] after (d, D) A [ends]” (inverse temporal relation of rule 1)
4. “ B met by A ” [*i.e.* A ends exactly when B starts] (degenerated case of rule 3 with $d = D = 0$, and inverse of 2)
5. “ A starts before (d, D) B [starts]” (Fig. 1b)
6. “ A starts B ” [*i.e.* A starts exactly when B starts] (degenerated case of rule 5 with $d = D = 0$)
7. “ B starts after (d, D) A [starts]” (inverse temporal relation of rule 5)
8. “ A ends before (d, D) B [ends]” (Fig. 1c)
9. “ A ends B ” [*i.e.* A ends exactly when B ends] (degenerated case of rule 8 with $d = D = 0$)
10. “ B ends after (d, D) A [ends]” (inverse temporal relation of rule 8)
11. “ A starts_before_end (d, D) B ”, *i.e.* A starts (d, D) time units before B ends (Fig. 1d)
12. “ T_2 ends_after_start (d, D) T_1 ”, *i.e.* B ends (d, D) time units after A starts (inverse temporal relation of rule 11)
13. “ A contains $((d_1, D_1)(d_2, D_2))$ B ”, *i.e.* A starts (d_1, D_1) time units before B starts, and A ends (d_2, D_2) time units after B ends (Fig. 1e)
14. “ B contained_by $((d_1, D_1)(d_2, D_2))$ A ” (inverse temporal relation of rule 13)
15. “ A equals B ” (degenerated case of rule 13 with $d_1 = D_1 = d_2 = D_2 = 0$)
16. “ A parallels $((d_1, D_1)(d_2, D_2))$ B ”, *i.e.* A starts (d_1, D_1) time units before B starts, and A ends (d_2, D_2) time units before B ends (Fig. 1f)
17. “ B paralleled_by $((d_1, D_1)(d_2, D_2))$ A ” (inverse temporal relation of rule 16)

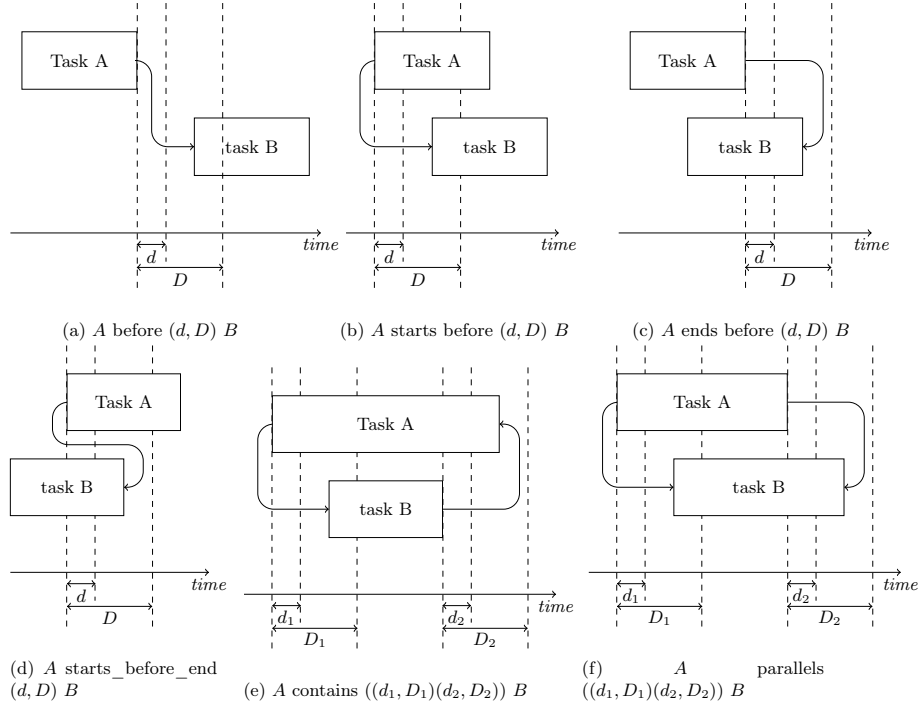


Fig. 1: Some temporal planning constraints

2.2 Patterns for Operational Planning [7]

As in [11], [7] was concerned with tasks scheduling for operational planning, each task having a beginning and an end. Both of these can be timely related to those of another task, as stated in the grammar and figures below.

```
// synchronisation between tasks
synch = BilateralSynch | UnilateralSynch
// lists of tasks
Tasks = task | Tasks, task
```

The synchronisation between tasks can be either bilateral (the execution of any of them is related to the execution of the others), or unilateral (the execution of a task is related to that of the other one, but the converse is not true).

In other words, in a bilateral synchronisation, all tasks occur or none of them does. Note that in [7] tasks can synchronise either at their beginning (*i.e.* they start together) or at their end (*i.e.* they finish at the same time). In this paper, we will be interested in events (beginning or end) and not in a full task. Therefore, bilateral synchronisation only specifies a set of interrelated events.

```
BilateralSynch = BILATERALSYNCH(Tasks)
```

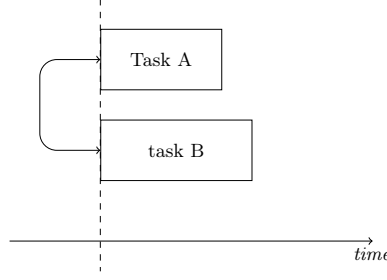



Fig. 2: Bilateral synchronisation between tasks A and B, on their start date.

Fig. 2 depicts the bilateral synchronisation `BILATERALSYNCH(A,B)`.

When the synchronisation is unilateral, a task occurs w.r.t. either a timing or another task, possibly with a delay.

```
UnilateralSynch = Relation(task1, task2, delay)
                  | Relation(task, delay)
Relation = AFTER | BEFORE | AT
```

Examples in Fig. 3 depict the following relations:

- (a) `AT(A,10)`: task A starts at time 10;
- (b) `AFTER(A,10)`: task A starts after time 10;
- (c) `AT(A,B,15)`: task A begins 15 units of time after the end of task B;
- (d) `AFTER(A,B,10)`: task A begins at least 10 units of time after the end of task B.

2.3 Observer Patterns for Real-Time Systems

In [5], we proposed a set of observer patterns encoding common properties encountered when verifying concurrent real-time systems. These patterns are based on observers, hence can be translated into pure reachability problems, thus avoiding the use of complex verification algorithms. These patterns are non-compositional, do not aim at completeness, but rather at exhibiting common properties met in the case studies of the literature.

The main patterns from this section are depicted in Fig. 4 where the arrows show that the occurrence of an event implies the occurrence of the related event.

BeforeDeadline The first pattern relates an event with an absolute timing.

```
BeforeDeadline = a no later than d
```

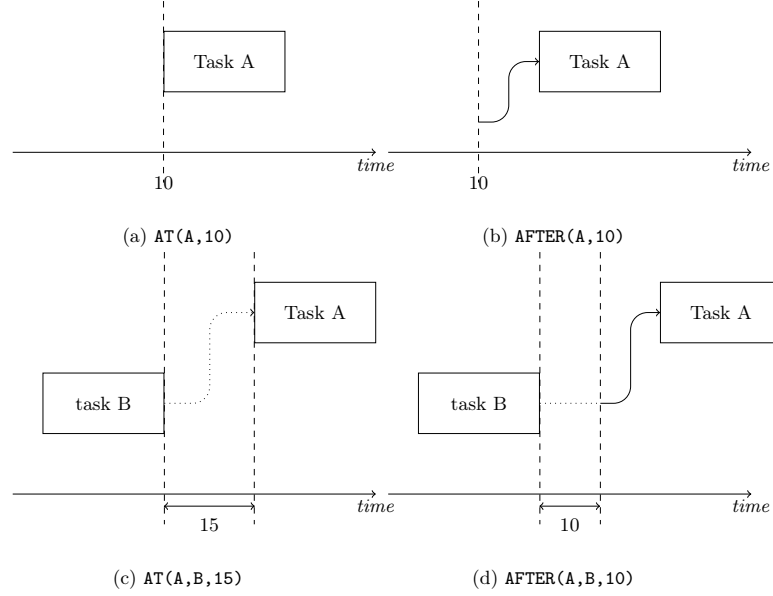


Fig. 3: Some unilateral synchronisations.

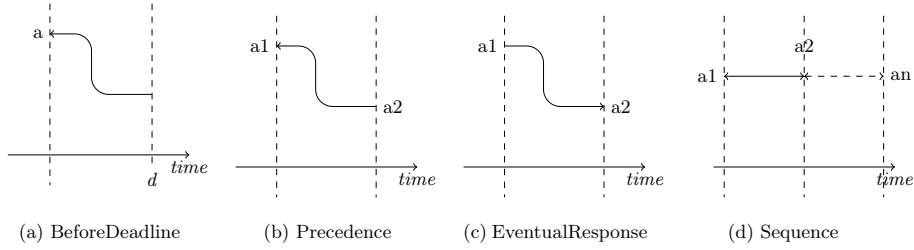


Fig. 4: Some observer patterns for real-time systems

Precedence The following patterns allow for expressing the precedence of the current event by another, with or without an explicit time frame.

The cyclic version of these patterns denotes that a pattern is repeatedly valid, whereas in the strict cyclic version the pattern is not only repeatedly valid, but no event mentioned in the pattern can happen in between (*i.e.* the events mentioned in the pattern are alternating).

The **precedence** pattern requires that, whenever event $a2$ happens, then the event $a1$ must have happened before (at least once). Note that $a2$ is not required to happen. In the **CyclicPrecedence** pattern, every time $a2$ happens, then the event $a1$ must have happened before (at least once) since the last occurrence of $a2$. In the strict cyclic version, every time $a2$ happens, then event $a1$ must have happened exactly once since the last occurrence of $a2$, *i.e.* $a1$ and $a2$ alternate. (They do not need to alternate forever though.) For example, in

the `CyclicPrecedence`, the sequence `a1 a1 a2` can happen but not `a1 a2 a2`, while in the `StrictCyclicPrecedence` none of them can happen.

This pattern is extended to a timed version in a straightforward manner.

```
Precedence = if a2 then a1 has happened before
CyclicPrecedence = everytime a2 then a1 has happened before
StrictCyclicPrecedence = everytime a2 then
    a1 has happened exactly once before

TimedPrecedence = if a2 then
    a1 has happened
    at most d units of time before
CyclicTimedPrecedence = everytime a2 then
    a1 has happened
    at most d units of time before
StrictCyclicTimedPrecedence = everytime a2 then
    a1 has happened exactly once
    at most d units of time before
```

Response Expressing that the current event will be followed by a response is formulated by the following pattern. This pattern is equivalent to the “eventually” in linear temporal logics. None of the two events is required to happen; however, if the first one does, then second must eventually happen too. The cyclic and strictly cyclic versions are defined as for precedence. A timed extension (as in timed temporal logics) is also defined.

```
EventualResponse = if a1 then eventually a2
CyclicEventualResponse = everytime a1 then eventually a2
StrictCyclicEventualResponse = everytime a1 then
    eventually a2 once before next a1

TimedResponse = if a1 then eventually a2 within d
CyclicTimedResponse = everytime a1 then eventually a2 within d
StrictCyclicTimedResponse = everytime a1 then
    eventually a2 within d
    once before next a1
```

Sequence Events can also be ordered as a sequence. None of the n events is required to happen; however, if some (or all) do, then they must follow exactly the order defined by the sequence. The cyclic version is straightforward. However, no strict cyclic version is defined, as it would be identical to the cyclic version.

```
Sequence = SEQUENCE a1, ..., an
CyclicSequence = always SEQUENCE a1, ..., an
```

```

eventlist =
    eventlist EVENT
    | EVENT

interval = (d, D) | [d, D] | [d, D) | (d, D]
timing = WITHIN interval

simplePattern =
    EVENT AT timing
    | EVENT EVENTUALLY timing EVENT
    | EVENT timing AFTER EVENT
    | SEQUENCE (eventlist)

pattern =
    pattern OR simplePattern
    | pattern AND simplePattern
    | ALWAYS simplePattern
    | simplePattern

SYNTACTIC SUGAR:
    AT LEAST d = WITHIN [d, infinity)
    AT MOST d = WITHIN [0, d]
    EXACTLY d = WITHIN [d, d]

```

Fig. 5: A grammar for unified patterns

Unreachability The last pattern of [5] is rather different from others, as it only expresses the model safety (*i.e.* non-reachability of a undesired state). It was considered in [5] because this property is by far the most commonly met in case studies from the literature, and because all other patterns can be reduced to (non-)reachability.

Unreachable = UNREACHABLE(Bad)

3 Towards a More Complete Patterns Language

The primitives in the grammars of Section 2.1 and Section 2.2 are dedicated to temporal or causal relations between tasks which are characterised by both their starting and ending times. But, in practice, most systems are concerned with individual events, as is the case in [5]. We present in this section a unified version of patterns previously introduced. We will show in Section 5 that our patterns subsume the primitives from Section 2.

We introduce a grammar for unified patterns Fig. 5. Our grammar considers individual *events* that can form a *list of events* in order to construct a *sequence*. The *timing* of events can be specified as being *within* a time frame (from *d* to

D time units, where $d \in \mathbb{R}_+$ and $D \in \mathbb{R}_+ \cup \{\infty\}$). The only restriction is that the interval $[d, D] \neq [0, \infty)$ (see Remark 1 *infra*).

Simple patterns express basic relations between individual events. An event can happen w.r.t. an absolute *timing* constraint. An event can *eventually* entail the occurrence of another event w.r.t. some *timing constraint*. Conversely, an event can occur only w.r.t. a timing *after* another event already occurred. Events can be ordered in a sequence, thus all occurring one after another.

Simple Patterns The **simple patterns** contain four kinds of relationships, that we describe in more details in the following.

The pattern fragment **EVENT AT** encodes an absolute timing; it is used to describe events that must happen exactly at an (absolute) time.

The pattern fragment **EVENT EVENTUALLY timing EVENT** encodes that, whenever the first event happens, then the second will eventually happen, with the timing constraint specified by **timing**. That is, if the first event happens, the second must happen. The converse is not true: if the second event happens, the first one did not necessarily happen before.

The pattern fragment **EVENT timing AFTER EVENT** encodes that, whenever the first event happens it is necessarily after the second one, together with some timing constraint. For example, **e2 WITHIN(d,D) AFTER e1** denotes that **e2** may or may not happen, but if **e2** happens, then it must be at least d and at most D time units after the first occurrence of **e1**. Also note that, if **e2** does not happen, then **e1** may or may not happen.

Finally, the **SEQUENCE** ensures that a list of events happen in the particular order specified.

Complex Patterns Patterns can be combined in order to form more complex ones. First, we can use Boolean **AND** and **OR** operators to express the conjunction of patterns (*i.e.* both must be executed, for patterns expressing systems, or must be valid, for patterns expressing the properties) or the disjunction (*i.e.* either one of them can be executed/valid).

The **ALWAYS** is a sort of fixpoint, with a semantics similar as the notion of “cyclic” pattern in [5]. That is, once the pattern has been executed / verified, then it must again be executed / verified. This typically describes a cyclic behaviour. We restrict here the **ALWAYS** pattern to simple patterns (**simple pattern** in Fig. 5 and not, *e.g.* **pattern**). The reason is on the one hand to keep our language simple¹, and on the other hand to make a translation to time Petri nets relatively easy.

Finally, it is often convenient to use some syntactic sugar for expressing timing constraints: **AT LEAST**, **AT MOST** and **EXACTLY**.

Remark 1 (untimed patterns). We could encode untimed patterns using our timed patterns (by allowing a syntactic sugar construct **UNTIMED** = **WITHIN** $[0,$

¹ Furthermore, while designing the patterns in [5], such **ALWAYS**-like properties were only encountered in the literature on (very) simple patterns.

infinity)), but we leave it out so as to keep the exposé simple. Indeed, although this does not bring theoretical problems, the translation of the untimed patterns into time Petri nets for verification purposes then exceeds the set of properties that can be checked using sole unreachability. In particular, the negation of the untimed “eventually” construct cannot be checked using the unreachability of a “bad” state, but it becomes necessary to additionally check the reachability of some “good state”; this was performed in [5]. Here, to keep the translation simple, we temporarily leave out the untimed patterns. Formalising the untimed patterns for the verification purpose will be performed in an extended version of this work.

4 Semantics: Translation to Time Petri Nets

Time Petri nets [13] are Petri nets where transition are equipped with a time interval, that specifies the minimum and maximum time for the transition to be enabled before it actually fires. The different patterns in the grammar of Fig. 5 are modelled as time Petri nets in Fig. 6a–6h. Let us now describe our translation. We start with simple (*i.e.* non-compositional) patterns, and then go for complex patterns (*i.e.* that rely on others).

Observers Let us recall the concept of observers, as formalised in [5]. Observers are standard subsystems, with some assumptions. An observer must not have any effect on the system, and must not prevent any behaviour to occur. In particular, it must not block time, nor prevent actions to occur, nor create deadlocks that would not occur otherwise. As a consequence, observers must be complete: in the example of timed automata, all actions declared by the observer must be allowed in any of the locations. Similarly, in time Petri nets, an observer must be able to synchronise at any time with any of the actions used on its transitions.

General Idea of our Translation Recall that our patterns aim at encoding both systems and properties. Although they are defined in a unified manner in Section 3, they must be differentiated when formalised using time Petri nets. Indeed, our patterns seen as properties reduce verification to simple reachability analysis (as in [2,1,5]).

For the verification, we define a “bad” place (labelled in Fig. 6a–6h using the “☹” symbol); this place is assumed to be unique, *i.e.* one must *fuse* all occurrences of this place when composing patterns. The verification can then be carried out as follows: given a model of the system specified using time Petri nets (but not necessarily specified using our specification patterns), and given a property of the system specified using our patterns and translated into a time Petri nets, we perform the synchronisation (on transitions) of the entire system. Then, the property (expressed by the pattern) is satisfied iff the “☹” place is unmarkable, *i.e.* cannot be marked in any marking of the synchronised net.

In order to differentiate between the specification of systems and the specification of properties, we depict in dotted red the places and transitions necessary

to add to our translated patterns so as to be able to perform verification. In other words, these dotted red places and transitions shall be omitted when specifying systems and not properties. Conversely, we depict in plain light blue the places and transitions only necessary for the system specification, but that must be omitted for the verification.

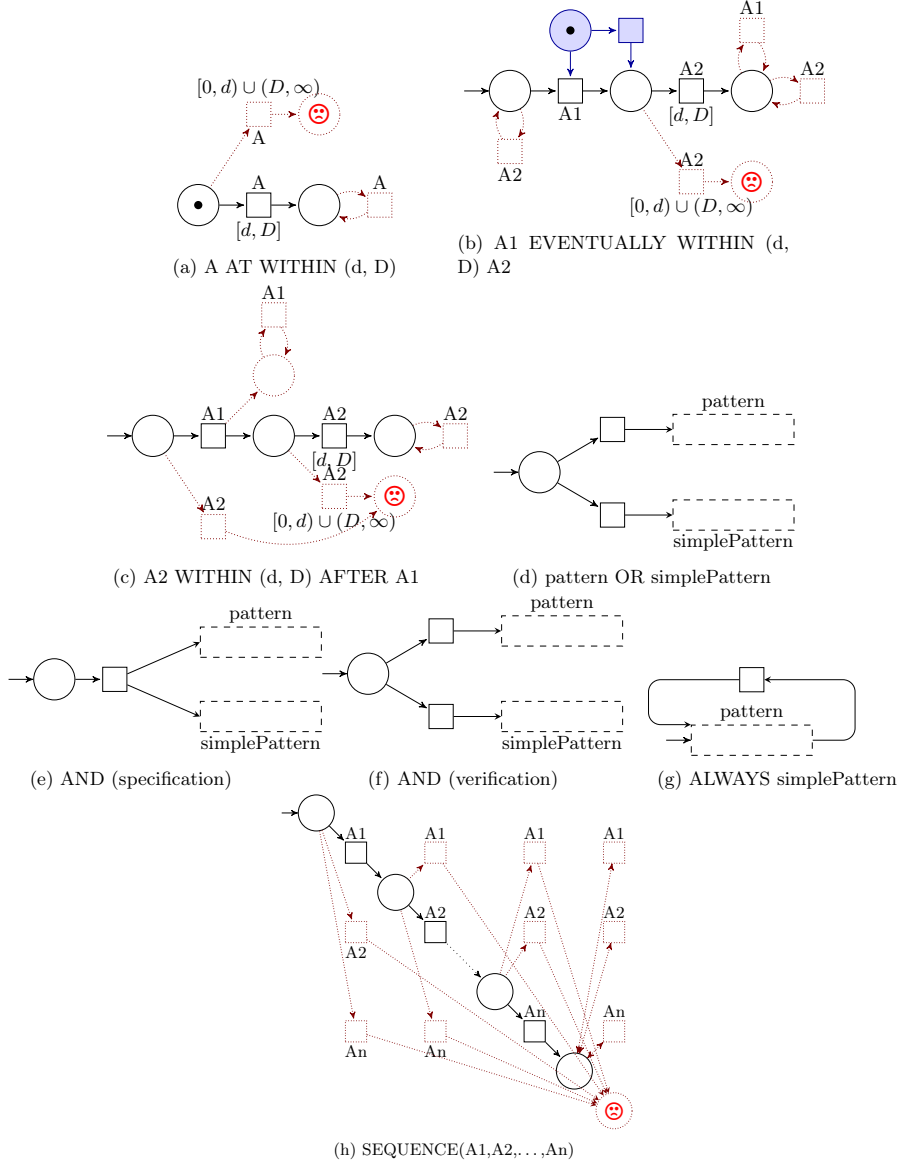


Fig. 6: Translation of our patterns into time Petri nets

Simple Patterns Fig. 6a gives the translation of the $A \text{ AT WITHIN } (d, D)$ pattern. For the property, the translation is straightforward: a place is followed by a timed transition with firing time $[d, D]$ labelled with A . This correctly encodes the fact that A must occur within $[d, D]$ after the system start. (We assume closed intervals in the remainder of the translation; open or semi-open intervals can be handled similarly.) Concerning the verification, in addition to the correct behaviour, we need also to specify the bad behaviour, hence in this case another transition that can occur only if the timing is not satisfied, leading to the “bad” place. That is, the bad place is reachable iff the property is violated. Additionally, for our pattern to be a good “observer” (*i.e.* that must not disturb the system), it must be able to synchronise at any time with the system on the transition the pattern declares (here only A). This explains the loop on transition A on the right-hand side of Fig. 6a. (In fact, self-loops should also be added to the bad place; we omit them for sake of space, but also because it is less important to block the system once the property has been proved invalid.)

Pattern $A1 \text{ EVENTUALLY WITHIN } (d, D) A2$ is modelled by the TPN in Fig. 6b. For the specification, two cases are admitted: one where $A1$ is fired, and one where it is not. Moreover the possibility of firing $A1$ depends on other actions in the system which may put a token in the initial place of the pattern (depicted by an incoming arrow). The additional places and transitions for the verification counterpart of this pattern are explained as follows: the first self-loop allows $A2$ to happen anytime as long as $A1$ has not happened. Then, if $A2$ happens strictly before or strictly after $[d, D]$, the observer enters the bad place. Otherwise, the property is satisfied, and both $A1$ and $A2$ can happen anytime, which is depicted using the two self-loops.

The $A2 \text{ WITHIN } (d, D) \text{ AFTER } A1$ pattern (given in Fig. 6c) is similar to the previous one but, in this case, it is not possible to have $A2$ without $A1$. As for the verification, note that $A2$ cannot occur before $A1$, hence the transition between the initial place and the bad place. Furthermore, several $A1$ may occur before $A2$ occurs: this is encoded using the second output from $A1$ and a self-loop. Thus the time for firing $A2$ is counted from the first occurrence of $A1$. The rest of the pattern is similar to the previous one.

The $\text{SEQUENCE}(A1, A2, \dots, An)$ pattern is given in Fig. 6h. Naturally, it is made of a sequence of transitions. Additionally, the verification version is such that, as soon as a transition violates the order imposed by the sequence, the system goes to the bad place. An additional self-loop in the last good place, synchronising on any transition, makes the observer non-blocking.

Complex Patterns These patterns are used to combine the previous ones (eventually with complex patterns as well). In Fig. 6d to Fig. 6g, they are pictured in dashed boxes, which would also include the “bad” place. For the specification, the complex patterns are straightforward: they syntactically combine existing patterns. For the verification, this is a little less simple: first, recall that all “bad” locations must be fused into a single one. Second, the “and” verification pattern becomes identical to the... “or” specification pattern: this is because the property $P1 \text{ AND } P2$ is violated if $P1$ is violated (*i.e.* the bad place is reachable in the

corresponding pattern) *or* P2 is violated. The “or” pattern is not translated for verification; this is because this cannot be checked with sole unreachability (see Section 6). Concerning the **ALWAYS** pattern for verification, one must fuse the last non-dotted place of the pattern (usually the right-most place in the figures) with the initial place. However, the self-loops (generally on **A1** and **A2**) on the last non-dotted place must be removed.

Initial Marking In addition to the tokens introduced by the absolute time patterns (**A AT WITHIN** (d, D)), a single initial token must be put in the top-most pattern of the composed pattern expression. (We assume that, if the top-most expression is a pattern **A AT WITHIN** (d, D), then no further token is added.)

5 Encoding Previous Patterns Using our Unified Patterns

In this section we show how all primitives from Section 2 can be expressed using our new set of patterns.

In order to express the relations between tasks described in Section 2.1 and Section 2.2 with these new primitives, a task **A** is transformed into two events, specifying the task Beginning (**A.start**) and its end (**A.end**).

5.1 Encoding Patterns from [11]

The expression of patterns from Section 2.1 is summarised in Table 1.

Note that several rules are expressed identically. For example, rule 2 is reflecting the point of view from event **A**, and rule 4 the one of event **B**, while in our patterns we express the relation as seen from an external observer.

5.2 Encoding Patterns from [7]

Table 2 shows the mapping for the patterns of Section 2.2.

Note that formula 5 implies that if **B** does not occur, neither does **A**. On the contrary, if **B** occurs, **A** can occur or not. If it does, it is d units of time after **B** ended. A similar remark applies to formula 6. Finally, in formula 7, **A** necessarily occurs d units of time after the end of **B**.

5.3 Encoding Patterns from [5]

Patterns from Section 2.3 are presented in Table 3. Our translation is straightforward. The only “trick” is the translation of the “strict cyclic” patterns of [5], that are encoded using both the cyclic version of these patterns (using **ALWAYS**) and the **SEQUENCE** pattern, that requires **A1** and **A2** to alternate.

6 Conclusion

We proposed a unified pattern mechanism to both specify and verify real-time systems, together with a semantics using time Petri nets. Our new set of patterns unifies the patterns of [11,7,5] into a single homogeneous pattern language.

Rule 1	A [ends] before (d, D) B [starts]	$B.start$ WITHIN (d, D) AFTER $A.end$
Rule 2	A meets B	$B.start$ EXACTLY 0 AFTER $A.end$
Rule 3	B [starts] after (d, D) A [ends]	$B.start$ WITHIN (d, D) AFTER $A.end$
Rule 4	B met by A	$B.start$ EXACTLY 0 AFTER $A.end$
Rule 5	A starts before (d, D) B [starts]	$B.start$ WITHIN (d, D) AFTER $A.start$
Rule 6	A starts B	$B.start$ EXACTLY 0 AFTER $A.start$
Rule 7	B starts after (d, D) A [starts]	$B.start$ WITHIN (d, D) AFTER $A.start$
Rule 8	A ends before (d, D) B [ends]	$B.end$ WITHIN (d, D) AFTER $A.end$
Rule 9	A ends B	$B.end$ EXACTLY 0 AFTER $A.end$
Rule 10	A ends before (d, D) B [ends]	$B.end$ WITHIN (d, D) AFTER $A.end$
Rule 11	A starts before end (d, D) B	$B.end$ WITHIN (d, D) AFTER $A.start$
Rule 12	B ends after start (d, D) A	$B.end$ WITHIN (d, D) AFTER $A.start$
Rule 13	A contains $((d_1, D_1)(d_2, D_2))$ B	$B.start$ WITHIN (d_1, D_1) AFTER $A.start$ AND $A.end$ WITHIN (d_2, D_2) AFTER $B.end$
Rule 14	B contained by $((d_1, D_1)(d_2, D_2))$ A	$B.start$ WITHIN (d_1, D_1) AFTER $A.start$ AND $A.end$ WITHIN (d_2, D_2) AFTER $B.end$
Rule 15	A equals B	$B.start$ EXACTLY 0 AFTER $A.start$ AND $B.end$ EXACTLY 0 AFTER $A.end$
Rule 16	A parallels $((d_1, D_1)(d_2, D_2))$ B	$B.start$ WITHIN (d_1, D_1) AFTER $A.start$ AND $B.end$ WITHIN (d_2, D_2) AFTER $A.end$
Rule 17	B paralleled by A $((d_1, D_1)(d_2, D_2))$	$B.start$ WITHIN (d_1, D_1) AFTER $A.start$ AND $B.end$ WITHIN (d_2, D_2) AFTER $A.end$

Table 1: Encoding patterns from [11]

Future Works First, translating the untimed **EVENTUALLY** and the **OR** patterns for verification purposes is in our agenda; this will be done by checking, not only the unreachability of the bad, but also the reachability of a good place.

Second, more patterns from the literature should be integrated to our encoding. Although we shall not develop too complex a pattern system, so as to avoid giving birth to a complicated property language, the patterns in [12] seem interesting to us. Furthermore, the patterns of [9] seem to fit directly in our unified pattern systems, but this should be shown formally. It would also be interesting to formally compare the expressiveness of our patterns with [2,1] or (subsets of) temporal logics such as LTL/CTL.

Third, although it is relatively easy to convince oneself that we correctly encoded the patterns of [11,7,5], formally proving their semantic equivalence would be interesting. It would also be nice to provide tool support, helping a designer to write patterns to model a system and its properties.

Finally, our translation to time Petri nets was done manually. An alternative option would be to define an *ad-hoc* domain specific language (DSL), and then to use model transformation techniques (such as in [8]) to obtain time Petri nets.

Acknowledgement

We are grateful to the anonymous reviewers for useful comments.

Formula 1	BILATERALSYNCH(A,B)	A.start EXACTLY 0 AFTER B.start OR A.end EXACTLY 0 AFTER B.end
Formula 2	AT(A,d)	A.start AT EXACTLY d
Formula 3	AFTER(A,d)	A.start AT AT LEAST d
Formula 4	BEFORE(A,d)	A.start AT AT MOST d
Formula 5	AT(A,B,d)	A.start EXACTLY d AFTER B.end
Formula 6	AFTER(A,B,d)	A.start AT LEAST d AFTER B.end
Formula 7	BEFORE(A,B,d)	B.end EVENTUALLY AT MOST d A.start

Table 2: Encoding patterns from [7]

References

1. Luca Aceto, Patricia Bouyer, Augusto Burgueño, and Kim Guldstrand Larsen. The power of reachability testing for timed automata. In Vikraman Arvind and Ramaswamy Ramanujam, editors, *FSTTCS*, volume 1530 of *LNCS*, pages 245–256. Springer, 1998.
2. Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In *TACAS*, volume 1384 of *LNCS*, pages 263–280. Springer, 1998.
3. Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.
4. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
5. Étienne André. Observer patterns for real-time systems. In *ICECCS*, pages 125–134. IEEE Computer Society, 2013.
6. Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.
7. Sébastien Bardin and Laure Petrucci. COAST : des réseaux de Petri à la planification assistée. In *AFADL*, pages 285–298, 2004.
8. Julien DeAntoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau, and Benoît Combemale. Towards a meta-language for the concurrency concern in DSLs. In *DATE*, pages 313–316. ACM, 2015.
9. Jin Song Dong, Ping Hao, Shengchao Qin, Jun Sun, and Wang Yi. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.
10. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
11. Lina Khatib, Nicola Muscettola, and Klaus Havelund. Mapping temporal planning constraints into timed automata. In *TIME*, pages 21–27. IEEE Computer Society, 2001.
12. Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*, pages 112–124. British Computer Society, 2009.
13. Philip Meir Merlin. *A study of the recoverability of computing systems*. PhD thesis, University of California, Irvine, 1974.
14. Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.
15. Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using Stateful Timed CSP. *ACM Transactions on Software Engineering and Methodology*, 22(1):3.1–3.29, 2013.

Formula 1	Precedence = if A2 then A1 has happened before	A2 AT LEAST 0 AFTER A1
Formula 2	CyclicPrecedence = everytime A2 then A1 has happened before	ALWAYS (A2 AT LEAST 0 AFTER A1)
Formula 3	StrictCyclicPrecedence = everytime A2 then A1 has happened exactly once before	ALWAYS SEQUENCE(a1,a2)
Formula 4	EventualResponse = if A1 then eventually A2	A1 EVENTUALLY AT LEAST 0 A2
Formula 5	CyclicEventualResponse = everytime A1 then eventually A2 (before next A1)	ALWAYS (A1 EVENTUALLY AT LEAST 0 AFTER A2)
Formula 6	StrictCyclicEventualResponse = everytime A1 then eventually A2 once before next A1	ALWAYS (A1 EVENTUALLY AT LEAST 0 A2) AND SEQUENCE(A1,A2)
Formula 7	BeforeDeadline = A no later than d	A AT AT MOST d
Formula 8	TimedPrecedence = if A2 then A1 has happened at most d units of time before	A2 AT MOST d AFTER A1
Formula 9	CyclicTimedPrecedence = everytime A2 then A1 has happened at most d units of time before	ALWAYS (A2 AT MOST d AFTER A1)
Formula 10	StrictCyclicTimedPrecedence = everytime A2 then A1 has happened exactly once at most d units of time before	ALWAYS (A2 AT MOST d AFTER A1) AND ALWAYS SEQUENCE(A1, A2)
Formula 11	TimedResponse = if A1 then eventually A2 within d	A1 EVENTUALLY AT MOST d A2
Formula 12	CyclicTimedResponse = everytime A1 then eventually A2 within d	ALWAYS (A1 EVENTUALLY AT MOST d A2)
Formula 13	StrictCyclicTimedResponse = everytime A1 then eventually A2 within d once before next A1	ALWAYS (A1 EVENTUALLY AT MOST d A2) AND ALWAYS SEQUENCE (A1, A2)
Formula 14	Sequence = sequence A1, ..., An	SEQUENCE(A1,A2,...,An)
Formula 15	CyclicSequence = always sequence A1, ..., An	ALWAYS SEQUENCE(A1,A2,...,An)

Table 3: Encoding patterns from [5]

Negotiations and Petri Nets

Jörg Desel¹ and Javier Esparza²

¹ Fakultät für Mathematik und Informatik, FernUniversität in Hagen, Germany
`joerg.desel@fernuni-hagen.de`

² Fakultät für Informatik, Technische Universität München, Germany
`esparza@in.tum.de`

Abstract. Negotiations have recently been introduced as a model of concurrency with multi-party negotiation atoms as primitive. This paper studies the relation between negotiations and Petri nets. In particular, we show that each negotiation can be translated into a 1-safe labelled Petri net with equivalent behaviour. In the general case, this Petri net is exponentially larger than the negotiation. For deterministic negotiations however, the corresponding Petri net has linear size compared to the negotiation, and it enjoys the free-choice property. We show that for this class the negotiation is sound if and only if the corresponding Petri net is sound. Finally, we have a look at the converse direction; given a Petri net; can we find a corresponding negotiation?

Keywords: Negotiations, Petri nets, soundness, free-choice nets

1 Introduction

Distributed negotiations have been identified as a paradigm for process interaction since some decades, in particular in the context of multi-agent systems. A distributed negotiation is based on a set of agents that communicate with each other to eventually reach a common decision. It can be viewed as a protocol with atomic negotiations as smallest elements. Multiparty negotiations can employ more than two agents, both in the entire negotiation and in its atoms. A natural way to formally model distributed negotiations is to model the behaviour of the agents separately and then to model the communication between agents by composition of these agent models. Petri nets and related process languages have been used with this aim, see e.g. [2, 8, 7].

In [4, 5] we have introduced a novel approach to formally model negotiations. We argue that this model is sometimes more intuitive than Petri nets for negotiations, but it can also be applied to other application areas which are based on the same communication principles. Like Petri nets, our formalism has a graphical representation. *Atomic negotiations* are represented as nodes, with a specific representation of the participating agents. Roughly speaking, the semantics of a negotiation atom is that these agents, called participants of the atom, come together (and are thus not distributed and do not need any communication means during the atomic negotiation) to agree on one of some possible

outcomes. Given an outcome, the model specifies, for each participating agent, the next possible atomic negotiations in which it can participate. Agents have local states which are only changed when an agent participates in a negotiation. Atomic negotiations are combined into *distributed negotiations*. The state of a distributed negotiation is determined by the atomic negotiations which the agents can participate in next and by all local states. As in Petri nets, these two aspects are carefully distinguished; the current next possible atomic negotiations are represented as *markings* of negotiations.

Our previous contributions [4, 5] concentrate on the analysis of negotiations. In particular, we studied the efficient analysis of well-behavedness of negotiations by means of structural reduction rules. Our work was inspired by known reduction rules of Petri nets but leads to significantly better results when a translation to Petri nets is avoided, at least for the general case. The present paper makes the relation to Petri nets explicit, providing a translation rule from distributed negotiations to Petri nets. It turns out that, for restricted classes of negotiations, the corresponding Petri nets enjoy nice properties, and in this case the converse direction is possible, too.

The paper is organised as follows. Section 2 repeats the syntax and semantics of negotiations. Section 3 provides the translation to Petri nets with the same behaviour. Section 4 discusses properties of these Petri nets. In Section 5 we show that Petri nets enjoying these properties can be translated back to negotiations, this way characterizing a class of Petri nets representable by negotiations.

2 Negotiations: Syntax and Semantics

We recall the main definitions of [4, 5] for syntax and semantics of negotiations. Let A be a finite set (of *agents*), representing potential parties of a negotiation. Each agent $a \in A$ has a (possibly infinite) nonempty set Q_a of *internal states* with a distinguished subset $Q_{0a} \subseteq Q_a$ of *initial states*. We denote by Q_A the cartesian product $\prod_{a \in A} Q_a$. So a state is represented by a tuple $(q_{a_1}, \dots, q_{a_{|A|}}) \in Q_A$. A *transformer* is a left-total relation $\tau \subseteq Q_A \times Q_A$, representing a nondeterministic state transforming function. Given $S \subseteq A$, we say that a transformer τ is an *S-transformer* if, for each $a_i \notin S$, $((q_{a_1}, \dots, q_{a_i}, \dots, q_{a_{|A|}}), (q'_{a_1}, \dots, q'_{a_i}, \dots, q'_{a_{|A|}})) \in \tau$ implies $q_{a_i} = q'_{a_i}$. So an *S-transformer* only transforms internal states of agents in S or in a subset of S .

Internal states of agents and their transformers won't play an important role in this contribution. As will become clear later, states do not influence behaviour in negotiations, i.e., we can consider the control flow and data aspects separately. For the Petri net translation to be defined, local states and their transformers can be modelled by means of token colours and transition modes, respectively, i.e. by means of Coloured Petri nets. These Coloured Petri nets are without guards, because guards restrict transition occurrences by regarding data values.

2.1 Atomic Negotiations

Definition 1. An atomic negotiation, or just an atom, over a set of agents A is a triple $n = (P, R, \delta)$, where $P \subseteq A$ is a nonempty set of parties or participants of n , R is a finite, nonempty set (results), and δ is a mapping assigning to each result $r \in R$ a P -transformer $\delta(r)$.

In the sequel, P_n , R_n and δ_n will denote the components of an atom n . For each result $r \in R_n$, the pair (n, r) is called an *outcome*. The difference between results and outcomes is that the same result can belong to different atoms whereas the sets of outcomes are pairwise disjoint. If we choose disjoint sets for the respective sets of results then we do not have to distinguish results and outcomes.

If the states of the agents before an atomic negotiation n are given by a tuple q and the result of the negotiation is r , then the agents change their states to q' for some $(q, q') \in \delta_n(r)$. Only the parties of n can change their internal states. However, it is not required that a P_n -transformer $\delta_n(r)$ actually changes the states of all agents in P_n . Each result $r \in R_n$ is possible, independent of the previous internal states of the parties of n .

As a simple example, consider an atomic negotiation n_{FD} with parties F (Father) and D (teenage Daughter). The goal of the negotiation is to determine whether D can go to a party, and the time (a number between 8 and 12) at which she must return home. The possible results are **{yes, no, ask_mother}**. Both sets Q_F and Q_D contain a state *angry* plus a state t for every time $T_1 \leq t \leq T_2$ in a given interval $[T_1, T_2]$. The transformer $\delta_{n_{FD}}$ includes

$$\begin{aligned} \delta_{n_{FD}}(\mathbf{yes}) &= \{ ((t_f, t_d), (t, t)) \mid t_f \leq t \leq t_d \vee t_d \leq t \leq t_f \} \\ \delta_{n_{FD}}(\mathbf{no}) &= \{ ((t_f, t_d), (angry, angry)) \} \\ \delta_{n_{FD}}(\mathbf{ask_mother}) &= \{ ((t_f, t_d), (t_f, t_d)) \} \end{aligned}$$

where t_f and t_d are variables used to denote that F is in state $t_f \neq \mathbf{angry}$ and D in state $t_d \neq \mathbf{angry}$ before engaging in the negotiation atom n_{FD} . Moreover, if one of the local states before the negotiation atom was *angry*, then $\delta_{n_{FD}}$ specifies that both agents will be *angry* after executing the atom.

If both parties are not *angry* and the result is **yes**, then F and D agree on a time t which is not earlier and not later than both suggested times. If it is **no**, then there is a quarrel and both parties get angry. If it is **ask_mother**, then the parties keep their previous times.

2.2 Combining Atomic Negotiations

If the result of the atomic negotiation above is **ask_mother**, then n_{FD} is followed by a second atomic negotiation n_{DM} between D and M (Mother). The combined negotiation is the composition of n_{FD} and n_{DM} , where the possible internal states of M are the same as those of F and D, and n_{DM} is a “copy” of n_{FD} , but without the **ask_mother** result. In order to compose atomic negotiations, we add a *transition function* \mathcal{X} that assigns to every triple (n, a, r) consisting of an atom n , a participant a of n , and a result r of n a set $\mathcal{X}(n, a, r)$ of atoms. Intuitively, this

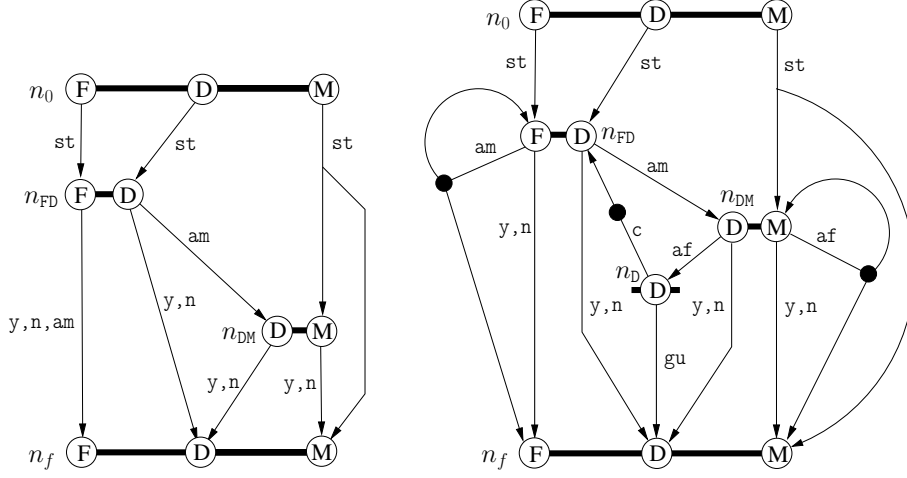


Fig. 1. An acyclic negotiation and the ping-pong negotiation.

is the set of atomic negotiations agent a is ready to engage in after the atom n , if the result of n is r .

Definition 2. Given a finite set of agents A and a finite set of atoms N over A , let $T(N)$ denote the set of triples (n, a, r) such that $n \in N$, $a \in P_n$, and $r \in R_n$. A (distributed) negotiation is a tuple $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$, where $n_0, n_f \in N$ are the initial and final atoms, and $\mathcal{X}: T(N) \rightarrow 2^N$ is the transition function. Further, \mathcal{N} satisfies the following properties:

- (1) every agent of A participates in both n_0 and n_f ;
- (2) for every $(n, a, r) \in T(N)$: $\mathcal{X}(n, a, r) = \emptyset$ iff $n = n_f$.

The graph associated with \mathcal{N} has vertices N and edges

$$\{(n, n') \in N \times N \mid \exists (n, a, r) \in T(N): n' \in \mathcal{X}(n, a, r)\}.$$

The initial and final atoms mark the beginning and the end of the negotiation (and sometimes this is their only role). We may have $n_0 = n_f$. In this case, due to (2), $N = \{n_0\}$, i.e., the negotiation has only one single atom. Notice that n_f has, as all other atoms, at least one result $\text{end} \in R_{n_f}$.

2.3 Graphical Representation of Negotiations

Negotiations are graphically represented as shown in Figure 1. For each atom $n \in N$ we draw a bar; for each participant a of P_n we draw a circle on the bar, called a *port*. For each $(n, a, r) \in T(N)$ with $n \neq n_f$, a hyperarc leads from the port of a in n to all the ports of a in the atoms of $\mathcal{X}(n, a, r)$, labelled by the result r . Figure 1 shows on the left the graphical representation of a negotiation where Father (F), Daughter (D) and Mother (M) are the involved agents. After the initial atom n_0 , which has only one possible result **st** (**start**), the negotiation atoms

sketched above take place. Notice that possibly Father and Daughter come to an agreement without involving Mother. So the agents of a negotiation can be viewed as potential participants, which necessarily participate only in the initial and the final atom. Instead of multiple (hyper)arcs connecting the same input port to the same output ports we draw a single (hyper)arc with multiple labels. In the figure, we write **y** for **yes**, **n** for **no**, and **am** for **ask mother**. Since n_f has no outgoing arc, the results of n_f do not appear in the graphical representation.

The negotiation on the right (ignore the black dots on the arcs for the moment) is the ping-pong negotiation, well-known in every family. The n_{DM} atom has now an extra result **ask_father** (**af**), and Daughter can be sent back and forth between Mother and Father. After each round, D “negotiates with herself” (atom n_D) with possible outcomes **continue** (**c**) and **give up** (**gu**).

2.4 Semantics

A *marking* of a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ is a mapping $\mathbf{x}: A \rightarrow 2^N$. Intuitively, $\mathbf{x}(a)$ is the set of atoms that agent a is currently ready to engage in next. The *initial* and *final* markings, denoted by \mathbf{x}_0 and \mathbf{x}_f , are given by $\mathbf{x}_0(a) = \{n_0\}$ and $\mathbf{x}_f(a) = \emptyset$ for every $a \in A$. Obviously, the set of markings is finite.

A marking \mathbf{x} *enables* an atom n if $n \in \mathbf{x}(a)$ for every $a \in P_n$, i.e., if every agent that participates in n is currently ready to engage in n . If \mathbf{x} enables n , then n can take place and its participants agree on a result r ; we say that the outcome (n, r) *occurs*. The occurrence of (n, r) produces a next marking \mathbf{x}' given by $\mathbf{x}'(a) = \mathcal{X}(n, a, r)$ for every $a \in P_n$, and $\mathbf{x}'(a) = \mathbf{x}(a)$ for every $a \in A \setminus P_n$.

We write $\mathbf{x} \xrightarrow{(n,r)} \mathbf{x}'$ to denote this, and call it a *small step*.

We write $\mathbf{x}_1 \xrightarrow{\sigma}$ to denote that there is a sequence

$$\mathbf{x}_1 \xrightarrow{(n_1, r_1)} \mathbf{x}_2 \xrightarrow{(n_2, r_2)} \dots \xrightarrow{(n_{k-1}, r_{k-1})} \mathbf{x}_k \xrightarrow{(n_k, r_k)} \mathbf{x}_{k+1} \dots$$

of small steps such that $\sigma = (n_1, r_1) \dots (n_k, r_k) \dots$. If $\mathbf{x}_1 \xrightarrow{\sigma}$, then σ is an *occurrence sequence* from the marking \mathbf{x}_1 , and \mathbf{x}_1 enables σ . If σ is finite, then we write $\mathbf{x}_1 \xrightarrow{\sigma} \mathbf{x}_{k+1}$ and say that \mathbf{x}_{k+1} is *reachable* from \mathbf{x}_1 . If \mathbf{x}_1 is the initial marking then we call σ *initial occurrence sequence*. If moreover \mathbf{x}_{k+1} is the final marking \mathbf{x}_f , then σ is a *large step*.

As a consequence of this definition, for each agent a , $\mathbf{x}(a)$ is always either $\{n_0\}$ or equals $\mathcal{X}(n, a, r)$ for some outcome (n, r) . The marking \mathbf{x}_f can only be reached by the occurrence of (n_f, end) (**end** being a possible result of n_f), and it does not enable any atom.

Reachable markings can be graphically represented by placing tokens (black dots) on the forking points of the hyperarcs (or in the middle of an arc). Thus, both the initial marking and the final marking are represented by no tokens, and all other reachable markings are represented by exactly one token per agent.

Figure 1 shows on the right the marking in which Father is ready to engage in the atomic negotiations n_{FD} and n_f , Daughter is only ready to engage in n_{FD} , and Mother is ready to engage in both n_{DM} and n_f .

As mentioned before, the enabledness of an atom does not depend on the internal states of the agents involved; it suffices that all agents are ready to engage in this atom, no matter which internal states they have. Moreover, each result of the atom is possible, independent from the internal states. A given result then determines a state transformer and thus possible next states.

2.5 Reachability Graphs

As known from any family, an occurrence sequence of a negotiation can be arbitrarily long (see the ping-pong negotiation above). Therefore, the set of possible occurrence sequences can be infinite. Since we have markings and steps, an obvious way to describe behaviour with finite means is by reachability graphs:

Definition 3. *The reachability graph of a negotiation \mathcal{N} has all markings reachable from \mathbf{x}_0 as vertices, and an arc leading from \mathbf{x} to \mathbf{x}' and annotated by (n, r) whenever $\mathbf{x} \xrightarrow{(n, r)} \mathbf{x}'$. The initial marking \mathbf{x}_0 is the distinguished initial vertex.*

Generally, atoms with disjoint sets of parties can proceed concurrently, whereas atoms sharing a party cannot. Formally, if two outcomes (n_1, r_1) and (n_2, r_2) are enabled by the same reachable marking \mathbf{x} and $P_{n_1} \cap P_{n_2} = \emptyset$ then the outcomes can occur concurrently. The condition $P_{n_1} \cap P_{n_2} = \emptyset$ is also necessary for concurrent occurrences of outcomes because, in our model, a single agent cannot be engaged concurrently in two different atoms, and because two state transformers cannot operate concurrently on the local state of an agent. Thus concurrency between outcomes depends only on the involved atoms (and their parties) and not on the results.

Concurrency is formally captured by the *concurrent step reachability graph*, defined next. A *concurrent step* enabled at a reachable marking \mathbf{x} is a nonempty set of pairwise concurrent outcomes, each of them enabled by \mathbf{x} . It is immediate to see that all the outcomes of a concurrent step can be executed subsequently in arbitrary order and that the marking finally reached does not depend on the chosen order. We call this marking *reached by the concurrent step*.

Definition 4. *The concurrent step reachability graph of a negotiation \mathcal{N} has all markings reachable from \mathbf{x}_0 as vertices. An arc, annotated by a nonempty set of outcomes, leads from \mathbf{x} to \mathbf{x}' whenever the outcomes of this set are pairwise concurrent and the concurrent step leads from \mathbf{x} to \mathbf{x}' . Again, \mathbf{x}_0 is the distinguished initial vertex.*

3 From Negotiations to Petri Nets

We assume that the reader is acquainted with (low-level) initially marked Petri nets, the occurrence rule, reachable markings, liveness, and the graphical representation of nets as directed graphs. For each place, there are directed arcs from all *input transitions* to the place and directed arcs from the place to all *output*

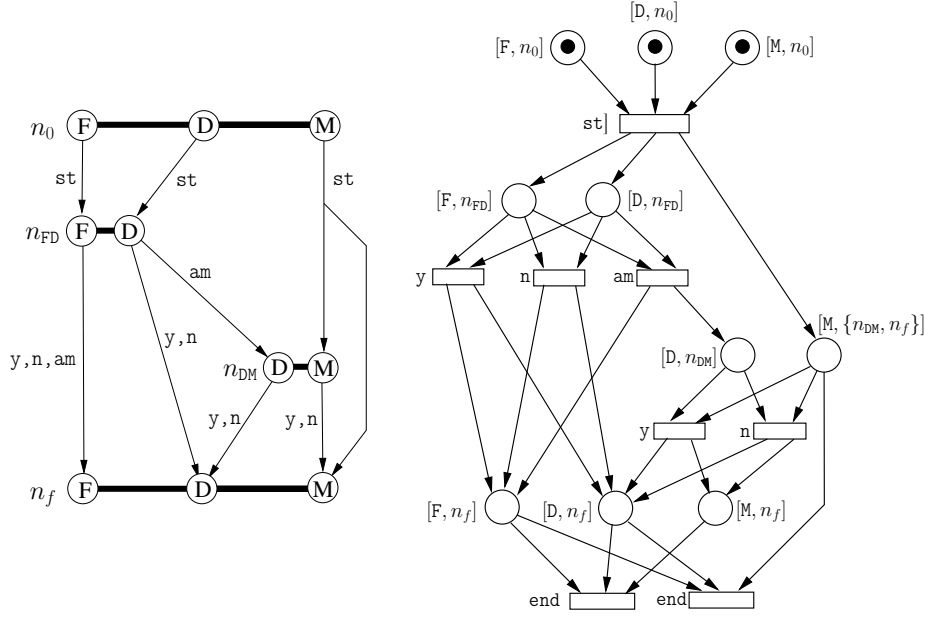


Fig. 2. Petri net semantics of the negotiation of Figure 1

transitions. Input places and output places of transitions are defined analogously. A labelled Petri net is a Petri net with a labelling function λ , mapping transitions to some set of labels. Graphically, the label $\lambda(t)$ of a transition t is depicted as an annotation of t .

3.1 Examples

The semantics of negotiations uses many notions from Petri net theory. In this section, we provide a translation and begin with an example.

Figure 2 shows on the right the net for the negotiation shown on the left (which was also shown in Figure 1). Since the number of places of the net equals the number of ports of the negotiation, one might assume that the relation between ports and places is a simple one-to-one mapping. Moreover, the transitions of the net have an obvious relation to the outcomes, i.e., to the results of the negotiation atoms (if the two **end**-transitions are ignored).

Now we have a look at the two **end**-transitions of the Petri net. The left transition refers to the last result of the negotiation's occurrence sequence $(n_0, st), (n_{FD}, am), (n_{DM}, y), (n_f, end)$, where **end** is a result of n_f . The right transition refers to the last result of the occurrence sequence $(n_0, st), (n_{FD}, y), (n_f, end)$. Hence, roughly speaking, the left transition refers to the left branch of the (only) proper hyperarc of the graphical representation of the negotiation, and the right transition refers to the right branch.

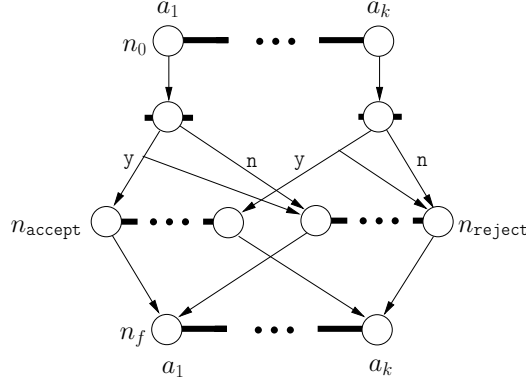


Fig. 3. A (not yet completely correct) negotiation for unanimous vote (all agents participate in all atoms)

For a negotiation with more than one proper hyperarc, each occurrence sequence can involve a particular branching of a hyperarc (moreover, an atom can occur more than once, leading to different branches of the same hyperarc). For k hyperarcs with binary branching, this results in 2^k possible patterns. As can be seen in the following example, this can result in exponentially many transitions of the associated Petri net.

Figure 3 shows a class of negotiations with parameter k , involving agents a_1, \dots, a_k . These negotiations represent a distributed voting process. Each agent votes with possible outcome **yes** or **no** (one-party-negotiations). For each **yes**-outcome there are two possible next atoms, n_{accept} and n_{reject} , whereas for each **no**-outcome n_{reject} is the only possibility. So the atom n_{accept} is only enabled if all agents vote **yes**, while the atom n_{reject} is always enabled when all agents have voted.

A Petri net representing this behaviour necessarily has to distinguish the k possible **yes**-outcomes and **no**-outcomes, because final acceptance is only possible if all agents have accepted. So we need $2 \cdot k$ corresponding places, k for acceptance and k for rejection. When all agents came to a result, one of 2^k possible markings is reached. Only for one of these markings (all agents accepted), final acceptance is possible, and this will be represented by one transition. For each of the $2^k - 1$ alternative constellations, we need a separate transition to remove the tokens and come to final rejection. So we end up with 2^k transitions.

3.2 Formal Translation of Negotiations

We associate with a negotiation $\mathcal{N} = (N, n_0, n_f, \mathcal{X})$ a (labelled) Petri net. The places of this net are, for each atom n except n_f , the pairs $[a, S]$ such that $a \in P_n$, $r \in R_n$, and $\mathcal{X}(n, a, r) = S$, plus, for each $a \in A$, the pair $[a, \{n_0\}]$. Observe that the number of places is linear in the size of \mathcal{N} (which might exceed $|N|$ significantly, because, for each n in N , for each a in P_n and for each result

$r \in R_n$ we have a set of possible successor negotiations in \mathcal{X}). In the sequel (and in the figures) we write $[a, n]$ instead of $[a, \{n\}]$. The *initial marking* assigns one token to each place $[a, \{n_0\}]$ and no token to all other places.

The net has a set of transitions $T(n, r)$ for each outcome (n, r) . An input place of a transition in $T(n, r)$ reflects that a party of negotiation n is actually ready to engage in n (and possibly in other atoms as well). For a single agent, there might be more than one such place, resulting in several transitions. Each transition in $T(n, r)$ has input places referring to all involved parties, which results in a transition for each combination of respective input places.

Formally, let $P_n = \{a_1, \dots, a_k\}$. $T(n, r)$ contains a transition $[n, r, L]$ for every tuple $L = ([a_1, S_1], \dots, [a_k, S_k])$ such that $n \in S_1 \cap \dots \cap S_k$. The set of input places of $[n, r, L]$ is $\{[a_1, S_1], \dots, [a_k, S_k]\}$, and its set of output places is $\{[a_1, \mathcal{X}(n, a_1, r)], \dots, [a_k, \mathcal{X}(n, a_k, r)]\}$. All transitions of the set $T(n, r)$ are labelled by the outcome (n, r) . They all have the same output places. Moreover, they have the same number of input and output places, both of them equal to the number of parties of n .

For the negotiation on the left of Figure 2, we get seven sets of transitions: $T(n_0, \text{st})$, $T(n_{\text{FD}}, \text{y})$, $T(n_{\text{FD}}, \text{n})$, $T(n_{\text{FD}}, \text{am})$, $T(n_{\text{DM}}, \text{y})$, $T(n_{\text{DM}}, \text{n})$, and $T(n_f, \text{end})$. All of them are singletons, with the exception of $T(n_f, \text{end})$, which contains the two transitions shown at the bottom of the figure. In the figure, we annotate transitions only by results r instead of outcomes (n, r) . Notice that here we assume a unique result **end** of n_f .

Proposition 1. *For each atom $n \neq n_f$, each transition labelled by (n, r) has exactly one input place $[a, X]$ for each agent $a \in P_n$, and exactly one output place $[a, Y]$ for each agent $a \in P_n$. Transitions labelled by (n_f, end) have no output places.* \square

Corollary 1. *For each agent a , the number of tokens on places $[a, X]$ never increases. Since this number is one initially, it is at most one for each reachable marking.* \square

Corollary 2. *The net associated with a negotiation is 1-safe, i.e., no reachable marking assigns more than one token to a place.* \square

Lemma 1. *The net associated with a negotiation is deterministic, i.e., no reachable marking enables two distinct transitions with the same label.*

Proof. A transition labelled by (n, r) has an input place for each participant of n . Two equally labelled transitions cannot have identical sets of input places by construction. Hence, for at least one agent a there is a place $[a, X]$ which is input place of one of the transitions and a distinct place $[a, Y]$ which is input place of the other transition. Since, by Corollary 1, each reachable marking marks at most one of these two places, each reachable marking enables at most one of the transitions. \square

The (sequential) behaviour of a labelled Petri net is represented by its reachability graph:

Definition 5. *The reachability graph of a Petri net has all reachable markings m as vertices, an arc annotated by t leading from m to m' when m enables transition t and the occurrence of t leads to m' , and a distinguished initial marking m_0 . The label reachability graph of a labelled Petri net is obtained from its reachability graph by replacing each transition by its label.*

In terms of reachability graphs, a labelled Petri net is deterministic if and only if its label reachability graph has no vertex with two outgoing edges which carry the same label. An occurrence sequence of a deterministic labelled Petri net is fully determined by the sequence of transition labels, as shown in the following proposition, and so is the sequence of markings reached.

For a labelling function λ and an occurrence sequence $\sigma = t_1 t_2 t_3 \dots$, we write $\lambda(\sigma)$ for the sequence of labels $\lambda(t_1) \lambda(t_2) \lambda(t_3) \dots$ in the sequel.

Proposition 2. *Let σ_1 and σ_2 be two finite, initially enabled occurrence sequences of a deterministic labelled Petri net with labelling function λ . Let m_1 be the marking reached by σ_1 , and let m_2 be the marking reached by σ_2 . If $\lambda(\sigma_1) = \lambda(\sigma_2)$ then $m_1 = m_2$. \square*

3.3 Behavioural Equivalence between Negotiations and Nets

In this subsection, we will employ the usual notion of isomorphism between reachability graphs:

Definition 6. *Two reachability graphs are isomorphic if there exists a bijective mapping φ between their sets of vertices, mapping the initial vertex of the first graph to the initial vertex of the second graph, such that there is an edge from u to v labelled by some t in the first graph if and only if there is an edge from $\lambda(u)$ to $\lambda(v)$ labelled by t in the second graph.*

Reachability graph isomorphism is a very strong behavioral equivalence notion for sequential behaviour. If moreover the concurrent step reachability graphs of two models are isomorphic, then also the concurrent behaviour of the systems coincide. We will show the existence of both isomorphisms between negotiations and associated Petri nets.

Proposition 3. *The reachability graph of a negotiation and the label reachability graph of the associated labelled Petri net are isomorphic.*

Proof. (Sketch). We interpret a token on a place $[a, \{n_1, \dots, n_k\}]$ on the negotiation side as “agent a is ready to engage in the atoms of the set $\{n_1, \dots, n_k\}$ ”. It is immediate to see that this holds initially. By construction of the Petri net, a small step (n, r) of the negotiation is mimicked by an occurrence of a transition of the set $T(n, r)$, and hence by a transition labelled by (n, r) . By construction, the marking of the negotiation reached by the occurrence of the outcome corresponds to the marking of the net reached by the occurrence of the transition. \square

For comparing the concurrent behaviour of negotiations and associated labelled Petri nets, we have to define concurrent enabledness of transitions. This is easy in our setting, because the considered nets are 1-safe.

Definition 7. *Two transitions t and t' of a 1-safe Petri net are concurrently enabled at a reachable marking m if m enables both t and t' and if moreover t and t' have no common input place.*

Concurrent behaviour is captured by the concurrent step reachability graph and, for labelled Petri nets, by its label version. In the following definition, a set of transitions is said to be *concurrently enabled* if any two distinct transitions in this set are concurrently enabled.

Definition 8. *The concurrent step reachability graph of a Petri net has all reachable markings m as vertices, a distinguished initial marking m_0 and an arc labelled by U leading from m to m' when m concurrently enables a nonempty set U of transition and the occurrence of all transitions of U (in any order) leads from m to m' .*

The label concurrent step reachability graph of a labelled Petri net is obtained from its concurrent step reachability graph by replacing each set of transitions by the multiset of its labels.

Fortunately, in our setting two equally labelled transitions are never enabled concurrently, so that the labels of concurrent steps will never be proper multisets, but just sets.

Lemma 2. *If two outcomes (n, r) and (n', r') of a negotiation are concurrently enabled at a marking reached by an initial occurrence sequence σ , then there is an initially enabled occurrence sequence μ of the associated labelled Petri net such that $\lambda(\mu) = \sigma$ and the marking reached by μ concurrently enables two transitions labelled by (n, r) and (n', r') respectively.*

Conversely, if a marking of the (λ) -labelled Petri net reached by an occurrence sequence μ concurrently enables two transitions t and t' , then the marking of the negotiation reached by $\lambda(\mu)$ concurrently enables the two outcomes $\lambda(t)$ and $\lambda(t')$.

Proof. (Sketch). By construction of the Petri net, a transition t has an input place $[a, X]$ only if $\lambda(t) = (n, r)$ for an agent $a \in P_n$. Assume that two enabled transitions are not concurrent. Then they share an input place $[a, X]$ only if their labels refer to two outcomes (n, r) and (n', r') such that $a \in P_n$ and $a \in P_{n'}$. So $P_n \cap P_{n'} \neq \emptyset$, and thus the two outcomes are not concurrent.

Conversely, if two outcomes (n, r) and (n', r') are enabled but not concurrent, then some agent a belongs to both P_n and $P_{n'}$. In the Petri net, each transition labelled by (n, r) or by (n', r') has an input place $[a, X]$. Since each reachable marking marks only one place $[a, X]$ by Corollary 1, two distinct enabled transitions labelled by (n, r) or by (n', r') share this marked input place, whence they are not concurrent. \square

Corollary 3. *The concurrent step reachability graph of a negotiation and the label concurrent step reachability graph of its associated Petri net are isomorphic.* \square

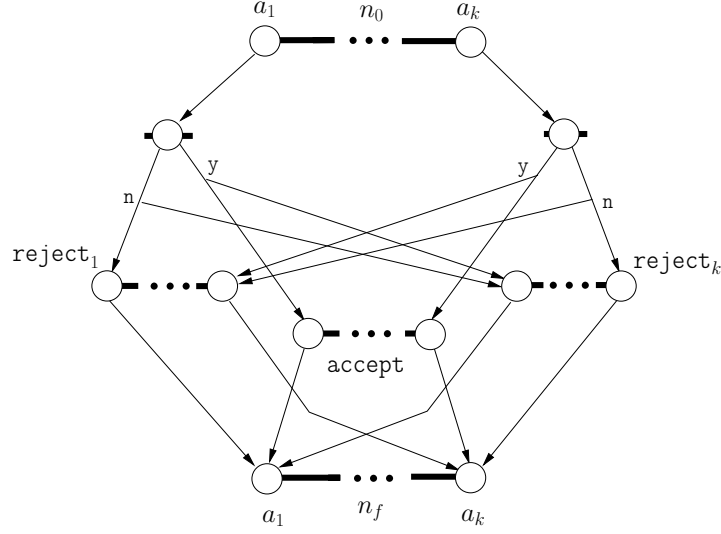


Fig. 4. A corrected negotiation for unanimous vote (all agents participate in all atoms)

3.4 Excursion: On the Voting Example

The reader possibly finds unsatisfactory that the negotiation given in Figure 3 can reject even when all parties vote yes. This results in 2^k respective transitions of the Petri net. If we want to avoid this possibility in the Petri net, we just remove the single transition that removes tokens from all **accept**-places and enables overall rejection. For the negotiation, we found the following work-around: we replace the atom n_{reject} by k rejecting atoms **reject_i**, for $1 \leq i \leq k$. If agent a_i votes yes, then it is ready to engage in **accept** and in all **reject_j** such that $j \neq i$. Any of the **reject_j**-atoms have a single result that leads to final rejection. When all agents vote **yes** then none of the **reject_i**-atoms are enabled, whence only overall acceptance can take place. Notice that this construction is a bit clumsy (see Figure 4), but still does not require exponentially many elements, as the associated Petri net does.

4 Properties of the Net Associated with a Negotiation

4.1 S-components

An *S-component* of a Petri net is a subnet such that, for each place of the subnet, all input- and output-transitions belong to the subnet as well, and such that each transition of the subnet has exactly one input- and exactly one output-place of the subnet. It is immediate to see that the number of tokens in an S-component never changes. A net is covered by S-components if each place and each transition belongs to an S-component. Nets covered by S-components carrying exactly one

token are necessarily 1-safe. For example, every live and 1-safe free-choice net enjoys this nice property [3].

Petri nets associated with negotiations are not covered by S-components, only because the **end**-transitions have no output places. However, if we add an arc from each **end**-transition to each initially marked place, then the resulting net is covered by S-components:

Proposition 4. *The Petri net associated with a negotiation, with additional arcs from each **end**-transition to each initially marked place, is covered by S-components.*

Proof. (Sketch). For each agent a , the subnet generated by all places $[a, X]$ and all transitions labelled by (n, r) , where $a \in P_n$, is an S-component (being generated implies that the arcs of the subnet are all arcs of the original net connecting nodes of the subnet). An arbitrary place of the net belongs to one such subnet, because it corresponds to an agent. Each transition has a label (n, r) , and each atom n has a nonempty set of participants. \square

4.2 Soundness

The following notion of sound negotiations was inspired by van der Aalst's soundness of workflow nets [1].

Definition 9. *A negotiation is sound if each outcome occurs in some initial occurrence sequence and if, moreover, each finite occurrence sequence is a large step or can be extended to a large step.*

All the negotiations shown in the figures of this paper are sound. For an example of an unsound negotiation, consider again the ping-pong negotiation shown in Figure 1 on the right hand side. Imagine that Daughter could choose to start negotiating with Father or with Mother. This could be expressed by replacing the arc from port D of n_0 to port D of n_{FD} by a hyperarc from port D of n_0 to ports D of both n_{FD} and n_{DM} . If the first negotiation is between Daughter and Mother, and if it is successful, a marking is reached where both Daughter and Mother can only engage in the final atom n_f , whereas father is still only able to participate in n_{FD} . So the distributed negotiation has reached a marking which is neither final nor enables any outcome. We call such a marking a *deadlock*. Clearly, sound negotiations have no reachable deadlocks.

Since the Petri nets associated with negotiations are not workflow nets, we cannot immediately compare the soundness notions of workflow nets and of negotiations. Instead, we first provide a translation of nets associated with negotiations to workflow nets. It turns out that a sound negotiation does not necessarily lead to a sound workflow net in the general case. However, for the subclass of deterministic negotiations the two concepts coincide, as will be shown next.

We begin with a very simple equivalence transformation of nets:

Definition 10. Two Petri nets N and N' are in the relation \mathcal{R} if

- either N has two distinct places with identical sets of input transitions, identical sets of output transitions and equal initial markings, and N' is obtained from N by deletion of one of these places (and adjacent arcs),
- or N has a place without output transition, and N' is obtained from N by deletion of this place.

The symmetrical, reflexive and transitive closure of \mathcal{R} is called place equivalence.

Obviously, two place-equivalent nets have identical behaviour, i.e., their reachability graphs are isomorphic and so are their concurrent step reachability graphs. Notice, however, that place-equivalence does not respect 1-safety. If the only place that violates 1-safety has no output-transition, then deletion of this place can make a net 1-safe.

A *workflow net* is a Petri net with two distinguished places p_{in} and p_{out} such that p_{in} has no input transition, p_{out} has no output transition and, for each place or transition x , there are directed paths from p_{in} to x and from x to p_{out} . The initial marking of a workflow net assigns one token to the place p_{in} and no token to all other places. Workflow nets also have a *final marking*, assigning only one token to p_{out} . A workflow net is *sound* if it has no dead transitions (i.e., each transition is in an initially enabled occurrence sequence) and, moreover, each initially enabled occurrence sequence is a prefix of an occurrence sequence leading to the final marking.

Proposition 5. The net associated with a sound negotiation is place-equivalent to a workflow net.

Proof. (Sketch). We derive a single input place p_{in} by deleting all but one of the initially marked places. We add a new place p_{out} with all **end**-transitions as input transitions. Both transformations apparently lead to place equivalence nets.

Since, by soundness of the negotiation, every atom (and therefore every outcome) can be enabled, a token can be moved from the initial atom to any other atom, and therefore there is a directed path from the initial atom to any other atom (more precisely, there is a path in the graph of the negotiation). By the construction of the Petri net, there are according paths from the place p_{in} to arbitrary places and transitions of the net.

Again by soundness of the negotiation, every occurrence sequence can be extended to a large step, i.e., the final atom can eventually be enabled and the final marking reached. So every “token” can be led to the final atom, and therefore there are paths in the graph of the negotiation from every atom to the final atom. By construction of the Petri net, there are thus paths from any element to an **end**-transition, and finally to the new place p_{out} . \square

Unfortunately, soundness of a negotiation does not necessarily imply soundness of a related workflow net. The reason is that soundness requires that every atom can occur but not that every branch of a hyperarc is actually used. If,

for example, there would be an additional hyperarc in Figure 1 from the port F in n_0 to the ports F in n_{FD} and n_f instead of the arc from n_0 to n_{FD} , then the resulting negotiation would still be sound (actually, the behaviour does not change at all). In the associated Petri net, however, there would be an additional transition **end** with new input place $[F, \{n_{FD}, n_f\}]$ (and other input places) which never is enabled. This net is therefore not sound.

4.3 Deterministic Negotiations

In [5], we concentrate on *deterministic negotiations* which are negotiations without proper hyperarcs.

Definition 11. *A negotiation is deterministic if, for each atom n , agent $a \in P_n$ and result $r \in R_n$, $\mathcal{X}(n, a, r)$ contains at most one atom (and no atom only if $n = n_f$).*

The term deterministic is justified because there is no choice for an agent with respect to the next possible atom.

Since both, the exponential blow-up and the problem of useless arcs (branches of hyperarcs) stem from proper hyperarcs, we can expect that deterministic negotiations allow for better results. Actually, the Petri net associated with a deterministic negotiation is in fact much smaller, because all its places have the form $[a, X]$, where a is an agent and X is a singleton set of atoms. So the set of places is linear in agents and in atom.

Before discussing soundness of deterministic negotiations, we make a structural observation:

Proposition 6. *The net associated with a deterministic negotiation is a free-choice net, i.e., every two places either share no output transitions, or they share all their output transitions.*

Proof. (Sketch). Since, in nets associated with deterministic negotiations, each place has the form $[a, X]$, where X is a singleton set $\{n\}$, all its output transitions are labelled by (n, r) , r being a possible result of n . By construction, every other place $[b, \{n\}]$ has exactly the same output transitions as $[a, \{n\}]$ whereas all other places have no common output transition with $[a, \{n\}]$. \square

Proposition 7. *The net associated with a deterministic negotiation is sound if and only if it is place equivalent to a sound workflow net.*

Proof. (Sketch). Observe that the translation from the negotiation to the associated Petri net is much easier in this case: for each atom n we add places $[a, n]$ for each a in P_n and transitions (n, r) for each $r \in R_n$. There are no two transitions for any outcome (n, r) , and so transition labels are not necessary (formally, we can label each transition by itself). For each such place $[a, n]$ of an atom n and each such transition (n, r) we add an arc from $[a, n]$ to (n, r) . Finally we add arcs from transitions (n, r) to places $[b, n']$ whenever $\mathcal{X}(n, a, r) = \{n'\}$ and $b \in P_{n'}$.

It is immediate that to see this net is free-choice and that the behaviour of the negotiation is precisely mimicked by the net. So the negotiation is sound if and only if the net has no dead transitions and moreover can always reach the final (empty) marking.

The result follows since the net can, as above, be translated into a place equivalent workflow net. \square

5 From Nets to Negotiations

In this section we study the converse direction: Given a labelled Petri net, is there a negotiation such that the net is associated with the negotiation? Obviously, for a positive answer the net has to enjoy all the properties derived before. In particular, it must have disjoint S-components and initially marked input places. However, in the general case it appears to be difficult to characterise nets that have corresponding negotiations.

We will provide an answer for the case of sound deterministic negotiations and sound free-choice workflow nets.

Proposition 8. *Every sound free-choice workflow net is place equivalent to a net which is associated with a sound deterministic negotiation.*

Proof. (Sketch). A workflow net is sound if and only if the net with an additional feedback transition moving the token from p_{out} back to p_{in} is live and 1-safe [1]. Live and 1-safe workflow nets are covered by S-components [3]. Therefore a sound workflow net is covered by S-components as well. However, these S-components have not necessarily disjoint sets of places. Consequently, we cannot easily find candidates for agents involved in the negotiation to be constructed.

Instead we proceed as follows: We choose a minimal set of S-components that cover the net. Since each S-component of a live net has to carry a token, all these S-components contain the place p_{in} . Each S-component will be an agent of the net to be constructed, and each conflict cluster (i.e., each maximal set of places together with their common output transitions) a negotiation atom.

Each place p of the net is contained in at least one S-component of the cover. Let C_p be the set of all S-components of the derived minimal cover containing p . If C_p contains more than one S-component, we duplicate the place p , getting a new place p' with input and output transitions like p . Now the new net still has a cover by S-components, where one of the S-components containing p now contains p' instead. Repetition of this procedure eventually leads to a net where each place p belongs to exactly one S-component C_p of the cover. Finally we delete the place p_{out} . Both operations, duplication of places and deletion of p_{out} , lead to place-equivalent nets.

The resulting net is associated with the following negotiation: The set of agents is the set of S-components of the minimal cover. The atoms are the conflict clusters of the net. The results of an atom are the transitions of the corresponding conflict cluster. The \mathcal{X} -function can be derived from the arcs of the Petri net leading from transitions to places. \square

6 Conclusions

This contribution presented the translation from distributed negotiations to Petri nets such that a negotiation and its associated Petri nets are behaviourally equivalent in a strong sense. In the general case, the Petri net is exponentially larger than the negotiation, whereas for deterministic negotiations its size is only linear. Petri nets do not inherit many properties from arbitrary negotiations, but for deterministic negotiations soundness and non-soundness is respected by the transformation to workflow-like Petri nets, whence in this case the reverse translation is possible as well.

Analysis results for negotiations might be transferable to Petri nets and vice versa via the translation. In future work, we will study this question in particular for the respective sound and complete sets of reduction rules for negotiations [4, 5] and for free-choice Petri nets [3].

Based on the reduction results, a very recent work [6] introduces a global specification language for negotiations and characterises the negotiations expressible with this language. Similar results for Petri nets could be derived via the translation procedure of this paper.

References

1. van der Aalst, W.M.P.: The application of Petri nets to workflow management. *J. Circuits, Syst. and Comput.* 08(01), 21–66 (1998)
2. Chen, Y., Peng, Y., Finin, T., Labrou, Y., Chu, B., Yao, J., Sun, R., Willhelm, B., Cost, S.: A negotiation-based multi-agent system for supply chain management. In: *In Proceedings of Agents 99 - Workshop on Agent Based Decision-Support for Managing the Internet-Enabled Supply-Chain.* pp. 15–20 (1999)
3. Desel, J., Esparza, J.: *Free Choice Petri Nets.* Cambridge University Press, New York, NY, USA (1995)
4. Esparza, J., Desel, J.: On negotiation as concurrency primitive. In: D’Argenio, P.R., Melgratti, H.C. (eds.) *CONCUR. Lecture Notes in Computer Science*, vol. 8052, pp. 440–454. Springer (2013), extended version in *arXiv:1307.2145*, <http://arxiv.org/abs/1403.4958>
5. Esparza, J., Desel, J.: On negotiation as concurrency primitive II: Deterministic cyclic negotiations. In: Muscholl, A. (ed.) *FoSSaCS. Lecture Notes in Computer Science*, vol. 8412, pp. 258–273. Springer (2014), extended version in *CoRR* [abs/1403.4958](http://arxiv.org/abs/1403.4958), <http://arxiv.org/abs/1403.4958>
6. Esparza, J., Desel, J.: Negotiation programs. In: Devillers, R., Valmari, A. (eds.) *Petri Nets. Lecture Notes in Computer Science*, vol. 9115, pp. 157–178. Springer (2015)
7. Simon, C.: *Negotiation Processes – The Semantic Process Language and Applications.* Shaker, Aachen, Germany (2008)
8. Xu, H., Shatz, S.M.: An agent-based Petri net model with application to seller/buyer design in electronic commerce. In: *Fifth International Symposium on Autonomous Decentralized Systems, ISADS 2001, Dallas, Texas, USA, March 26-28, 2001.* pp. 11–18. IEEE Computer Society (2001)

Non-Interference Notions Based on Reveals and Excludes Relations for Petri Nets

Luca Bernardinello, Görkem Kılınç, and Lucia Pomello

Dipartimento di informatica, sistemistica e comunicazione
Università degli Studi di Milano-Bicocca, Italy

Abstract. In distributed systems, it is often important that a user is not able to infer if a given action has been performed by another component, while still being able to interact with that component. This kind of problems has been studied with the help of a notion of “interference” in formal models of concurrent systems (e.g. CCS, Petri nets). Here, we propose several new notions of interference for ordinary Petri nets, study some of their properties, and compare them with notions already proposed in the literature. Our new notions rely on the unfolding of Petri nets, and on an adaptation of the “reveals” relation for ordinary Petri nets, previously defined on occurrence nets, and on a new relation, called “excludes”, here introduced for detecting negative information flow.

Keywords: information flow, non-interference, reveals, excludes, Petri nets, unfolding.

1 Introduction

In distributed systems, information flows among components. The flow can be used to rule the behavior of the system, to guarantee the correct synchronization of tasks, to implement a communication protocol, and so on.

In some cases, a flow of information from one component to another is actually a *leakage*: that piece of information should not have passed from here to there. Such unwanted flows can endanger the working of the system.

In this paper, we study formal notions of unwanted information flow, based on a general notion of *non-interference*, within the theory of Petri nets, and compare our approach with existing approaches.

Non-interference was first defined for deterministic programs [1]. Later, several adaptations were proposed for more abstract settings, like transition systems, usually related to observational semantics [2–6].

Broadly speaking, these approaches assume that the actions performed in a system belong to two types, conventionally called *high* (hidden) and *low* (observable). A system is then said to be free from interference if a user, by interacting only via low actions, cannot deduce information about which high actions have been performed.

This approach was formalized in terms of 1-safe Petri nets in [7], relying on known observational equivalences, including bisimulation. Similarly to Busi and Gorrieri [7], in this paper we analyze systems that can perform high and low level actions and we check if an observer, who knows the structure of the system, can deduce information about the high actions by observing low actions. We rely on a progress assumption which was ignored in non-interference notions in the literature.

We propose new notions of non-interference for ordinary Petri nets. They deal with positive information flow as well as negative information flow, regarding both past and future occurrences and are based on unfoldings and on reveals and excludes relations which are formally defined in Section 3. *Reveals* was originally defined as a relation between events of an occurrence net in [8] and applied in fault diagnosis. Here, we adapt this relation to transitions of Petri nets. Intuitively, a transition t_1 reveals another transition t_2 if, by observing the occurrence of t_1 , it is possible to deduce the occurrence of t_2 . *Excludes* is a new relation between transitions of a Petri net, which is introduced in order to detect negative information flow. A transition t_1 excludes another transition t_2 if, by observing the occurrence of t_1 , it is possible to deduce that t_2 has not yet occurred and will not occur in the future, i.e., they never appear together in the same run.

The first notion of non-interference we introduce is called *Reveals based Non-Interference (RNI)* and it states that a net is secure if no low transition reveals any high transition. This new notion is introduced in Section 4.1. We also propose more restrictive notions called *k-Extended-Reveals based Non-Interference (k-ERNI)* and *n-Repeated-Reveals based Non-Interference (n-ReRNI)*, they are based on observation of multiple occurrences of low transitions. These two parametric non-interference notions are introduced and discussed in Section 4.2 and Section 4.3. In Section 4.4, *Positive/Negative Non-Interference (PNNI)* is introduced on the basis of both the reveals and excludes relations between low and high transitions capturing both positive and negative information flow. The new notions are discussed and compared with each other while they are introduced. In Section 5, we compare, on the basis of examples, the new introduced notions with the ones already introduced in the literature and mentioned at the beginning of Section 4. Finally, Section 6 concludes the paper and discusses some possible developments.

2 Basic Definitions

In this section we collect preliminary definitions and set the notation which will be used in the rest of the paper.

Let $R \subseteq I \times I$ be a binary relation, the transitive closure of R is denoted by R^+ ; the reflexive and transitive closure of R is denoted by R^* .

A *net* is a triple $N = (B, E, F)$, where B and E are disjoint sets, and $F \subseteq (B \times E) \cup (E \times B)$ is called the *flow relation*. The pre-set of an element $x \in B \cup E$

is the set $\bullet x = \{y \in B \cup E \mid (y, x) \in F\}$. The post-set of x is the set $x^\bullet = \{y \in B \cup E \mid (x, y) \in F\}$.

An (ordinary) Petri net $N = (P, T, F, m_0)$ is defined by a net (P, T, F) , and an initial marking $m_0 : P \rightarrow \mathbb{N}$. The elements of P are called *places*, the elements of T are called *transitions*. A net is finite if the sets of places and of transitions are finite.

A *marking* is a map $m : P \rightarrow \mathbb{N}$. A marking m is safe if $m(p) \in \{0, 1\}$ for all $p \in P$. Markings represent global states of a net.

A transition t is *enabled* at a marking m if, for each $p \in \bullet t$, $m(p) > 0$. We write $m[t\rangle$ when t is enabled at m . A transition enabled at a marking can *fire*, producing a new marking. Let t be enabled at m ; then, the firing of t in m produces the new marking m' , defined as follows:

$$m'(p) = \begin{cases} m(p) - 1 & \text{for all } p \in \bullet t \setminus t^\bullet \\ m(p) + 1 & \text{for all } p \in t^\bullet \setminus \bullet t \\ m(p) & \text{in all other cases} \end{cases}$$

We will write $m[t\rangle m'$ to mean that t is enabled at m , and that firing t in m produces m' .

A marking q is *reachable* from a marking m if there exist transitions $t_1 \dots t_{k+1}$ and intermediate markings $m_1 \dots m_k$ such that

$$m[t_1\rangle m_1[t_2\rangle m_2 \dots m_k[t_{k+1}\rangle q$$

The set of markings reachable from m will be denoted by $[m\rangle$. If all the markings reachable from m_0 are safe, then $N = (P, T, F, m_0)$ is said to be 1-safe (or, shortly, safe).

Let $N = (B, E, F)$ be a net, and $x, y \in B \cup E$. If there exist $e_1, e_2 \in E$, such that $e_1 \neq e_2$, $e_1 F^* x$, $e_2 F^* y$, and there is $b \in \bullet e_1 \cap \bullet e_2$, then we write $x \# y$.

A net $N = (B, E, F)$ is an *occurrence net* if the following restrictions hold:

1. $\forall x \in B \cup E : \neg(x F^+ x)$
2. $\forall x \in B \cup E : \neg(x \# x)$
3. $\forall e \in E : \{x \in B \cup E \mid x F^* e\}$ is finite
4. $\forall b \in B : |\bullet b| \leq 1$

The set of minimal elements of an occurrence net N with respect to F^* will be denoted by ${}^\circ N$. The elements of B are called *conditions* and the elements of E are called *events*. If $x \# y$ in an occurrence net, then we say that x and y are in conflict. Let $e \in E$ be an event in an occurrence net; then the *past* of e is the set of events preceding e in the partial order given by F^* : $\uparrow e = \{t \in E \mid t F^* e\}$. An occurrence net represents the alternative histories of a process; therefore its underlying graph is acyclic, and paths branching from a condition, corresponding to a choice between alternative behaviors, never converge.

A *run* of an occurrence net $N = (B, E, F)$ is a set R of events which is closed with respect to the past, and free of conflicts: (1) for each $e \in R$, $\uparrow e \subseteq R$; (2)

for each $e_1, e_2 \in R$, $\neg(e_1 \# e_2)$. A run is maximal if it is maximal with respect to set inclusion.

Let $N_i = (P_i, T_i, F_i)$ be a net for $i = 1, 2$. A map $\pi : P_1 \cup T_1 \rightarrow P_2 \cup T_2$ is a morphism from N_1 to N_2 if:

1. $\pi(P_1) \subseteq P_2$; $\pi(T_1) \subseteq T_2$
2. $\forall t \in T_1$ the restriction of π to $\bullet t$ is a bijection from $\bullet t$ to $\bullet \pi(t)$
3. $\forall t \in T_1$ the restriction of π to t^\bullet is a bijection from t^\bullet to $\pi(t)^\bullet$

In the rest of the paper, we will consider finite Petri nets, i.e., Petri nets whose underlying net is finite, except for occurrence nets. Of course, Petri nets may have infinite behavior. Moreover, we assume that all transitions of a Petri net have non-empty preset, i.e., all have input places.

A *branching process* of a Petri net $N = (P, T, F, m_0)$ is a pair (O, π) , where $O = (B, E, G)$ is an occurrence net, and π is a morphism from O to N such that:

1. $\forall p \in P \ m_0(p) = |\pi^{-1}(p) \cap {}^\circ O|$
2. $\forall x, y \in E$, if $\bullet x = \bullet y$ and $\pi(x) = \pi(y)$, then $x = y$

A branching process $\Pi_1 = (O_1, \pi_1)$ is a prefix of $\Pi_2 = (O_2, \pi_2)$ if there is an injective morphism f from O_1 to O_2 which is a bijection when restricted to ${}^\circ O_1$, and such that $\pi_1 = \pi_2 f$.

Any finite Petri net N has a unique branching process which is maximal with respect to the prefix relation. This maximal process, called the *unfolding* of N , will be denoted by $\text{Unf}(N) = ((B, E, F), \lambda)$, where λ is the morphism from (B, E, F) to N [9]. In Fig. 1, a Petri net with its infinite unfolding is illustrated.

The following definition will be used in the rest of the paper to denote the set of events of an unfolding corresponding to a specific transition of a given Petri net.

Definition 1. Let $N = (P, T, F, m_0)$ be a Petri net, $\text{Unf}(N) = ((B, E, F), \lambda)$ be its unfolding and $t \in T$, the set of events corresponding to t is denoted $E_t = \{e \in E \mid \lambda(e) = t\}$.

The following definitions concern the *reveals* relation, originally introduced in [8] and applied to diagnostics problems. This notion has been further studied in [10] and [11].

Definition 2. Let $O = (B, E, F)$ be an occurrence net, $\Omega \subseteq 2^E$ be the set of its maximal runs, and e_1, e_2 be two of its events. Event e_1 reveals e_2 , denoted $e_1 \triangleright e_2$, iff $\forall \sigma \in \Omega, e_1 \in \sigma \implies e_2 \in \sigma$

Definition 3. [10] Let $O = (B, E, F)$ be an occurrence net, $\Omega \subseteq 2^E$ be the set of its maximal runs, and A, B two sets of events. A extended-reveals B , $A \rightarrow B$, iff $\forall \omega \in \Omega, A \subseteq \omega \implies B \cap \omega \neq \emptyset$.

In other words, a set of events, A , extended-reveals another set of events, B , written $A \rightarrow B$, iff every maximal run that contains A also hits B . The reveals relation can be expressed as extended-reveals relation between singletons: $a \triangleright b$ can be written as $\{a\} \rightarrow \{b\}$.

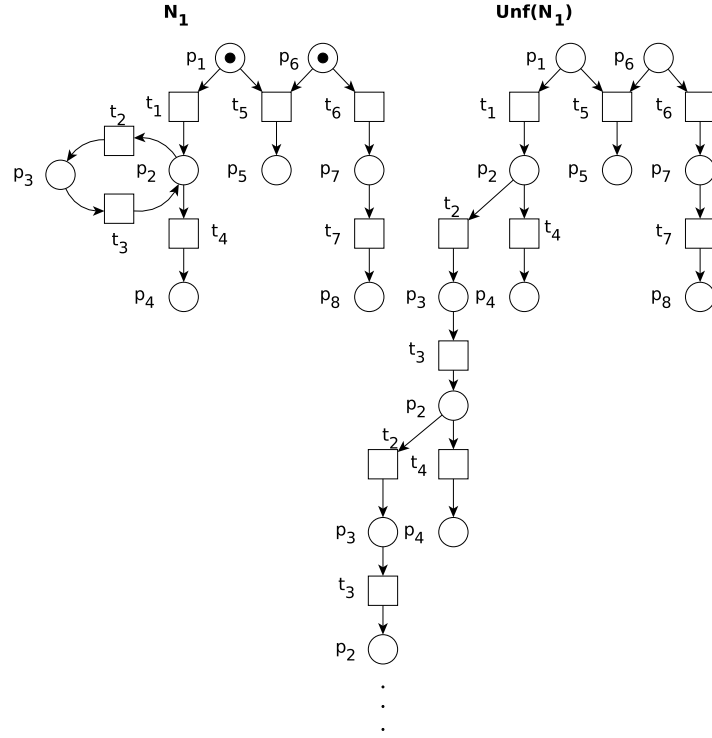


Fig. 1. A Petri net and its unfolding

Example 1. To give a simple example on the original reveals and extended-reveals notions, we examine the occurrence net in Fig. 2. In this net, $e_2 \triangleright e_4$ and $e_4 \triangleright e_2$. In general reveals relation is not symmetrical. As an example, $e_6 \triangleright e_4$ but $e_4 \not\triangleright e_6$ since after e_4 , e_7 can occur instead of e_6 .

In the same occurrence net, the occurrence of e_1 does not necessarily mean that e_5 will occur, but e_1 together with e_2 extended-reveals e_5 , denoted as $\{e_1, e_2\} \rightarrow \{e_5\}$. The occurrence of e_4 reveals neither e_6 nor e_7 . However, it reveals that either e_6 or e_7 will occur, denoted as $\{e_4\} \rightarrow \{e_6, e_7\}$.

3 Excludes and Reveals Relations on Petri Nets

In this section, we first introduce a new relation between transitions, called *excludes*, which will be used to detect negative information flow. Later, we define a *reveals* and an *extended-reveals* relation on the set of transitions of a Petri net, relying on the corresponding relations on occurrence nets as recalled in Section 2. Moreover, we introduce a new parametric relation, called *repeated-reveals*, again on the set of transitions of a Petri net. Reveals, extended-reveals and repeated-

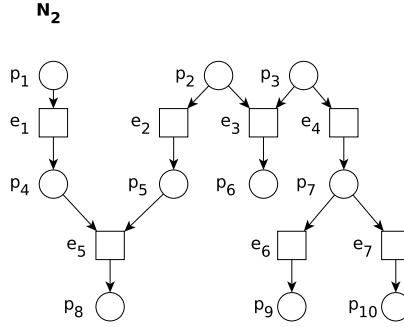


Fig. 2. An occurrence net.

reveals relations will be used to detect positive information flow, however they can also be applied in other areas, e.g. fault diagnosis as explored in [8] by using original reveals relation on occurrence nets. In the following three definitions we assume progress in the behavior of the nets, which means that a constantly enabled transition occurs if it is not disabled by another transition. This means that we consider only maximal runs in the unfolding.

Definition 4. Let $N = (P, T, F, m_0)$ be a Petri net and $\text{Unf}(N) = ((B, E, F), \lambda)$ be its unfolding, Ω be the set of all its maximal runs. Let $t_1, t_2 \in T$ be two transitions, we say t_1 excludes t_2 , denoted $t_1 \text{ ex } t_2$, iff $\forall \omega \in \Omega \ E_{t_1} \cap \omega \neq \emptyset \implies E_{t_2} \cap \omega = \emptyset$, i.e., they never appear in the same run.

It is easy to see that excludes is a symmetric relation and it is not transitive as well as obviously not reflexive.

In the case of Petri nets whose underlying net is an acyclic graph, if two transitions are in conflict, i.e., they are both enabled and the firing of one disables the other one, then one excludes the other. However, in general, transitions which are in conflict can still appear in the same maximal run and therefore they could be in not-excludes relation.

Example 2. The transitions t_2 and t_4 of N_1 in Fig. 1 are in conflict whereas $\neg(t_2 \text{ ex } t_4)$. In the unfolding in the same figure, it is possible to see a maximal run including occurrences of both.

$t_5 \text{ ex } t_4$ although they are not in conflict.

$t_7 \text{ ex } t_5$, $t_5 \text{ ex } t_1$ but $\neg(t_7 \text{ ex } t_1)$, indeed the relation is not transitive.

Definition 5. Let $N = (P, T, F, m_0)$ be a Petri net, and $\text{Unf}(N) = ((B, E, F), \lambda)$, be the unfolding of N . Let Ω be the set of all maximal runs of N . Let $t_1, t_2 \in T$ be two transitions, we say that t_1 reveals t_2 , denoted $t_1 \triangleright_{tr} t_2$, iff $\forall \omega \in \Omega \ E_{t_1} \cap \omega \neq \emptyset \implies E_{t_2} \cap \omega \neq \emptyset$.

We say transition t_1 reveals transition t_2 if and only if each maximal run which contains an occurrence of t_1 also contains at least one occurrence of t_2 . This

means that for each observation of t_1 , t_2 has been already observed or will be observed.

Remark 1. The reveals relation on transitions is *reflexive* and *transitive*, i.e., let $N = (P, T, F, m_0)$ be a Petri net, $t_1, t_2, t_3 \in T$, then $t_1 \triangleright_{tr} t_1$, and $(t_1 \triangleright_{tr} t_2 \wedge t_2 \triangleright_{tr} t_3) \implies t_1 \triangleright_{tr} t_3$.

Example 3. In the net N_1 , in Fig. 1, t_3 reveals both t_2 and t_1 . It is easy to notice that to be able to fire t_3 we must first fire t_1 and t_2 . In fact, in the unfolding, $\text{Unf}(N_1)$, given in Fig. 1, for each occurrence of t_3 there is at least one occurrence of t_2 and similarly, for each occurrence of t_3 there is at least one occurrence of t_1 . However, t_1 does not reveal t_2 or t_3 , since there is a run in which t_1 occurs and neither t_2 nor t_3 occurs. If an observer, who knows the structure of N_1 , can only observe t_1 he cannot have information about t_2 or t_3 , however if he is able to observe t_3 , he can deduce that t_2 and t_1 must have occurred.

Transition t_1 also reveals transition t_6 because when t_1 fires, t_5 cannot fire anymore and, since the net progresses, t_6 must fire. Since we do not assume strong fairness, $t_1 \not\triangleright_{tr} t_4$, after the occurrence of t_1 , t_2 and t_3 can loop forever. Reveals relation is not only about past occurrences but also about future occurrences. Observing t_1 does not tell us when t_6 fires. It might have fired already or it will fire in the future. $t_1 \triangleright_{tr} t_6$ tells us that when t_1 occurs, an occurrence of t_6 is inevitable.

Remark 2. Reveals relation is neither symmetric nor antisymmetric. For example, in Fig. 1, $t_2 \triangleright_{tr} t_3$ and $t_3 \triangleright_{tr} t_2$, however $t_2 \triangleright_{tr} t_1$ and $t_1 \not\triangleright_{tr} t_2$.

In some cases, one transition alone does not give much information about the behavior of the net whereas a set of transitions together can give some information about the behavior of the net. This relation is defined as in the following.

Definition 6. Let $N = (P, T, F, m_0)$ be a Petri net, $\text{Unf}(N) = ((B, E, F), \lambda)$ be its unfolding and Ω be the set of all maximal runs. Let $W, Z \subseteq T$ and W extended-reveals Z , denoted $W \rightarrow_{tr} Z$, iff $\forall \omega \in \Omega$

$$\bigwedge_{t \in W} (\omega \cap E_t \neq \emptyset) \implies \bigvee_{t \in Z} (\omega \cap E_t \neq \emptyset)$$

We say that a set of transitions W extended-reveals another set of transitions Z , if and only if each maximal run, which contains at least an occurrence of each transition in W , also contains at least an occurrence of a transition in Z .

The reveals relation on transitions, $t_1 \triangleright_{tr} t_2$, corresponds to the extended-reveals relation between singletons, $\{t_1\} \rightarrow_{tr} \{t_2\}$.

Example 4. In the net shown in Fig. 3, t_2 alone does not reveal t_5 , whereas t_2 and t_3 together tell us that t_5 will fire, denoted as $\{t_2, t_3\} \rightarrow_{tr} \{t_5\}$. In the same net, the occurrence of t_5 tells us that either t_8 or t_9 will fire, denoted as $\{t_5\} \rightarrow_{tr} \{t_8, t_9\}$. Similarly, $\{t_7, t_8\} \rightarrow_{tr} \{t_{10}\}$, i.e., there is no maximal run which includes occurrences of t_7 , t_8 and not t_{10} .

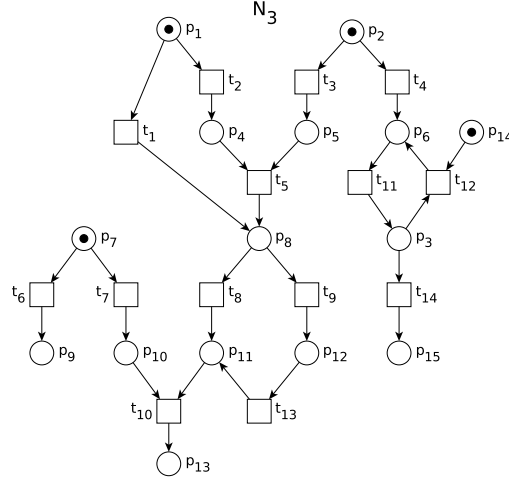


Fig. 3.

In some cases, repeated occurrences of the same transition can give more information about the behavior of a net than only one occurrence of that transition. A relation based on this fact is defined in the following.

Definition 7. Let $N = (P, T, F, m_0)$ be a Petri net, $\text{Unf}(N) = ((B, E, F), \lambda)$ be its unfolding and R be the set of all runs. Let $t_1, t_2 \in T$ be two transitions of N , and n be a positive integer. Let $R_{t_i}^n = \{\omega \in R : |\omega \cap E_{t_i}| = n\}$ and $\Omega_{t_i}^n$ denotes the set of maximal runs in $R_{t_i}^n$ with respect to set inclusion (i.e., $\Omega_{t_i}^n \subseteq R_{t_i}^n$ such that if $u, v \in \Omega_{t_i}^n \wedge u \subseteq v$ then $u = v$).

If $\Omega_{t_1}^n \neq \emptyset$ then t_1 n -repeated reveals t_2 , denoted $t_1 \text{ Re}_{\triangleright_{tr}}^n t_2$, iff $\forall \omega \in \Omega_{t_1}^n E_{t_2} \cap \omega \neq \emptyset$.

If $\Omega_{t_1}^n = \emptyset$ then $t_1 \text{ Re}_{\triangleright_{tr}}^n t_2$ is not defined.

Notation. $t_1 \text{ Re}_{\not\triangleright_{tr}}^n t_2$ will denote that there is at least one run in $\Omega_{t_1}^n$ such that t_1 appears n times and t_2 does not appear. $\neg(t_1 \text{ Re}_{\triangleright_{tr}}^n t_2)$ will denote that either $t_1 \text{ Re}_{\triangleright_{tr}}^n t_2$ is not defined, or $t_1 \text{ Re}_{\not\triangleright_{tr}}^n t_2$.

Example 5. Let us consider N_3 in Fig. 3. Transition t_{11} does not reveal t_{12} , however if the occurrence of t_{11} is observed twice then it is evident that t_{12} occurred, therefore t_{11} 2-Repeated reveals t_{12} , denoted $t_{11} \text{ Re}_{\triangleright_{tr}}^2 t_{12}$, whereas $t_{11} \text{ Re}_{\not\triangleright_{tr}}^1 t_{12}$ since after the first occurrence of t_{11} , t_{14} can fire instead of t_{12} .

Note that $t_{11} \text{ Re}_{\triangleright_{tr}}^3 t_{12}$ and $t_{11} \text{ Re}_{\not\triangleright_{tr}}^3 t_{12}$ are both not defined since t_{11} can fire at most twice, therefore in this case $\neg(t_{11} \text{ Re}_{\triangleright_{tr}}^3 t_{12})$.

Proposition 1. Let $N = (P, T, F, m_0)$ be a Petri net, $\text{Unf}(N) = ((B, E, F), \lambda)$ its unfolding and R be the set of all runs. Let $t_1, t_2 \in T$ be two transitions of N ,

$$t_1 \text{ Re}_{\triangleright_{tr}}^1 t_2 \implies t_1 \triangleright_{tr} t_2$$

Proof. Let $R_{t_1}^1 = \{\omega \in R : |\omega \cap E_{t_1}| = 1\}$ and $\Omega_{t_1}^1$ be the set of maximal runs in $R_{t_1}^1$. If $t_1 \text{ Re}_{\triangleright_{tr}}^1 t_2$, then $\Omega_{t_1}^1 \neq \emptyset$ and $\forall \omega \in \Omega_{t_1}^1 \omega \cap E_{t_2} \neq \emptyset$. Let σ be an arbitrary maximal run of $\text{Unf}(N)$. Suppose that $\sigma \cap E_{t_1} \neq \emptyset$ then we can always take a run $\omega \in \Omega_{t_1}^1$ such that $\omega \subseteq \sigma$. Then we know that σ contains at least one occurrence of t_2 and so $t_1 \triangleright_{tr} t_2$. \square

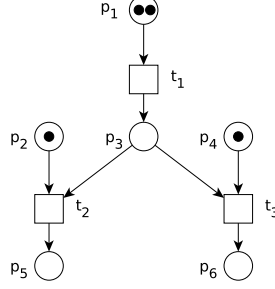


Fig. 4.

However, the implication of the previous proposition does not hold in the other direction. In fact, consider the net in Fig. 4, $t_1 \triangleright_{tr} t_2$, $t_1 \triangleright_{tr} t_3$, $t_1 \text{ Re}_{\not\triangleright_{tr}}^1 t_2$ and $t_1 \text{ Re}_{\not\triangleright_{tr}}^1 t_3$. The main difference is that we consider only maximal runs for reveals relation. For this net there is only one maximal run which contains t_1 (twice), t_2 and t_3 . However, there is a run in $\Omega_{t_1}^1$ in which t_1 appears and t_2 does not appear, as well as a run in which t_1 appears and t_3 does not appear. All runs in $\Omega_{t_1}^2$, i.e., including t_1 twice, contain both t_2 and t_3 , i.e., $t_1 \text{ Re}_{\triangleright_{tr}}^2 t_2$ and $t_1 \text{ Re}_{\triangleright_{tr}}^2 t_3$.

Proposition 2. Let $N = (P, T, F, m_0)$ be a Petri net, $\text{Unf}(N) = ((B, E, F), \lambda)$ be its unfolding and R be the set of all runs. Let $t_1, t_2 \in T$ be two transitions, if $t_1 \text{ Re}_{\triangleright_{tr}}^n t_2$ and $\Omega_{t_1}^{n+1} \neq \emptyset$ then $t_1 \text{ Re}_{\triangleright_{tr}}^{n+1} t_2$.

Proof. Let $R_{t_1}^n = \{\omega \in R : |\omega \cap E_{t_1}| = n\}$ and $\Omega_{t_1}^n$ be the set of maximal runs in $R_{t_1}^n$. If $t_1 \text{ Re}_{\triangleright_{tr}}^n t_2$, then $\Omega_{t_1}^n \neq \emptyset$ and $\forall \omega \in \Omega_{t_1}^n \omega \cap E_{t_2} \neq \emptyset$. Let $\sigma \in \Omega_{t_1}^{n+1}$, we can always choose a run $\omega \in \Omega_{t_1}^n$ such that $\omega \subseteq \sigma$. Then we know that $\sigma \cap E_{t_2} \neq \emptyset$, so $t_1 \text{ Re}_{\triangleright_{tr}}^{n+1} t_2$. \square

4 Non-interference

In this section, before introducing the new notions, we briefly recall the most used non-interference notions in the literature and discuss our motivation for introducing new non-interference notions based on reveals and excludes relations.

The notions recalled in the following are based on some notion of low observability of a system. It is what can be observed of a system from the point of view of low users.

There are mainly two kinds of information flows that non-interference notions deal with. These are *positive information flow* and *negative information flow*. A positive information flow arises when the occurrence of a high level transition can be deduced from the low level behavior of the system, whereas a negative information flow is concerned with the non-occurrences of a high transition.

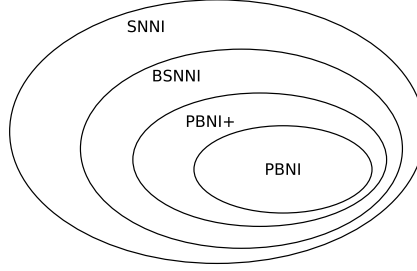


Fig. 5. Relation between some existing interference notions in the literature. $SNNI \equiv NDC$, $BSNNI \subseteq SNNI$, $SBNDC \equiv BNDC \equiv PBNI+ \subseteq BSNNI$, $PBNI \subseteq PBNI+$ (see [7])

In the following, we will use acronyms to denote the set of nets satisfying the corresponding security notion.

The less restrictive notion, introduced in [6, 3] and also studied on 1-safe Petri nets in [7], is *Strong Nondeterministic Non-Interference* (*SNNI*). It is a trace-based property (trace as sequence of event occurrences), that intuitively says that a system is secure if what the low part can see does not depend on what the high level part does. If a net system N is *SNNI* secure, then it should offer, from the low point of view, the same traces as the system where the high level transitions are prevented. In *SNNI* secure systems, information can flow from low to high but not from high to low. A different characterization of the same notion, called *Non-Deducibility on Composition* (*NDC*), is given in [7].

While *SNNI* is based on trace equivalence, the more restrictive notions *Bisimulation based Strong Nondeterministic Non-Interference* (*BSNNI*) and *Bisimulation based Non-Deducible on Composition* (*BNDC*) are based on bisimulation.

Strong Bisimulation based Non-Deducible on Composition (*SBNDC*) is an alternative characterization of *BNDC* [6, 3]. In fact, Busi and Gorrieri in [7] show that *BNDC* is equivalent to *SBNDC*, and it is stronger than *BSNNI*.

Another non-interference notion called *Place Based Non-Interference* (*PBNI*) was introduced in [7]. It is based on the absence of some kinds of specific places in the net, namely causal and conflict places. A causal place is a place between a low transition and a high transition such that the low transition consumes the token from the place which was produced by the high transition. A conflict place is a place such that at least one low transition and one high transition consume a token from it. A net is considered to be *PBNI* secure in the absence of such

places. In [7], it is shown that if a net is *PBNI* secure then it is also *SBNDC* secure.

In [12], a similar notion, called *Positive Place Based Non-Interference (PBNI+)*, is proposed by introducing the notions of active causal and active conflict places. *PBNI+* is weaker than *PBNI* and it coincides with *SBNDC*.

The overall relationship between these mentioned notions is illustrated in Fig. 5. In the rest of the paper, we will refer only to the notions which are illustrated in the figure since the others are equivalent to those.

With respect to the above mentioned different kinds of information flow, *SNNI*, *BSNNI* and *PBNI+* deal with positive information flow, whereas *PBNI* deals also with negative information flow.

All these notions seem to aim mainly at deducing past occurrences of high transitions, for example they all consider system N_6 in Fig. 7 secure, whereas, by considering progress, after the occurrence of l , a low user deduces h is inevitable and therefore N_6 is not secure with respect to the ability of deducing information about the future behavior.

Differently from the previous notions, the ones we are going to propose do not only capture information flow about past occurrences of high transitions, but also information flow about inevitable or impossible future occurrences of high transitions.

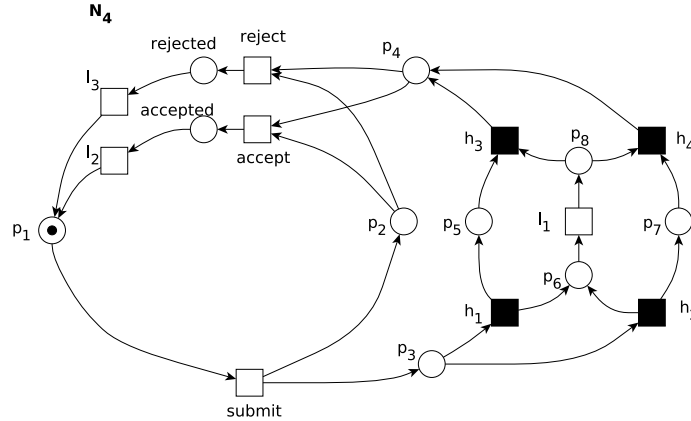


Fig. 6. A net modeling paper submission and evaluation.

In some cases, the mere ability to deduce that some high transition has occurred is not a security threat, provided the low user cannot know which one occurred.

Let us illustrate this issue with the help of an example. The net in Fig. 6 represents a system in which a user can repeatedly submit a paper to a committee, each time receiving a judgment (accept or reject). The black squares represent

high transitions. The review process can follow either of two paths, and we do not want the user to know which one was chosen. When the user receives an answer, he knows that some high transition occurred, however he cannot infer which one.

For this reason, the new notions we are going to introduce in the following will consider such a system secure, whereas it is not secure with respect to *SNNI*, and the other above recalled notions.

In the sequel, the set of high transitions will be denoted by H and the set of low transitions will be denoted by L .

4.1 Non-Interference Based on Reveals

Reveals-based Non-Interference accepts a net as secure if no low transition reveals any high transition.

Definition 8. Let $N = (P, T, F, m_0)$ be a Petri net, $T = H \cup L$, $H \cap L = \emptyset$, $L, H \neq \emptyset$. N is secure with respect to Reveals-based Non-Interference (RNI) iff $\forall l \in L \forall h \in H: l \not\triangleright_{tr} h$.

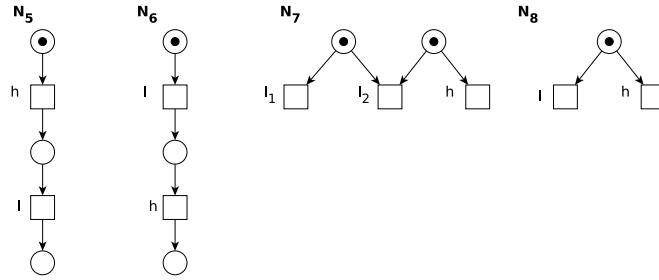


Fig. 7.

Example 6. N_4 in Fig. 6 is *RNI* secure. N_5 and N_6 in Fig. 7 are not secure with respect to *RNI*, since in both nets a low transition reveals a high transition, i.e., $l \triangleright_{tr} h$. An observer who knows the structure of the net can deduce that h has already fired in N_5 by observing l . For N_6 , again by observing l , he can deduce that h will fire. N_7 in Fig. 7 is also not secure in this context because the observation of l_1 tells the observer that h has already fired or will fire since l_2 cannot fire anymore.

With *RNI*, we are able to capture positive information flow. Moreover, we not only capture past occurrences of high transitions but also future occurrences, and this is because of the progress assumption.

Although it is useful to capture positive information flow, *RNI* is not able to capture the negative information flow. N_8 in Fig. 7 is considered to be secure with respect to *RNI* since it cannot capture the flow between h and l . However, an observer could deduce that h has not fired and will not fire in the future by observing the occurrence of l . In Section 4.4 we will introduce a notion which deals with this kind of information flow.

4.2 Non-Interference Based on Extended-Reveals

As explained in Section 3, in some cases, a transition does not tell much about the behavior of the net, whereas a set of transitions together gives some more information. Extended-reveals deals with this relation between transitions of a Petri net. We propose to use this relation in order to define a new non-interference notion in which the occurrences of a set of low transition together give information about some high transitions.

Definition 9. Let $N = (P, T, F, m_0)$ be a Petri net, $T = H \cup L$, $H \cap L = \emptyset$, $L, H \neq \emptyset$, $|L| \geq k \geq 1$. N is secure with respect to k -Extended-Reveals based Non-Interference (k -ERNI) iff $\forall \{l_1, \dots, l_k\} \subseteq L \forall h \in H, \{l_1, \dots, l_k\} \not\rightarrow_{tr} \{h\}$.

N is ERNI secure if it satisfies the above condition for $k = |L|$.

Intuitively, we say that a net is k -ERNI secure, if an attacker is not able to deduce information about the hidden part of the net by observing occurrences of k low level transitions. If a net is k -ERNI secure then it is secure with respect to all n -ERNI where $1 \leq n \leq k$.

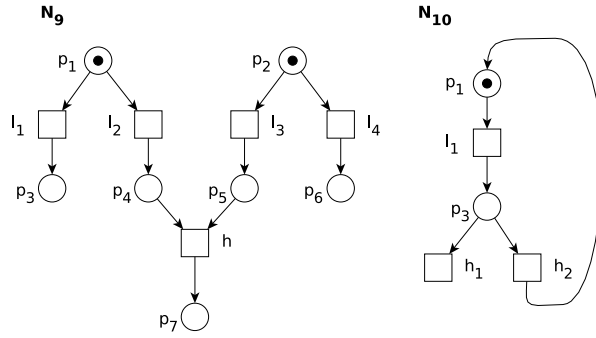


Fig. 8.

Example 7. N_9 in Fig. 8 is not secure with respect to 2-ERNI. When l_2 and l_3 occur, a low level observer can deduce that h will occur, i.e., $\{l_2, l_3\} \rightarrow_{tr} \{h\}$. In this net, the occurrence of only one low transition does not give sufficient

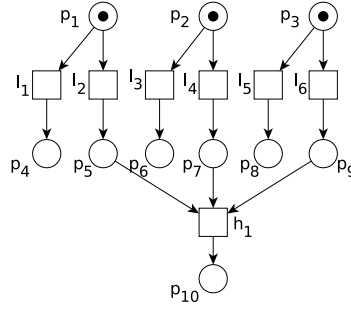


Fig. 9.

information about any high transitions, whereas the occurrence of two low level transitions together does. In the net in Fig. 9, no low transition alone reveals a high transition as well as no pair of low level transitions reveals a high transition. However, $\{l_2, l_4, l_6\} \rightarrow_{tr} \{h_1\}$, i.e., a low user, observing that all these three transitions occurred, can deduce that h_1 will inevitably occur. Thus, this net is 2-ERNI secure whereas it is not 3-ERNI secure.

Obviously, 1-ERNI coincides with RNI, where no low transition alone reveals a high transition. Moreover, $k\text{-ERNI} \subseteq \text{RNI}$, for $k \geq 1$. N_9 is RNI secure since none of the low transitions reveals a high transition alone.

4.3 Non-Interference Based on Repeated-Reveals

Another case can be the one in which an attacker is not able to deduce information by observing low transitions and this is because only repeated occurrence of a low transition gives information about the hidden part of the net. Thus, we assume that the attacker can count the occurrences of low transitions and so he can deduce information about the high transitions.

Definition 10. Let $N = (P, T, F, m_0)$ be a Petri net, $T = H \cup L$, $H \cap L = \emptyset$, $L, H \neq \emptyset$. Let $\text{Unf}(N)$ be the unfolding of N , where $\text{Unf}(N) = ((B, E, F, c_0), \lambda)$, $\lambda : B \cup E \rightarrow P \cup T$. Let $n > 0$.

N is secure with respect to n -Repeated-Reveals based Non-Interference ($n\text{-ReRNI}$) iff $\forall l \in L \ \forall h \in H \ \forall m \leq n \ \neg(l \text{ Re}_{\triangleright_{tr}}^m h)$.

N is ReRNI, iff it is $n\text{-ReRNI}$ for all $n > 0$.

Proposition 3. $n\text{-ReRNI} \implies (n-1)\text{-ReRNI}$

The proof follows from the definition.

Example 8. N_{10} in Fig. 8 is not 2-ReRNI secure. Although the first occurrence of l_1 does not reveal a high transition, by observing its second occurrence an observer can deduce that h_2 occurred. However, the net is RNI secure as well

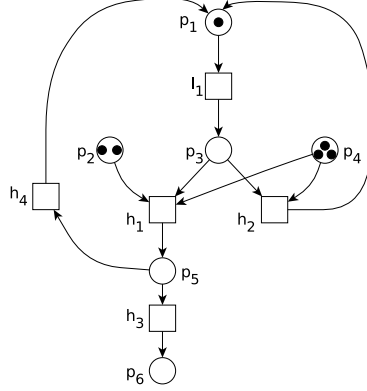


Fig. 10.

as *ERNI* secure. In the net in Fig. 10, an observer cannot infer about the high transitions by observing l_1 occurring only once. Also the second occurrence of l_1 does not tell the observer which high transition occurred or will occur. However, the observer can deduce that h_2 has already occurred or will occur inevitably if he observes three occurrences of l_1 . Therefore, this net is *2-ReRNI* secure but it is not *3-ReRNI* secure. Note that if the transition h_3 was absent then every maximal run would include at least one occurrence of h_2 and then, even without observing l_1 , the occurrence of h_2 would be inevitable.

The following proposition is directly derived from Prop. 1.

Proposition 4. *If a net is RNI secure then it is 1-ReRNI secure.*

However, the previous implication does not hold in the opposite direction. Consider the net in Fig. 4 and let t_1 be a low transition, t_2 and t_3 be high transitions. This net is *1-ReRNI* secure since the first occurrence of t_1 does not reveal information about t_2 and t_3 , as discussed in Example 5. However the net is not *RNI* secure since $t_1 \triangleright_{tr} t_2$ and $t_1 \triangleright_{tr} t_3$. Note that this net is not secure with respect to *2-ReRNI* since the second occurrence of t_1 reveals both t_2 and t_3 , i.e. $t_1 \text{ } Re_{\triangleright_{tr}}^2 t_2$ and $t_1 \text{ } Re_{\triangleright_{tr}}^2 t_3$.

Although *k-ERNI* and *n-ReRNI* are not comparable since they are parametric notions which are based on observing different things (for *k-ERNI* it is observation of occurrences of different low transitions together whereas for *n-ReRNI* it is observation of multiple occurrences of the same low transition) there are nets which are secure with respect to both and which are secure with respect to only one of them.

Both *k-ERNI* and *n-ReRNI* catch positive information flow about the past or future occurrences of high transitions, whereas they allow negative information flow. In the following we will introduce a notion considering both positive and negative information flow.

4.4 Positive/Negative Non-Interference Based on Reveals and Excludes

Until now we explored positive information flow on Petri nets. In order to catch negative information flow which is related to non-occurrence of high transitions, we need to consider the excludes relation between low and high transitions, as introduced in Def. 4.

Definition 11. Let $N = (P, T, F, m_0)$ be a Petri net, $T = H \cup L$, $H \cap L = \emptyset$, $L, H \neq \emptyset$. N is secure with respect to Positive/Negative Non-Interference (PNNI) iff $\forall l \in L \forall h \in H$, $l \not\vdash_{tr} h$ and $\neg(l \underline{ex} h)$.

If in a Petri net N , no low transition reveals a high transition and no low transition excludes a high transition, N is considered to be *PNNI* secure. *PNNI* is stronger than *RNI*, i.e., $PNNI \subseteq RNI$, and this follows directly from the definitions. In order to be *PNNI* secure, a net has to be *RNI* secure (no low transition reveals a high transition) and to satisfy an additional requirement (no low transition excludes a high transition).

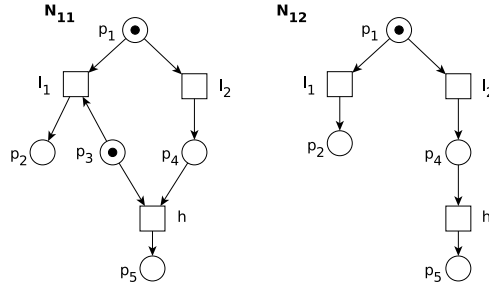


Fig. 11.

Example 9. Both N_{11} and N_{12} in Fig. 11 are not *PNNI* secure since a low transition l_1 excludes a high transition h . Thus, by observing occurrence of l_1 , an observer can deduce that h did not occur and will not occur.

N_{13} in Fig. 12 is not secure with respect to *PNNI* because of the negative information flow, i.e., l_2 excludes h_1 as well as it excludes h_2 . An observer can deduce that none of the high transitions occurred and they will not occur in the future by observing l_2 or l_3 . This net is *RNI*, *ERNI* and *ReRNI* secure.

In the same figure, N_{14} is a *PNNI* secure Petri net. No low transition reveals a high transition as well as no low transition excludes a high transition. However an observer is able to deduce that h_1 will occur inevitably by observing the occurrences of both l_2 and l_3 , i.e., $\{l_2, l_3\} \rightarrow_{tr} \{h_1\}$. In other words, this net is not 2-*ERNI* while it is *RNI* and *ReRNI* secure.

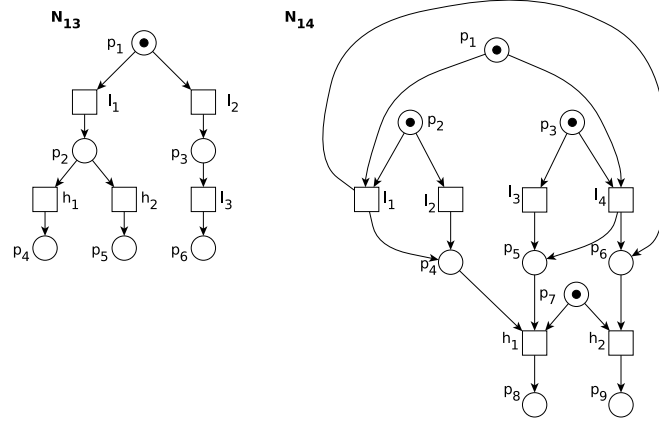


Fig. 12.

As seen in the previous example, $PNNI$ is strictly stronger than RNI .

$PNNI$ and $k-ERNI$ are intersecting for any k , $PNNI \cap k-ERNI \neq \emptyset$, $PNNI \setminus k-ERNI \neq \emptyset$, $k-ERNI \setminus PNNI \neq \emptyset$. None of them is stronger than the other one. The net N_{15} in Fig. 13 is both $ERNI$ and $PNNI$ secure, whereas N_{16} in Fig. 13 is not $PNNI$ secure, however it is $ERNI$ secure. N_{14} of Fig. 12 is $PNNI$ secure, whereas it is not secure with respect to $2-ERNI$ as it is discussed in example 9.

$PNNI$ and $n-ReRNI$ are also intersecting for any n . A net which is both $PNNI$ and $ReRNI$ secure is the one in Fig. 6. The net in Fig. 10 is not secure with respect to $3-ReRNI$ whereas it is $PNNI$ secure. If we add to the net another low transition l_2 which consumes a token from p_5 , the net becomes not secure with respect to $PNNI$ as well as with respect to RNI , since l_2 reveals h_1 .

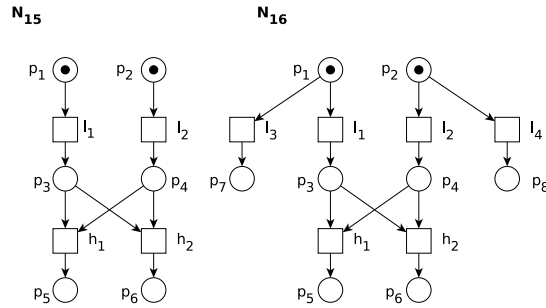


Fig. 13.

5 Comparison of Non-interference Notions

We have introduced new notions of non-interference for Petri nets. These notions are based on the reveals and the excludes relations and on the progress assumption.

One major difference between these notions with the existing ones, recalled in Section 4, is that the new notions explicitly consider the information flow both about the past and the future occurrences of high transitions. For example, if a low user can tell that the occurrence of a high transition is inevitable in the future, such a system is considered to be not secure according to the notions we have here introduced, whereas it is considered secure by the old notions such as *SNNI*, *BSNNI*, *PBNI+* and *PBNI*. Similarly, for the negative information flow, we consider both past and future non-occurrences of high transitions.

Another important difference is shown by N_4 in Fig. 6. This net is not secure according to *SNNI* even if a low user cannot infer which high transitions actually occurred. On the other hand, it is secure with respect to all non-interference notions based on reveals and excludes, since these require the capability of differentiating among the high transitions.

Moreover, the notions recalled in Section 4 are defined for 1-safe Petri nets, whereas *RNI*, *k-ERNI*, *n-ReRNI* and *PNNI* are defined for general Petri nets.

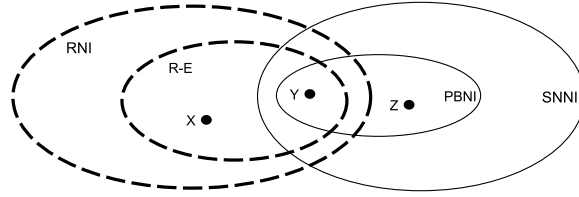


Fig. 14.

Figure 14 illustrates the relation between our notions and the other notions we have discussed so far. For the sake of simplicity, we only consider the weakest (*SNNI*) and the strongest (*PBNI*) notions from the ones recalled in Section 4. with the weakest of the new notions, i.e., *RNI*, and with the intersection set, denoted *R-E* in Fig. 14, of the new notions *RNI*, *k-ERNI*, *n-ReRNI* and *PNNI*.

We will examine three examples to discuss the differences of these classes.

A net which is secure with respect to all notions based on reveals and excludes and which is not secure with respect to *SNNI* is denoted by *X* in Fig. 14 and it is the one in Fig. 6. We consider this net secure since an observer cannot differentiate among the high transitions even if he can know some high actions have been performed (or will be performed). However, this net is not secure with respect to *SNNI*.

The net denoted by Y in Fig. 14 is secure with respect to all non-interference notions based on reveals and excludes as well as with respect to $PBNI$. This net can be N_{15} in Fig. 13. This net is secure since no low transition reveals a high transition (alone or together with another transition) as well as no low transition excludes a high transition. Thus there is neither positive nor negative information flow. It is also secure with respect to $PBNI$ due to the fact that there is no active causal or active conflict place.

Two nets which are secure with respect to $PBNI$ but not secure with respect to any of the non-interference notions based on reveals and excludes, denoted by Z in Fig. 14, are for example N_6 in Fig. 7 and N_{12} in Fig. 11.

6 Conclusion

In this paper, we have proposed several new notions of non-interference for Petri nets, and compared them with notions already proposed in the literature. In this approach, the transitions of a system net are partitioned into two disjoint sets: the low and the high transitions. A system net is considered *secure*, or *free from interference*, if, from the observation of the occurrence of a low transition, or a set of low transitions, it is not possible to infer information on the occurrence of a high transition. Our new non-interference notions rely on net unfolding and on two relations among transitions. The first one is an adaptation to Petri nets of the *reveals* relation, previously defined on occurrence nets and not yet considered in this context; in particular we have introduced a class of parametrized reveals relations for Petri nets. The second relation is called *excludes* and it has been introduced here with the aim of capturing negative information flow.

The notion of RNI states that a net is secure if no low transition reveals any high transition. We have shown that this notion captures some situations which were not captured by the existing notions. We also propose more restrictive notions: $k-ERNI$ based on observing occurrences of multiple low transitions and $n-ReRNI$ based on the ability of the low user to count the occurrences of a low transition.

By adding the *excludes* relation to the picture, we allow one to infer negative information, namely the fact that a high transition has not occurred and will not occur. This is the basis of $PNNI$. The paper includes a comparison between the notions introduced here and those found in the literature on the subject.

The notions proposed in this paper, and further variants of them, should now be tested on more realistic cases. Our aim is to build a collection of different non-interference properties, so that a system designer, or a system analyzer, can choose those more appropriate to a specific case. A generalization could be a non-interference notion based on a parametric reveals relation between multisets of transitions.

We are currently starting to explore algorithms to check non-interference. In particular, along a similar line to that followed in [13], we are evaluating the use of finite prefixes of the unfoldings of nets.

We are also interested in further investigating the excludes relation and the possibility to apply it in different contexts.

Acknowledgements

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. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. (1982) 11–20
2. Ryan, P.Y.A.: Mathematical models of computer security. [14] 1–62
3. Focardi, R., Gorrieri, R.: A taxonomy of security properties for process algebras. *Journal of Computer Security* **3** (1995) 5–34
4. Roscoe, A.W.: Csp and determinism in security modelling. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (1995) 114–127
5. Ryan, P.Y.A., Schneider, S.A.: Process algebra and non-interference. In: CSFW, IEEE Computer Society (1999) 214–227
6. Focardi, R., Gorrieri, R.: Classification of security properties (part i: Information flow). [14] 331–396
7. Busi, N., Gorrieri, R.: A survey on non-interference with Petri nets. In Desel, J., Reisig, W., Rozenberg, G., eds.: *Lectures on Concurrency and Petri Nets*. Volume 3098 of *Lecture Notes in Computer Science*, Springer (2003) 328–344
8. Haar, S.: Unfold and cover: Qualitative diagnosability for Petri nets. In: *Proc. 46th IEEE Conference on Decision and Control*. (2007)
9. Engelfriet, J.: Branching processes of Petri nets. *Acta Inf.* **28** (1991) 575–591
10. Balaguer, S., Chatain, T., Haar, S.: Building tight occurrence nets from reveals relations. In Caillaud, B., Carmona, J., Hiraishi, K., eds.: *11th International Conference on Application of Concurrency to System Design, ACS D 2011, Newcastle Upon Tyne, UK, 20-24 June, 2011*, IEEE (2011) 44–53
11. Balaguer, S., Chatain, T., Haar, S.: Building occurrence nets from reveals relations. *Fundam. Inform.* **123** (2013) 245–272
12. Busi, N., Gorrieri, R.: Positive non-interference in elementary and trace nets. In Cortadella, J., Reisig, W., eds.: *Applications and Theory of Petri Nets 2004, 25th International Conference, ICATPN 2004, Bologna, Italy, June 21-25, 2004, Proceedings*. Volume 3099 of *Lecture Notes in Computer Science*, Springer (2004) 1–16
13. Baldan, P., Carraro, A.: Non-interference by unfolding. In Ciardo, G., Kindler, E., eds.: *Application and Theory of Petri Nets and Concurrency - 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23-27, 2014. Proceedings*. Volume 8489 of *Lecture Notes in Computer Science*, Springer (2014) 190–209
14. Focardi, R., Gorrieri, R., eds.: *Foundations of Security Analysis and Design, Tutorial Lectures* [revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design, FOSAD 2000, Bertinoro, Italy, September 2000]. In Focardi, R., Gorrieri, R., eds.: *FOSAD*. Volume 2171 of *Lecture Notes in Computer Science*, Springer (2001)

Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification

Kent Inge Fagerland Simonsen^{1,2}, Lars M. Kristensen¹, and Ekkart Kindler²

¹ Department of Computing, Bergen University College, Norway

² DTU Compute, Technical University of Denmark, Denmark

Abstract. PetriCode is a tool that supports automated generation of protocol software from a restricted class of Coloured Petri Nets (CPNs) called Pragmatics Annotated Coloured Petri Nets (PA-CPNs). PetriCode and PA-CPNs have been designed with five main requirements in mind, which include the same model being used for verification and code generation. The PetriCode approach has been discussed and evaluated in earlier papers already. In this paper, we give a formal definition of PA-CPNs and demonstrate how the specific structure of PA-CPNs can be exploited for verification purposes.

1 Introduction

Coloured Petri Nets (CPNs) [3] and CPN Tools have been widely used for modelling and verifying protocols. Examples include application layer protocols such as IOTP, SIP and WAP, transport layer protocols such as TCP, DCCP and SCTP, and network layer protocols such as DYMO, AODV, and ERDP [2, 8]. Formal modelling and verification have been useful in gaining insight into the operation of the protocols 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 to obtain an implementation of the protocol under consideration. In particular, rather limited research has been conducted into approaches that support automatic generation of protocol implementations from such CPN models. The earlier approaches have either restricted the target platform for code generation to the Standard ML language used by the CPN Tools simulator or have considered a specific target language based on platform-specific additions to the CPN models.

This has motivated us to develop an approach and an accompanying tool called PetriCode to support the automated generation of protocol software from CPN models. Our code generation approach is designed to satisfy five main requirements. Firstly, the approach must support *platform independence*, i.e., it must enable code generation for multiple languages and platforms from the same CPN model. Secondly, the approach must support *integration* of the generated code with third-party code. In particular, it must support *upwards integration*, i.e., the generated code must expose an explicit interface for service invocation, and it must support *downwards integration*, i.e., the ability of the generated code

to invoke and rely on underlying libraries. Thirdly, it must support *verification* in that the code generation capability should not introduce complexity problems for verification of the CPN models. Fourthly, the generated code must be *readable* to enable code review and performance enhancements. Finally, the approach must be *scalable* to industrial-sized protocols.

The foundation of our approach is a slightly restricted subclass of CPNs called *Pragmatic Annotated CPNs (PA-CPNs)*. The restrictions make explicit the structure of the protocol system, its principals, channels, and services. A key feature of this class of CPNs are so-called *code generation pragmatics*, which are syntactical annotations to certain elements of the PA-CPNs. These pragmatics represent concepts from the domain of communication protocols and protocol software, which are used to indicate the purpose of the respective modelling element. The role of the pragmatics is to extend the CPN modelling language with domain-specific elements and make implicit knowledge of the modeller explicit in the CPN model such that it can be exploited for code generation.

In earlier work [16], we have introduced PA-CPNs informally, and presented the PetriCode tool [17]. In [18], we demonstrated platform independence, integrateability, and readability of the generated code. In [19], we applied the approach for automatically generating an implementation of the industrial-strength WebSocket protocol. This included demonstrating that the generated code was interoperable with other implementations of the WebSocket protocol.

The contribution of this paper compared to our earlier work is threefold. Firstly, motivated by the practical relevance of the net class demonstrated in earlier work, we give a formal definition of PA-CPNs. Secondly, we discuss the process of developing protocol software with our approach from a methodology perspective. Thirdly, we show that PA-CPNs are amenable to verification. Specifically, we show how the structural restrictions of PA-CPNs allow us to add *service testers* to the model of the protocol, which reduce the state space of the model. Furthermore, the structural restrictions of PA-CPNs induce a natural progress measure that can be exploited for verification purposes by the *sweep-line state space exploration method* [4].

The rest of this paper is organised as follows: Section 2 provides the background definitions and notation of CPNs that are used throughout this paper. Section 3 gives the formal definition of PA-CPNs accompanied by an example outlining how PA-CPNs can be used to model a transport protocol. Section 4 discusses the modelling process of PA-CPNs from an application perspective. Section 5 formalises the concepts of tree decomposability of control flow nets which are central in generating code for the protocol services. Section 6 shows how to define progress measures for the sweep-line method based on service and service tester modules of PA-CPNs, and experimentally evaluate their effect on the verification of the transport protocol example. Finally, in Sect. 7, we sum up the conclusions and discuss related work. We assume that the reader is familiar with the basic concepts of Petri nets and high-level Petri nets such as CPNs. This paper is a condensed version of a technical report [20], which contains more motivation and detailed explanations of examples and concepts.

2 Background Definitions on Coloured Petri Nets

The definition of PA-CPNs is based on the standard definition of hierarchical CPNs [3]. Here, we briefly rephrase the definitions of CPNs. Readers familiar with these definitions can skip this section. In this paper, we provide the syntactical definitions of CPNs only, which will be restricted when defining PA-CPNs. Since PA-CPNs are a syntactical restriction of CPNs, we do not need change the semantics of CPNs at all.

A hierarchical CPN consists of a finite set of CPN modules, which we discuss first. Figures 1 and 2 show some CPN modules of our example (which will be used later). The modules of a hierarchical CPNs are related to each other via substitution transitions (shown with a double border) which can have associated submodules, and by linking places connected to the substitution transitions (called socket places) to places (called port places) on the associated submodules.

A *CPN module* consists of a set of *places* P and a set of *transition* T connected by a set of *directed arcs* A connecting either a transition and a place or a place and a transition. A CPN module additionally has a set of *colour sets* (types) Σ containing the types that can be used as colour sets of places and for typing a set of *variables* V which can be used in arc expressions and transition guards. In the formal definition, the colour set of each place (by convention written below a place) is specified by means of a *colour set function* that maps each place to a colour set determining the kind of tokens that may reside on the place. Each directed arc in a CPN module has an associated arc expression used to determine the tokens added and removed by the occurrence of an enabled transition and is specified by an *arc expression function*. The arc expression of each arc may contain variables from the set of variables V . The arc expressions are required to have a type such that the evaluation of an arc expression on an arc connected to a place p results in a multi-set of tokens over the colour set of the place. Transitions may have an associated guard expression specified by means of a *guard function* G which associates a boolean expression with each transition. The initial marking of each place is specified by means of an *initialisation function* I which maps each place into a (possibly empty) multi-set over the colour set of the place.

Definition 1 formally defines a CPN module. In the definition, we use $Type[v]$ to denote the type of a variable v , and we use $EXPR_V$ to denote the set of expressions with free variables contained in a set of variables V . For an expression e containing a set of free variables V , we denote by $e\langle b \rangle$ the result of evaluating e in a binding b that assigns a value to each variable in V . We use $Type[e]$ for an expression e (an arc expression, a guard, or an initial marking) to denote the type of e . For a non-empty set S , we use S_{MS} to denote the type corresponding to the set of all multi-sets over S .

Definition 1. A *Coloured Petri Net Module* (Def. 6.1 in [3]) is a tuple $CPN_M = (CPN, T_{sub}, P_{port}, PT)$, such that:

1. $CPN = (P, T, A, \Sigma, V, C, G, E, I)$ is a Coloured Petri Net (Def. 4.2 in [3]) where:

- (a) P is a finite set of **places** and T is a finite set of **transitions** T such that $P \cap T = \emptyset$.
- (b) $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed **arcs**.
- (c) Σ is a finite set of non-empty **colour sets** and V is a finite set of **typed variables** such that $\text{Type}[v] \in \Sigma$ for all variables $v \in V$.
- (d) $C : P \rightarrow \Sigma$ is a **colour set function** that assigns a colour set to each place.
- (e) $E : A \rightarrow \text{EXPR}_V$ is an **arc expression function** that assigns an arc expression to each arc a such that $\text{Type}[E(a)] = C(p)_{MS}$, where p is the place connected to the arc a .
- (f) $G : T \rightarrow \text{EXPR}_V$ is a **guard function** that assigns a guard to each transition t such that $\text{Type}[G(t)] = \text{Bool}$.
- (g) $I : P \rightarrow \text{EXPR}_\emptyset$ is an **initialisation function** that assigns an initialisation expression to each place p such that $\text{Type}[I(p)] = C(p)_{MS}$.
- 2. $T_{\text{sub}} \subseteq T$ is a set of **substitution transitions**.
- 3. $P_{\text{port}} \subseteq P$ is a set of **port places**.
- 4. $PT : P_{\text{port}} \rightarrow \{\text{IN}, \text{OUT}, \text{I/O}\}$ is a **port type function** that assigns a port type to each port place.

Socket places are not defined explicitly as part of a module because they are implicitly given via the arcs connected to the substitution transitions. For a substitution transition t , we denote by $ST(t)$ a mapping that maps each socket place p into its type, i.e., $ST(t)(p) = \text{IN}$ if p is an input socket, $ST(t)(p) = \text{OUT}$ if p is an output socket, and $ST(t)(p) = \text{I/O}$ if p is an input/output socket.

The definition of a hierarchical CPN is provided below. A hierarchical CPN consists of a set of disjoint CPN modules, a submodule function assigning a (sub)module to each substitution transition, and a port-socket relation that associates port places in a submodule to the socket places of its upper layer module. The set of socket places for a substitution transition t consists of the places connected to the substitution transition and is denoted by $P_{\text{sock}}(t)$. The definition requires that the module hierarchy (to be defined in Def. 3) is acyclic in order to ensure that there are only a finite number of instances of each module. Furthermore, port and socket places can only be associated with each other, if they have the same colour set and the same initial marking.

Definition 2. A **hierarchical Coloured Petri Net** (Def. 6.2 in [3]) is a four-tuple $CPN_H = (S, SM, PS, FS)$ where:

- 1. S is a finite set of **modules**. Each module is a **Coloured Petri Net Module** $s = ((P^s, T^s, A^s, \Sigma^s, V^s, C^s, G^s, E^s, I^s), T_{\text{sub}}^s, P_{\text{port}}^s, PT^s)$. It is required that $(P^{s_1} \cup T^{s_1}) \cap (P^{s_2} \cup T^{s_2}) = \emptyset$ for all $s_1, s_2 \in S$ with $s_1 \neq s_2$.
- 2. $SM : T_{\text{sub}} \rightarrow S$ is a **submodule function** that assigns a **submodule** to each substitution transition. It is required that the module hierarchy (see Definition 3) is acyclic.
- 3. PS is a **port-socket relation function** that assigns a **port-socket relation** $PS(t) \subseteq P_{\text{sock}}(t) \times P_{\text{port}}^{SM(t)}$ to each substitution transition t . It is

- required that $ST(t)(p) = PT(p')$, $C(p) = C(p')$, and $I(p)\langle\rangle = I(p')\langle\rangle$ for all $(p, p') \in PS(t)$ and all $t \in T_{sub}$.
4. $FS \subseteq 2^P$ is a set of non-empty and disjoint **fusion sets** such that $C(p) = C(p')$ and $I(p)\langle\rangle = I(p')\langle\rangle$ for all $p, p' \in fs$ and all $fs \in FS$.

The module hierarchy of a hierarchical CPN model is a directed graph with a node for each module and an arc leading from one module to another module if the latter module is a submodule of one of the substitution transitions of the former module. In the definition, T_{sub} denotes the union of all substitution transitions of the hierarchical CPN, and T_{sub}^s denotes all substitution transitions in a module s .

Definition 3. The **module hierarchy** for a hierarchical Coloured Petri Net $CPN_H = (S, SM, PS, FS)$ is a directed graph $MH = (N_{MH}, A_{MH})$, where

1. $N_{MH} = S$ is the set of **nodes**.
2. $A_{MH} = \{(s_1, t, s_2) \in N_{MH} \times T_{sub} \times N_{MH} \mid t \in T_{sub}^{s_1} \wedge s_2 = SM(t)\}$ is the set of **arcs**.

The roots of MH are called **prime modules**, and the set of all prime modules is denoted S_{PM} .

3 Pragmatic Annotated CPNs

PA-CPNs mandate a particular structure of the CPN models and allow the CPN elements to be annotated with *pragmatics* used to direct the automated code generation. In the CPN model, pragmatics are shown by annotations enclosed in $\langle\langle\rangle\rangle$. Pragmatics can also have some parameters, which we discuss as they come; but we do not formalize parameters of pragmatics in general here.

A PA-CPN is organised into three levels of modules: the *protocol system level*, the *principal level*, and the *service level* – reflecting the typical structure of protocols. In order to better understand the structure of PA-CPNs, Figs. 1 and 2 show selected modules from each level of a PA-CPN model of the protocol that we use as a running example. The protocol consists of a sender and a receiver principal, with services for sending and receiving data messages, and for sending and receiving acknowledgements. The sender sends each data message, one at a time, with a bounded number of retransmissions awaiting an acknowledgement for each data packet. In addition to the two principals, the protocol system contains unreliable channels for transmitting messages. The complete PA-CPN model of the example protocol is available at [17].

We formally define PA-CPNs as a tuple consisting of a hierarchical CPN: one protocol system module (PSM), sets of principal level modules (PLMs) and service level modules (SLMs) and channel modules (CHMs), and a structural pragmatics mapping (SP) that maps substitution transitions (indicated by double borders) to pragmatics representing the annotations of substitution transitions.

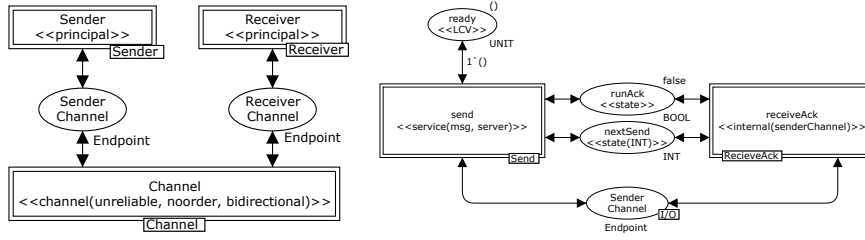


Fig. 1. The top-level CPN system level module (left) and principal level module for the sender principal (right) of the protocol example.

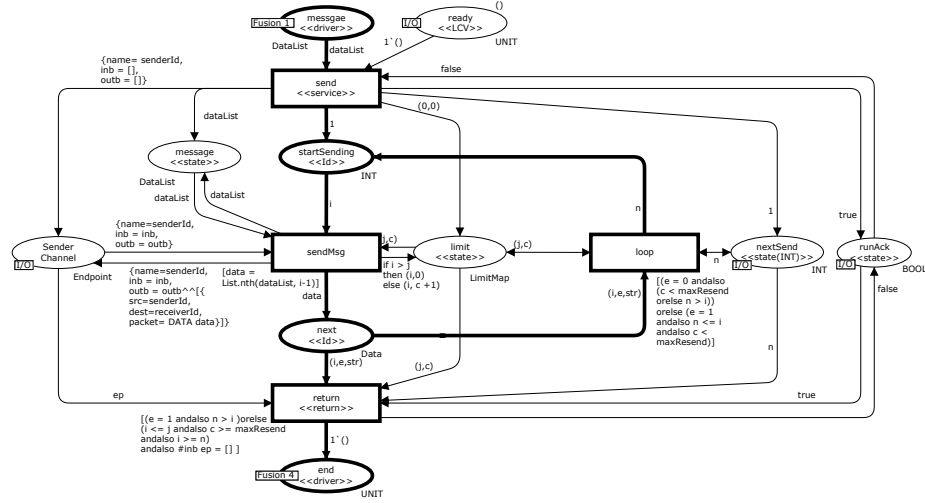


Fig. 2. The send service level module of the protocol example.

Definition 4. A *Pragmatics Annotated Coloured Petri Net (PA-CPN)* is a tuple $CPN_{PA} = (CPN_H, PSM, PLM, SLM, CHM, SP)$, where:

1. $CPN_H = (S, SM, PS, FS)$ is a hierarchical CPN with $PSM \in S$ being a **protocol system module** (Def. 5) and the only prime module of CPN_H .
2. $PLM \subseteq S$ is a set of **principal level modules** (Def. 6); $SLM \subseteq S$ is a set of **service level modules** (Def. 7) and $CHM \subseteq S$ is a set of **channel modules** s.t $\{\{PSM\}, PLM, SLM, CHM\}$ constitute a partitioning of S .
3. $SP : T_{sub} \rightarrow \{\text{principal}, \text{service}, \text{internal}, \text{channel}\}$ is a **structural pragmatics mapping** such that:
 - (a) Substitution transitions with $\langle\langle \text{principal} \rangle\rangle$ have an associated principal level module: $\forall t \in T_{sub} : SP(t) = \text{principal} \Rightarrow SM(t) \in PLM$.
 - (b) Substitution transitions with $\langle\langle \text{service} \rangle\rangle$ or $\langle\langle \text{internal} \rangle\rangle$ are associated with a service level module:

$$\forall t \in T_{sub} : SP(t) \in \{\text{service}, \text{internal}\} \Rightarrow SM(t) \in SLM.$$

- (c) *Substitution transitions with $\langle\langle\text{channel}\rangle\rangle$ are associated with a channel module: $\forall t \in T_{sub} : SP(t) = \text{channel} \Rightarrow SM(t) \in CHM$.*

It should be noted that channel modules do not play a role in the code generation; they constitute a CPN model artifact used to connect the principals for verification purposes. Therefore, we do not impose any specific requirements on the internal structure of channel modules.

Protocol system level. The module shown in Fig. 1(left) comprises the protocol system level of the PA-CPN model of the example. It specifies the two protocol principals in the system and the channel connecting them. The substitution transitions representing principals are specified using the `principal` pragmatic, and the substitution transitions representing channels are specified using the `channel` pragmatic. The PSM module is defined as a tuple consisting of a CPN module and a pragmatic mapping PM that associates a pragmatic to each substitution transition. The requirement on a protocol system module is that all substitution transitions must be substitution transitions that are annotated with either a `principal` or a `channel` pragmatic. Furthermore, two substitution transitions representing principals cannot be directly connected via a place: there must be a substitution transition representing a channel in between. This reflects the fact that principals can communicate via channels only.

Definition 5. A **Protocol System Module** of a PA-CPN with a structural pragmatics mapping SP is a tuple $CPN_{PSM} = (CPN^{PSM}, PM)$, where:

1. $CPN^{PSM} = ((P^{PSM}, T^{PSM}, A^{PSM}, \Sigma^{PSM}, V^{PSM}, C^{PSM}, G^{PSM}, E^{PSM}, I^{PSM}), T_{sub}^{PSM}, P_{port}^{PSM}, PT^{PSM})$ is a CPN module such that all transitions are substitution transitions: $T^{PSM} = T_{sub}^{PSM}$.
2. $PM : T_{sub}^{PSM} \rightarrow \{\text{principal}, \text{channel}\}$ is a **pragmatics mapping** s.t.:
 - (a) All substitution transitions are annotated with either a `principal` or `channel` pragmatic: $\forall t \in T_{sub}^{PSM} : PM(t) \in \{\text{principal}, \text{channel}\}$.
 - (b) The pragmatics mapping PM must coincide with the structural pragmatic mapping SP of PA-CPN: $\forall t \in T_{sub}^{PSM} : PM(t) = SP(t)$.
 - (c) All places are connected to at most one substitution transition with $\langle\langle\text{principal}\rangle\rangle$ and at most one substitution transition with $\langle\langle\text{channel}\rangle\rangle$:
 $\forall p \in P^{PSM} : \forall t_1, t_2 \in X(p) : PM(t_1) = PM(t_2) \Rightarrow t_1 = t_2$.

Principal level. On the principal level, there is one module for each principal of the protocol as defined by $\langle\langle\text{principal}\rangle\rangle$ on the protocol system level. The example protocol has two modules at the principal level corresponding to the sender and the receiver. Figure 1(right) shows the principal level module for the sender. A principal level module is required to model the *services* that the principal is providing, and the *internal states* and *life-cycle* of the principal. For the sender, there are two services as indicated by the `service` and `internal` pragmatics on the substitution transitions `send` (for sending messages) and `receiveAck` (for

receiving acknowledgements). Services that can be externally invoked are specified using the `service` pragmatic, whereas services that are to be invoked only internally are specified using the `internal` pragmatic. The non-port places of a principal level module (places drawn without a double border) can be annotated with either a `state` or an LCV pragmatic. Places annotated with a `state` pragmatic represent internal states of the principal. In Fig. 1(right), there are two places with $\langle\langle\text{state}\rangle\rangle$ used to enforce a stop-and-wait pattern in sending data messages and receiving acknowledgements. Places annotated with an LCV pragmatic represent the life-cycle of the principal by putting restrictions on the order in which services can be invoked. As an example, the place `ready` in Fig. 1(right) ensures that only one message at a time is sent using the `send` service.

Definition 6. A *Principal Level Module* of a PA-CPN is a tuple $CPN_{PLM} = (CPN_{PLM}, T_{sub}^{PLM}, P_{port}^{PLM}, PT^{PLM}, PLP)$ where:

1. $CPN_{PLM} = ((P^{PLM}, T^{PLM}, A^{PLM}, \Sigma^{PLM}, V^{PLM}, C^{PLM}, G^{PLM}, E^{PLM}, I^{PLM}), T_{sub}^{PLM}, P_{port}^{PLM}, PT^{PLM})$ is a CPN module with only substitution transitions: $T^{PLM} = T_{sub}^{PLM}$.
2. $PLP : T_{sub}^{PLM} \cup P_{port}^{PLM} \setminus P_{port}^{PLM} \rightarrow \{\text{service}, \text{internal}, \text{state}, \text{LCV}\}$ is a **principal level pragmatics mapping** satisfying:
 - (a) All non-port places are annotated with either a `state` or a LCV pragmatic: $\forall p \in P^{PLM} \setminus P_{port}^{PLM} \Rightarrow PLP(p) \in \{\text{state}, \text{LCV}\}$
 - (b) All substitution transitions are annotated with a `service` or `internal` pragmatic: $\forall t \in T_{sub}^{PLM} : PLP(t) \in \{\text{service}, \text{internal}\}$.

Service level. The service level modules specify the detailed behaviour of the individual services and constitute the lowest level modules in a PA-CPN model. In particular, there are no substitution transitions in modules at this level. The module in Fig. 2 is an example of a module at the service level. It models the behaviour of the `send` service in a control-flow oriented manner. The control-flow path, which defines the control flow of the service, is made explicit via the use of the `Id` pragmatics. The entry point of the service is indicated by annotating a single transition with $\langle\langle\text{service}\rangle\rangle$, and the exit (termination) point of the service is indicated by annotating a single transition with $\langle\langle\text{return}\rangle\rangle$. In addition, non-port places can be annotated with a `state` pragmatic to indicate that this place models a local state of the service. The driver pragmatic is used by service tester modules (Sect. 6) to facilitate verification. The places with $\langle\langle\text{Id}\rangle\rangle$ determine a subnet of the module, which we call the *underlying control-flow net*: it is obtained by removing all CPN inscriptions and considering only places with $\langle\langle\text{Id}\rangle\rangle$ and transitions connected to these places, which in Fig. 2, are indicated by places, transitions, and arcs with thick border. This control-flow net must follow a certain structure so that there is a one-to-one correspondence to control-flow constructs of typical programming languages. This requirement is called *tree decomposability* and is formally defined in Sect. 5.

A service level module is defined as consisting of a CPN module without substitution transitions and with service level pragmatics as described above.

Note that we use the symbol $\exists!$ to indicate that there “exists exactly on element” with the respective property.

Definition 7. A *Service Level Module* of a PA-CPN is a tuple $CPN_{SLM} = (CPN_{SLM}, T_{sub}^{SLM}, P_{port}^{SLM}, PT^{SLM}, SLP)$ where:

1. $CPN_{SLM} = ((P^{SLM}, T^{SLM}, A^{SLM}, \Sigma^{SLM}, V^{SLM}, C^{SLM}, G^{SLM}, E^{SLM}, I^{SLM}), T_{sub}^{SLM}, P_{port}^{SLM}, PT^{SLM})$ is a CPN module without substitution transitions: $T_{sub}^{SLM} = \emptyset$.
2. $SLP : T^{SLM} \cup P^{SLM} \setminus P_{port}^{SLM} \rightarrow \{Id, state, service, return, driver\}$ is a **service level pragmatic mapping** satisfying:
 - (a) Each place is either annotated with *Id*, *state*, *driver* or is a port place : $\forall p \in P^{SLM} \setminus P_{port}^{SLM} : SLP(p) \in \{Id, state, driver\}$.
 - (b) There exists exactly one transition with $\langle\langle service \rangle\rangle$ and exactly one transition with $\langle\langle return \rangle\rangle$:
 $\exists! t \in T^{SLM} : SLP(t) = service$ and $\exists! t \in T^{SLM} : SLP(t) = return$.
3. For all $t \in T^{SLM}$ and $p \in P^{SLM}$ we have:
 - (a) Transitions consume one token from input places with an *Id* pragmatic:
 $(p, t) \in A^{SLM} \wedge SLP(p) = Id \Rightarrow |E(p, t)(b)| = 1$ for all bindings b of t .
 - (b) Transitions produce one token on output places with an *Id* pragmatic:
 $(t, p) \in A^{SLM} \wedge SLP(p) = Id \Rightarrow |E(t, p)(b)| = 1$ for all bindings b of t .
 - (c) Only transitions with $\langle\langle service \rangle\rangle$ can have input places with $\langle\langle driver \rangle\rangle$:
 $(p, t) \in A^{SLM} \wedge SLP(p) = driver \Rightarrow SLP(t) = service$
 - (d) Only transitions with $\langle\langle return \rangle\rangle$ can have output places with $\langle\langle driver \rangle\rangle$ pragmatic: $(t, p) \in A^{SLM} \wedge SLP(p) = driver \Rightarrow SLP(t) = return$
4. The underlying control flow net of CPN_{SLM} is tree decomposable (Defs. 9,11).

4 Protocol Modelling Process

In the previous sections, we have formalised the structural restrictions of CPNs and the pragmatics extensions that make them *Pragmatic Annotated CPNs* (PA-CPNs); some additional restrictions on the control-flow structure and the service testers will be formalized later in Sect. 5 and 6. Since it is the modellers responsibility to come up with a model meeting these requirements, we briefly discuss the choices underlying the definition of PACPNs and their structural restrictions concerning the modelling process and some methodology for developing protocol software with PA-CPNs here.

The structural requirements of PA-CPNs have been distilled from the experience with earlier CPN models of protocols. The structure and annotations of PA-CPNs are designed to help the modeller come up with a clear model and to give clear guidelines for creating a model that – at the same time – can be used for code generation as well as for verification. As such, the structure of PA-CPNs should be driven by the protocol and its purpose rather than by the artifacts of Petri nets. This is, in particular, reflected by structuring the model in three layers: *protocol system*, *principal*, and *service layer*.

The top layer, the *protocol system layer*, identifies the overall structure of the protocol, which are the *principals* of the protocol and how the principals are connected by *channels* (see Fig. 1 (left) for an example). Each principal and each channel is represented by a substitution transition with a respective annotation, and places connecting the respective principals with channels. The behaviour of each principal is represented by *principal level module*, which identifies the *services* of the respective principal (see Fig. 1 (right) for an example) along with the states of the protocol and its life-cycle. The services are represented by substitution transitions annotated with the *service* pragmatics, the state and the life-cycle of the principal are represented by places with *state* and *LCV* pragmatics. The behaviour of each service is then modelled by a *service level module*, which is associated with the service substitution transitions on the *principal level module* (see Fig. 2 for an example). The service level module has access to the channels that the principal is connected to as well as to the principal's state and life-cycle variables. The most prominent structure (indicated by bold-faced places, transitions, and arcs) of the service level module is the control-flow structure, which is identified by the *Id* pragmatics and which needs to follow very specific rules so that it can be transformed to control-flow constructs of typical programming languages and result in human-readable code. The exact requirements are discussed in Sect. 5.

It should be noted that also the channels (on the protocol system level) need to be associated with PA-CPN modules, which model the exact behaviour of the respective channel. The modules for the channels are not used for code generation, since the generated code will use implementations of channels from the underlying platform (based on the properties required for these channels). But for verifying the protocol with standard CPN mechanisms, we need a CPN module for each channel, which however does not have any further structural restrictions.

Any model that meets the requirements of PA-CPNs can be used for code generation as well as for verification – irrespective of the way it was produced. The typical modelling process of protocols with PA-CPN starts at the top-level by identifying the principals of the protocol and how they are connected by channels. Then, the services of each principal are identified on the principal level, and then each service is modelled. So the general modelling direction is top-down. Of course, additional services and even additional principals could be added later, when need should be.

5 Tree Decomposability of Control Flow Nets

As discussed earlier, the control-flow structure of a service level module, called the underlying control-flow net, must correspond one-to-one to control-flow constructs of programming languages. The main purpose of this requirement is to generate readable code. In this section, we formally define the *underlying control flow net* of a service level module and its one-to-one correspondence to control-flow constructs. This is achieved by inductively decomposing the control-flow net

into a tree of sub-blocks, each of which corresponds to a control-flow construct: atomic step, sequence, choice and loop.

Figure 3 shows the *underlying control flow net* of the service level module from Fig. 2. All places and transitions in the rounded rectangle (representing the block border) are part of the block; an arrow from the block border to a place indicates the entry place; an arrow from a place to the block border indicates the exit place. The control flow net in Fig. 3 can be decomposed in a loop block, which in turn consists of an atomic block.

First, we define *blocks*: these are Petri nets with a fixed entry and exit place.

Definition 8. Let $N = (P, T, A)$ be a Petri net and $s, e \in P$. Then $B = (P, T, A, s, e)$ is called a **block** with **entry** s and **exit** e . The block is **atomic**, if $P = \{s, e\}$, $s \neq e$, $|T| = 1$ and for $t \in T$, we have $\bullet t = \{s\}$ and $t^\bullet = \{e\}$. The block has a **safe entry**, if $s \neq e$ and $\bullet s = \emptyset$. The block has a **safe exit**, if $s \neq e$ and $e^\bullet = \emptyset$.

For easing the following definitions, we introduce an additional notation: For a block B_i , we refer to its constituents by $B_i = (P_i, T_i, A_i, s_i, e_i)$ without explicitly naming them every time. The block that is underlying a service level module is determined by all the places with $\langle\langle \text{Id} \rangle\rangle$ pragmatics and the transitions in their pre- and postsets. The unique transition with $\langle\langle \text{service} \rangle\rangle$ defines the entry place, and the unique transition with $\langle\langle \text{return} \rangle\rangle$ defines the exit place of this block; note that for technical reasons, these two transitions are not part of the block. Therefore, these transitions are shown by dashed lines in Fig. 3. Formally, the control flow net underlying a service level module is defined as follows.

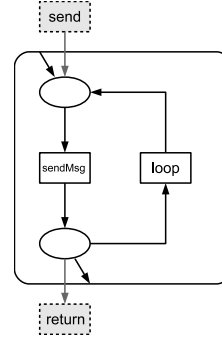


Fig. 3. Decomposition of the service level module in Fig. 2

Definition 9. Let CPN_{SLM} be a service level module as defined in Def. 7. Let $P = \{p \in P^{SLM} \setminus P_{port}^{SLM} \mid SLP(p) = Id\}$, let $T = T^{SLM} \cap \bullet P \cap P^\bullet$, and let $A = A^{SLM} \cap ((T \times P) \cup (P \times T))$; moreover, let $s \in P$ be the unique place such that there exists a transition $t \in T = T^{SLM}$ with $(t, s) \in A^{SLM}$ and $SLP(t) = \text{service}$, and let $e \in P$ be the unique place e such that there exists a transition $t \in T = T^{SLM}$ with $(e, t) \in A^{SLM}$ and $SLP(t) = \text{return}$. Then, $N = (P, T, A, s, e)$ is the **underlying control flow net** of CPN_{SLM} .

The control flow of the code that is being generated is obtained by decomposing the underlying control flow net of a service level module into sub-blocks representing the control-flow constructs. We define the decomposition in a very general way at first, which does not yet restrict the possible control-flow constructs. The decomposition into blocks, just makes sure that all parts of the block are covered by sub-blocks and that they overlap on entry and exit places

only. In a second step, the decomposition is restricted in such a way that the decomposition captures certain control flow constructs (Def. 11).

Definition 10. Let $B = (N, s, e)$ be a block with net $N = (P, T, F)$. A set of blocks B_1, \dots, B_n is a **decomposition** of B if the following conditions hold:

1. The sub-blocks contain only elements from B , i. e. for each $i \in \{1, \dots, n\}$, we have $P_i \subseteq P$, $T_i \subseteq T$, and $F_i \subseteq F \cap ((P_i \times T_i) \cup (T_i \times P_i))$.
2. The sub-blocks contain all elements of B , i. e. $P = \bigcup_{i=1}^n P_i$, $T = \bigcup_{i=1}^n T_i$, and $F = \bigcup_{i=1}^n F_i$.
3. The inner structure of all sub-blocks are disjoint, i. e. for each $i, j \in \{1, \dots, n\}$ with $i \neq j$, we have $T_i \cap T_j = \emptyset$ and $P_i \cap P_j = \{s_i, e_i\} \cap \{s_j, e_j\}$.

As the final step, we define when a decomposition of a block reflects some control flow construct. The definition does not only define decomposability into control flow constructs; it also defines a tree structure which reflects the control-flow structure of the block; the type of each node reflects the construct. The definition is illustrated in Fig. 4. The top left part of Fig. 4 shows the inductive definition of a loop construct: The assumptions are that two blocks $B1$ and $B2$ are identified already. $B1$ is any kind of block (represented by X) with a safe entry place s and a safe exit place e ; $B2$ is an atomic block with entry place e and exit place s . Thus, block $B1$ represents the loop body, and block $B2$ the iteration. Then, the union of both blocks and entry place s and exit place e , form a block B , which is a loop consisting of the loop body $B1$ and the atomic block $B2$ for the iteration. The definitions of choices and sequences are similar.

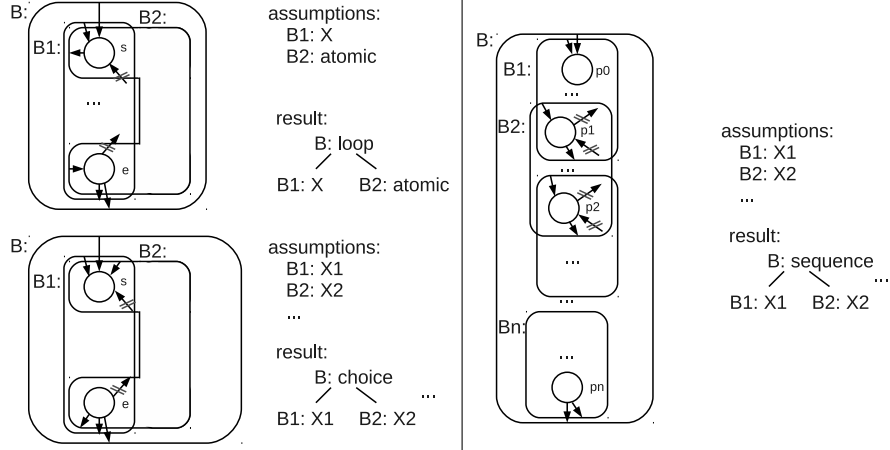


Fig. 4. Inductive definition of block trees

Definition 11 below formally defines block tree as illustrated in Fig. 4.

Definition 11. The **block trees** associated with a block are inductively defined:

- Atomic** If B is an atomic block, then the tree with the single node **B :atomic** is a **block tree** associated with B .
- Loop** If B is a block and B_1 and B_2 is a decomposition of B , and for some X , $B_1 : X$ is a block tree associated with B_1 , and $B_2 : \text{atomic}$ is a block tree associated with B_2 , and if B_1 has a safe entry and a safe exit s.t $s_1 = s$, $e_1 = e$, $s_2 = e$, $e_2 = s$, then the tree with top node **B :loop** and the sequence of sub-trees $B_1 : X$ and $B_2 : \text{atomic}$ is a **block tree** associated with B .
- Choice** If B is a block and for some n with $n \geq 2$ the set of blocks B_1, \dots, B_n is a decomposition of B , and have a safe entry and a safe exit, and $B_1 : X_1, \dots, B_n : X_n$ for some X_1, \dots, X_n are block trees associated with B_1, \dots, B_n , and if for all $i \in \{1, \dots, n\}$: $s_i = s$ and $e_i = e$, then the tree with top node **B :choice** with the sequence of sub-trees $B_i : X_i$ is a **block tree** associated with B .
- Sequence** If B is a block and for some n with $n \geq 2$ the set of blocks B_1, \dots, B_n is a decomposition of B , and, for some X_1, \dots, X_n , the trees $B_1 : X_1, \dots, B_n : X_n$ are block trees associated with B_1, \dots, B_n , and if there exist different places $p_0, \dots, p_n \in P$ such that $s = p_0$, $e = p_n$, and for each $i \in \{0, \dots, n-1\}$ we have $s_i = p_i$, $e_i = p_{i+1}$, and B_i has a safe exit or B_{i+1} has a safe entry, then the tree with top node **B :sequence** and the sequence of sub-trees $B_i : X_i$ is a **block tree** associated with B .

A net for which such a tree exists is said to be **tree decomposable**.

Note that in order to simplify the definition of tree decomposability, the tree decomposition of a block is not necessarily unique according to our definition. For example, a longer sequence of atomic blocks could be decomposed in different ways. In the PetriCode tool, such ambiguities are resolved by making sequences as large as possible. Note also that for two consecutive constructs in a sequence, it should not be possible to go back from the second to the first; therefore, the above definition requires that consecutive blocks have a safe entry or a safe exit. And there are some similar requirements for loops and choices.

6 Service Testers and Sweep-Line Verification

The service level modules constitute the active part of a PA-CPN model. The execution of an individual service provided by a principal starts at the transition with a $\langle\langle \text{service} \rangle\rangle$ pragmatic. The transitions annotated with a service pragmatic typically has a number of parameters which need to be bound to values in order for the transition to occur. An example of this is the **Send** service transition in Fig. 2 which has the variable **dataList** as a parameter. This means that there are often an infinite number of bindings for a service transition.

To control the execution of a PA-CPN model in verification by means of state space exploration, we introduce the concept of *service tester modules* which can be used to guide the verification process and represent a user of the services provided by the principal modules. An advantage of service testers is that they

contribute to reducing the state space during verification and enable progress measures for the sweep-line method [4] to be automatically computed.

The service tester modules are connected to the rest of the PA-CPN model through fusion sets, and the service tester modules invoke the service provided by the principal by putting tokens on fusion places and the service tester receives any results from the invoked services via tokens on these places.

Fusion sets and fusion places are standard constructs of hierarchical CPNs (see Def. 2). A fusion set consists of a set of fusion places such that removing (adding) tokens from (to) a fusion place is reflected on the markings of all members of the fusion set. In addition to the fusion places, *Id* pragmatics are used to make the control flow of the service tester explicit in a similar manner as for service level modules.

Figure 5 shows an example of a service tester module for the PA-CPN model introduced in Sect. 3. The service tester drives the execution of a CPN model through fusion places. A service tester module can have many places with *Id* pragmatics; but only one of them may contain a token initially (place *d0* in Fig. 5). The service tester first invokes the send service in Fig. 2 by putting a token in the fusion place *message*. Next, the service tester invokes the receive service in the receiver principal. Service tester modules are formalised below.

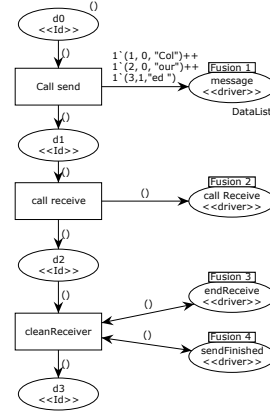


Fig. 5. Service tester module

Definition 12. A *Service Tester Module* is a tuple $CPN_{STM} = (CPN_{STM}, T_{sub}^{STM}, P_{port}^{STM}, PT^{STM}, TPM)$ where:

1. $CPN_{STM} = ((P^{STM}, T^{STM}, A^{STM}, \Sigma^{STM}, V^{STM}, C^{STM}, G^{STM}, E^{STM}, I^{STM}), T_{sub}^{STM}, P_{port}^{STM}, PT^{STM})$ is a CPN module with no substitution transitions: $T_{sub}^{STM} = \emptyset$.
2. $TPM : P^{STM} \rightarrow \{Id, driver, LCV\}$ is a **tester pragmatic mapping**.
3. $\exists! p \in I : |I^{STM}(p)\langle \rangle| = 1$, and for all $t \in T^{STM}$ and $p \in P^{STM}$ we have:
 - (a) Transitions consume one token from input places with an *Id* pragmatic: $(p, t) \in A^{STM} \wedge TPM(p) = Id \Rightarrow |E(p, t)\langle b \rangle| = 1$ for all bindings b of t .
 - (b) Transitions produce one token on output places with an *Id* pragmatic: $(t, p) \in A^{STM} \wedge TPM(p) = Id \Rightarrow |E(t, p)\langle b \rangle| = 1$ for all bindings b of t .
4. Transitions and places with an *LCV* pragmatic must be connected with a double arc: $\forall p \in P^{STM}, t \in T^{STM} : TPM(p) = LCV \Rightarrow ((t, p) \in A^{STM} \Leftrightarrow (p, t) \in A^{STM})$
5. The underlying control flow block of CPN_{STM} is tree decomposable (Defs. 9,11).

Service tester modules are connected to a PA-CPN by means of fusion places in order to control the execution of the services. We therefore define a PA-CPN

equipped with service tester modules as a hierarchical CPN consisting of a set of modules that constitute a PA-CPN according to Def. 4 and a set of service tester modules which are all prime modules. We also require that fusion places are connecting the service level modules and the service tester module so that they correspond to the invocation of services and collecting of a results from an executed service. As with PA-CPNs, the modeller must construct the service tester modules such that they satisfy the formal requirements. Due to space limitations we omit the formal definition of PA-CPNs with service testers which can be found as Def. 5.2 in [20].

The set of service tester modules determine the state space of the PA-CPN model under analysis. The service tester modules may specify a more or less strict execution order on the services being invoked. It is therefore possible to use the service tester modules to control the size of the state space of the PA-CPN model being verified. Below we show that in addition to the use of service testers, the structural requirements imposed by PA-CPNs can be exploited by the sweep-line method [4] to further reduce the peak memory usage during verification.

The sweep-line method addresses the state explosion problem by exploiting a notion of *progress* exhibited by many systems to store subsets of the state space in memory during state space exploration. To apply the sweep-line method, a *progress measure* must be provided for the model as formalised below where \mathcal{S} denotes the set of all states (markings), \rightarrow^* the reachability relation on the markings of the CPN model, and $\mathcal{R}(M_0)$ denotes the states reachable from the initial marking M_0 .

Definition 13. A *progress measure* is a tuple $\mathcal{P} = (O, \sqsubseteq, \psi)$ such that O is a set of *progress values*, \sqsubseteq is a total order on O , and $\psi : \mathcal{S} \rightarrow O$ is a *progress mapping*. \mathcal{P} is *monotonic* if $\forall s, s' \in \mathcal{R}(M_0) : s \rightarrow^* s' \Rightarrow \psi(s) \sqsubseteq \psi(s')$. Otherwise, \mathcal{P} is *non-monotonic*.

The subsets of states that need to be stored at the same time are determined via a *progress value* assigned to each state, and the method explores the states in a least-progress-first order. The sweep-line method explores states with a given progress value before progressing to the states with a higher progress value. When the method proceeds to consider states with a higher progress value, it deletes the states with a lower progress value from memory. If it turns out that the system regresses (a non-monotonic progress measure), then the method will mark states at the end of *regress edges* as *persistent* (i. e., store them permanently in memory) in order to ensure termination. In the presence of regression, the sweep-line method may visit the same state multiple times (for details, see [4]).

The structure imposed on CPNs by PA-CPNs and services testers means that PA-CPN models have several potential sources of progress. The control-flow in the service modules is one source of progress as there is a natural progression from the entry point of the service towards the exit point of the service. The life-cycle of a principal is another potential source of progress as there will often be an overall intended order in which the services provided are to be invoked,

and this will be reflected in the life-cycle variables of the principal. Finally, the service testers are also a source of progress as a service tester will inherently progress from the start of the test towards the end of the test. For our example protocol, the progress mapping can be defined as a vector of place-wise measures using the number of tokens on some of its places. This is written below where we omitted the parts of the model that we did not show in this paper and used $s(p)$ to denote the marking of a place p in the state s :

$$\psi(s) = (|s(d0)|, |s(d1)|, |s(d2)|, |s(d3)|, |s(startSnd)| + |s(next)|, |s(end)|) \quad (1)$$

Two such vectors can be compared lexicographically, meaning the order of the different entries represents their significance. The first four entries represent the progress in the service tester (Fig. 5). The next two entries represent the progress within the `send` service (Fig. 2). Note that since the places `startSending` (abbreviated as `startSnd` in (1) and (2)) and `next` are on a loop, tokens can flow back from place `next` to place `startSending`. The `end` place is actually the respective driver place from the tester, which propagates the progress between the service and tester. Therefore, the tokens on both places within this loop are counted the same (added up in the same entry of the vector). An alternative progress measure is shown below (omitting the parts of the model that we did not show in this paper):

$$\psi(s) = (|s(d0)|, |s(d1)|, |s(d2)|, |s(d3)|, |s(startSnd)|, |s(next)|, |s(end)|) \quad (2)$$

The difference between (1) and (2) is how loops are handled. In the progress measure (2), the places on loops are appended to the vector as if the loop was not there. In the present example this is shown by having replaced the $+$ operator in (1) between `startSending` and `next` with a comma in (2).

We generalise the above idea by defining progress measures on top of the tree decomposition of the blocks underlying the corresponding service tester module or the service level module. We define a simple progress measure and a complex one. The simple one is monotonic and adds up the number of all tokens within a top-level loop; the complex one is not monotonic, but takes progress within a loop into account. Since both definitions are very similar, we define only the complex progress measures formally here (the simple one can be found in [20]).

Definition 14. *Let BT be a block tree for a CPN module. The sequence of **complex progress measure entries** is defined inductively over the block tree BT of the CPN module:*

Atomic *If BT is B : atomic, then complex progress sequence consist of $|s|, |e|$ where s is the entry place of the block B and e is the exit place.*

Sequence *If BT is B : sequence with subblocks B_1, \dots, B_n , and $e_i^1, \dots, e_i^{k_i}$ are the complex progress sequences for B_i , then $e_1^1, \dots, e_1^{k_1-1}, e_2^1, \dots, e_2^{k_2-1}, \dots, e_n^1, \dots, e_n^{k_n}$ is the complex sequence for BT .*

Choice *If BT is B : choice with subblocks B_1, \dots, B_n , and $e_i^1, \dots, e_i^{k_i}$ are the complex progress sequence for each block B_i , then the sequence $e_1^1, \dots, e_1^{k_1-1}, e_2^1, \dots, e_2^{k_2-1}, \dots, e_n^1, \dots, e_n^{k_n}$ is the complex progress sequence for BT .*

Loop *If BT is $B : \text{loop}$ with places with sub-block B_1 and B_2 with the complex progress sequence e^1, \dots, e^n for B_1 , then e^1, \dots, e^n is the complex progress sequence for BT .*

A progress measure for the complete system can be built from the progress sequences (either the simple or the complex one) for the tester and service modules by concatenating the sequences. The concatenation would first choose the sequences for the service testers and then the sequences for all the service level modules. Note that if there is a driver place of a service tester attached to the service, this driver place would also be added to the progress measure sequence of the service level module at the end (as for the `end` place for the `send` service).

Table 1 shows some experimental performance results on the protocol example for different configurations (number of transmitted messages) and channel characteristics (lossy/non-lossy) using the sweep-line method with the simple and the complex progress measure. In the experiments, we consider exploration of the complete state space. This is done since the sweep-line method (unless combined with other reduction techniques) in the worst-case needs to explore all states in order to model check a property. One example of this is checking that in all terminal states, the protocol has correctly delivered all packets. Since the simple progress measure is monotonic, the number of explored states using that measure is identical to the number reachable states of the respective example, which for clarity are indicated in the first column again. Since the complex progress measure is not monotonic, some states might be visited (explored) multiple times. Therefore, the number of explored states is higher than the reachable states of the respective example. The ratio columns give the ratio in percent between the peak number of states stored (with the respective progress measure) and the number of reachable states. It can be seen that the runtime as well the peak memory use are better when using the complex progress measure. The complex measure provides better performance due to the fact that the `send` service has a loop as the top-level control-flow construct. It can be seen that the peak memory use with the complex progress measure is reduced to between 40 and 77%.

Table 1. Verification using simple and complex progress measure

Config	Simple PM					Complex PM			
	Reachable	Explored	Peak	Ratio	Time	Explored	Peak	Ratio	Time
1:noloss	156	156	77	49.3	<1 s	165	63	40.3	<1 s
1:lossy	186	186	99	53.2	<1 s	196	78	41.9	<1 s
3:noloss	2,222	2,222	2,014	90.6	<1 s	2790	1,582	71.2	<1 s
3:lossy	2,928	2,928	2,700	92.2	<1 s	4037	2,187	75.7	<1 s
7:noloss	117,584	117,584	115,373	98.1	216 s	143,531	86,636	73.6	32 s
7:lossy	160,620	160,620	158,888	98.1	532 s	263,608	124,661	77.6	80 s

7 Conclusions and Related Work

In this paper, we focused on the formal definition of PA-CPNs and how the structure of PA-CPNs can be exploited for more efficient verification. The PA-CPN net class has been motivated by the objective of developing a code generation approach to protocol software which allows the same model to be used for both code generation and verification – and which satisfies five main requirements: *platform independence*, *code integration*, *verifiability*, *readability*, and *scalability*. The development of PA-CPNs has been driven by practical experiments in order to empirically validate that the approach satisfied the five requirement in practice. The experimental results have been reported in earlier papers [16, 17, 19] using an implementation of the approach in the PetriCode tool.

The requirements of *platform independence*, *code integration*, *readability*, and *scalability* are relevant for use in practice: Platform independence ensures that the approach is not locked to a particular target programming language. Code integration ensures that the generated code can be integrated with existing other parts of the software. *Readability* of the generated code is important for developing trust in the approach, and for further maintaining the protocol software in the future. *Scalability* is important for being able to apply the approach to industrial strength protocols. Concerning *verifiability*, it often is the case that one model is used for verification, and then the protocol is implemented manually from that or generated from another model. This imposes extra work and decreases the confidence in that the actual software meets the requirements verified on the model. Therefore, we required the same model being used for code generation and for verification.

As stated in the introduction, CPNs have been primarily used for modelling and verifying protocols in the past. Still, related approaches for CPNs – and more generally for high-level Petri Nets (HLPNs) – have been developed. Below, we relate our work to other approaches using HPLNs for code generation by discussing them in the context of the five requirements that have driven the development of PA-CPNs.

Kaim [6] contains a generic discussion of aspects related to generating code from low-level and high-level Petri net models with the purpose of executing it outside the simulation environment where they are created. Kaim discusses both centralised and parallel approaches to interpretation of Petri net models. A main aspect of the parallel approach is a structural analysis of the model in order to identify subnets that can be mapped to different processes. In the PetriCode approach, the structural pragmatics provided by the modeller and the structural restrictions of PA-CPNs provide similar information. Kaim does not consider the issues of code integration and the readability of the generated code.

The approach presented by Philippi [14] is a hybrid of simulation-based and structural analysis approaches to code generation for HLPNs. The motivation for the hybrid approach is to produce more readable code than a pure simulation approach would because fewer checks are needed in the code. Philippi targets the Java platform only and is therefore not platform independent in its basic form. The generated code can be integrated into third party code in that the

API of the generated code is defined by UML class diagrams. The paper [14] does not discuss the scaling to large applications. Lassen et al. [11] aim to generate readable code by creating code with constructs that are similar to what human programmers would have created. Since the approach of Lassen is based on Java annotations of CPN models, the approach is tailored to the Java programming language and does not provide a generic infrastructure that supports code generation for different platforms.

Reinke [15] studies, in the context of the functional programming language Haskell, how to use language embedding for mapping constructs from HLPNs into Haskell code. The focus of Reinke is on generating code for a HLPN simulator. The work of Reinke is not aimed at providing a general mechanism for generating readable code and on integrating the code into a larger application. Kummer et al. [10] are concerned with the execution of reference nets in the context of the Renew tool which is based on the Java platform. Reference nets as supported by Renew are known to be verifiable [12] but the approach is specifically tailored to the Java platform. The work does not focus on integration at the code level but other means are providing for integrating the code into larger applications [1].

Mortensen's approach [13] is a simulation based approach based on extracting the generated simulation code from CPN Tools. As such the work of Mortensen is aimed at making an SML implementation of the modelled system and not on conducting verification of the models or to target multiple platforms. Furthermore, being a simulation based approach, the goal from the outset is not to generate code that is intended for humans to read. The use of a simulation-based approach also means that there is a considerable performance overhead due to the many enabling checks in the code. The approach of Kristensen et al. [7] is similar to the approach in [13]. PP-CPNs are used in [9] as the basis for code generation targeting the Erlang language but the approach is not designed to address readability of the generated code. Furthermore, the approach is tailored to the Erlang platform and may not be easily adapted to other platforms even though PP-CPNs and the intermediary representation of control-flow graphs are independent of the target language. Jørgensen et al. [5] propose an approach for generating BPEL code. The approach is targeted at BPEL and does not create code for other languages or aims to address verifiability, code integration, readability and scalability.

It follows from the discussion above that PA-CPNs and the PetriCode approach complement existing related approaches to code generation for high-level Petri Nets. Furthermore, none of the approach discussed above specifically address the domain of protocol software. This paper can be viewed as completing the development of the PA-CPN net class by giving a formal definition and hence establishing the formal foundation of our approach.

References

1. T. Betz et al. Integrating web services in Petri net-based agent applications. In *Proc. of PNSE'13*, pages 97–116, 2013.

2. 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 *LNCS*, pages 210–290. Springer, 2004.
3. K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
4. K. Jensen, L.M. Kristensen, and T. Mailund. The Sweep-line State Space Exploration Method. *Theoretical Computer Science*, 429:169–179, 2012.
5. J. B. Jørgensen and K. B. Lassen. Let's Go All the Way: From Requirements via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In *Proc. of CoopIS'05*, volume 3760 of *LNCS*, pages 22–39. Springer, 2005.
6. W. El Kaim and F. Kordon. Code generation. In C. Girault and R. Valk, editors, *Petri Nets for System Engineering*, chapter 21, pages 433–470. Springer, 2003.
7. L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *International Journal on Software Tools for Technology Transfer*, 10:5–14, 2008.
8. 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.
9. L.M. Kristensen and M. Westergaard. Automatic Structure-Based Code Generation from Coloured Petri Nets: A Proof of Concept. In *Proc. of FMICS'10*, volume 6371 of *LNCS*, pages 215–230. Springer, 2010.
10. 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.
11. K. B. Lassen and S. Tjell. Translating Colored Control Flow Nets into Readable Java via Annotated Java Workflow Nets. In *Proc. of 8th CPN Workshop*, 2007.
12. M. Mascheroni, T. Wagner, and L. Wüstenberg. Verifying Reference Nets by Means of Hypernets: A Plugin for Renew. In *Proc. of the PNSE'19*, Berichte des Fachbereichs Informatik, pages 39–54. Universität Hamburg, 2010.
13. K. H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ICATPN'00*, volume 1825 of *LNCS*, pages 367–386, 2000.
14. S. Philippi. Automatic code generation from high-level Petri-Nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444 – 1455, 2006.
15. C. Reinke. Haskell-coloured petri nets. In *Int. Workshop on Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 165–180, 1999.
16. 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.
17. K.I.F. Simonsen. PetriCode: A Tool for Template-based Code Generation from CPN Models. In *Proc. of WS-FMDS 2013*, volume 8368 of *LNCS*, pages 151–163, 2013. Project website: <http://www.petricode.org>.
18. K.I.F. Simonsen. An Evaluation of Automated Code Generation with the PetriCode Approach. In *In Proc. of PNSE '14*, volume 1160 of *CEUR Workshop Proceedings*, pages 295–312. CEUR-WS.org, 2014.
19. K.I.F. Simonsen and L.M. Kristensen. Implementing the WebSocket Protocol based on Formal Modelling and Automated Code Generation. In *Proc. of IFIP DAIS'2014*, volume 8460 of *LNCS*, pages 104–118. Springer, 2014.
20. K.I.F. Simonsen, L.M. Kristensen, and E. Kindler. Pragmatics Annotated Coloured Petri Nets for Protocol Software Generation and Verification. Technical Report 16, DTU Compute, <http://goo.gl/9j6lDz>, 2014.

Providing Petri Net-Based Semantics in Model Driven-Development for the RENEW Meta-Modeling Framework

David Mosteller, Lawrence Cabac, Michael Haustermann

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics

<http://www.informatik.uni-hamburg.de/TGI>

Abstract. This paper presents an approach to the development of modeling languages and automated generation of specific modeling tools based on meta-models. Modeling is one of the main tasks in engineering. Graphical modeling helps the engineer not only to understand the system but also to communicate with engineers and with other stakeholders that participate in the development (or analytic) process.

In order to be able to provide adequately adapted modeling techniques for a given domain, it is useful to allow to develop techniques, which are designed for their special purpose, i.e. domain-specific modeling languages (DSML). For this cause meta-modeling comes in handy. Meta-models provide a clear abstract syntax and model-driven design approaches allow for rapid prototyping of modeling languages. However, often the transformation and also the original (source model) as well as the transformed (target) model do not provide a clear semantics.

We present an approach to model-driven development that is based on Petri nets: high- or low-level Petri nets in various formalisms can be used as target models. Starting from the conceptual background and underlying thinking tool, following up with code templates, transformation engines, underlying semantics and the way our process support is implemented up to the final target engine Petri nets and Petri net tools can be used.

Keywords: RENEW, Petri nets, model-driven development, meta-modeling

1 Introduction

Meta-modeling enables us to build models in a more abstract way than we are used to today. For many purposes we prefer languages that solve a specific modeling quest. While there are several well established modeling techniques with a clear semantics, the purpose of the incorporated languages is more or less fixed. Annotations like those in UML can in combination with profiles enhance the expressiveness. However, it is difficult to build lean languages that cover exactly those domain aspects that are required in a certain context. In addition, normally there exist no tools that directly support those languages with

specific language constructs. To make a language easy to use, one usually needs direct tool support. The development of tools for building graphical models was a challenge some years ago. Nowadays it is relatively easy within environments like Eclipse and its meta-modeling plugins.¹ Even extensions that allow a simulation of models built with those languages are available. However, usually these execution environments are relatively restricted and do not scale. This is due to the fact that the execution engine has to be built separately.

The development of a DSML and a corresponding modeling tool includes a whole range of tasks. We will address in this contribution: (1) providing the possibility to define an abstract syntax to allow users to build a special purpose language, (2) providing a graphical environment to allow users to build special language constructs for their specific language concepts (based on textual and graphical representations), (3) providing a tool set that allows to build models based on the previously defined languages and (4) providing a simulation environment (especially based on reference nets [10]) that allows users to execute and simulate their models. The presented approach to developing modeling languages and tools (RMT approach) is extensively applied within our approach to developing agent-oriented software based on Petri nets (P*AOSE approach, [3,13,17]), in which the mutual interplay of modeling languages is omnipresent. It is however equally applicable to other domains. We provide a prototype, which offers the possibility to develop modeling languages and to generate corresponding modeling tools. The RENEW Meta-Modeling Framework (RMT framework)² was applied in several settings. The RMT framework constitutes a further development step of the model-driven approach, which has been already envisioned and partly applied during the development of the Agent Role Modeler (ARM, [4]). The ARM tool, which was developed without appropriate meta-modeling tool support, provides the modeling facility for agent organizations and knowledge bases.

The remainder of this paper is structured as follows: The conceptual background, which comprises the model-driven tool development, encompassing meta-modeling, graphical modeling, transformations and semantical issues, is discussed with respect to the requirements and specification of our solution in Section 2. The example presented in Section 3 demonstrates the approach and the applications of Petri nets as well as the application of other techniques during DSML development. Section 4 elaborates on the wider context of model-driven development and the approach of providing transformational semantics for modeling languages with Petri nets. In Section 5 we will summarize our results and will give an outlook of our further research directions opened by these results.

¹ Eclipse Modeling Framework, EMF, <https://www.eclipse.org/modeling/emf/>

² RMT: RENEW Meta-Modeling and Transformation Framework, tools and examples: <http://www.paose.net/wiki/Metamodeling>

2 Conceptual Approach

As our approach to software development is based on the model-driven construction of software systems our aim is to provide a tool chain using model-driven techniques. We want to support the agile development of graphical modeling languages. Therefore, we rely on the concepts of software language engineering [9] and apply model-driven techniques to generate tools from abstract models. In the following we elaborate on the techniques required to realize a framework based on generating modeling tools. RENEW provides the basis for our meta-modeling framework. It serves as a graphical framework for the flexible construction of graphical models and at the same time provides the execution and simulation environment of Petri net models, which serve as target languages that provide the transformational semantics for the designed languages. This approach allows for the analysis of Petri net models and for the validation of model properties. Our conceptual approach is based on the idea of bootstrapping the required modeling tools using model-driven techniques. Following the concepts of software language engineering the development of modeling languages encompasses three aspects: abstract syntax, concrete syntax and semantics. Translating these concepts into the area of generative tool development leads to a set of descriptions defining the different aspects of software languages [15]: structure, constraints, representation and behavior. The structure (abstract syntax) and the representation (concrete syntax) of modeling languages will be addressed in the following section. The behavior (semantics) is covered in Section 2.2.

2.1 Meta-Modeling and Tool Generation

In this section we elaborate on the first part of the DSML development process. First we need to define the syntax of the new language (or technique). The abstract syntax of a language is specified by a meta-model, which defines the structure of the language. Our tool set, which is based on RENEW, supports the modeling of the abstract syntax directly through the technique of Concept Diagrams (cf. [3, Chapter 12]). Concept Diagrams are simplified Class Diagrams, which are usually used to design type hierarchies or agent ontologies (in the context of P*AOSE). In the context of this work the type hierarchies of Concept Diagrams are utilized to model the meta-models of the designed DSML, i.e. the abstract syntax.

Additionally, in order to define the representation of the elements we also need to define the concrete syntax. The concrete syntax, i.e. the representation of the syntactic elements, is defined through a mapping from the syntactic element to its graphical representation (representational mapping). The representational mapping includes concrete graphical or textual syntax as well as serialization representations. Typically, if the language should not be restricted to graphical standard figures, the layout, concrete form, etc. has to be defined in a form close to the implementation language. However, we provide also the possibility that the syntactic elements may be defined directly within the RENEW environment using the graphical user interface. Each provided graphical representation is

stored as one template drawing and the representational mapping refers not to an implementation but to a template (graphical component). Alternatively some standard elements are provided, which can be configured in terms of stylesheets to define the representation for the language constructs.

In addition to the abstract and the concrete syntax, we need to configure the user interface of the modeling tool that provides the modeling facility – in the RENEW environment the modeling tool is integrated as a plugin. The configuration is done defining another mapping for task bar tool buttons and their design together with some general information about the modeling tool, such as the file extension or the ordering of tool buttons in the task bar.

The RENEW plugins that provide the modeling facility for the stylesheets and the tool configurations are themselves meta-model based. They have been generated – in a bootstrapping fashion – using the RMT approach.

Figure 1 shows the defining artifacts of a modeling language's syntax in the top. These artifacts are expressed within the scope of the meta-meta-model – the RMT meta-model – and can thus be used to generate a domain-specific modeling tool, which then provides the possibility to design a model, using the technique; e.g. a (domain-specific) modeling language.

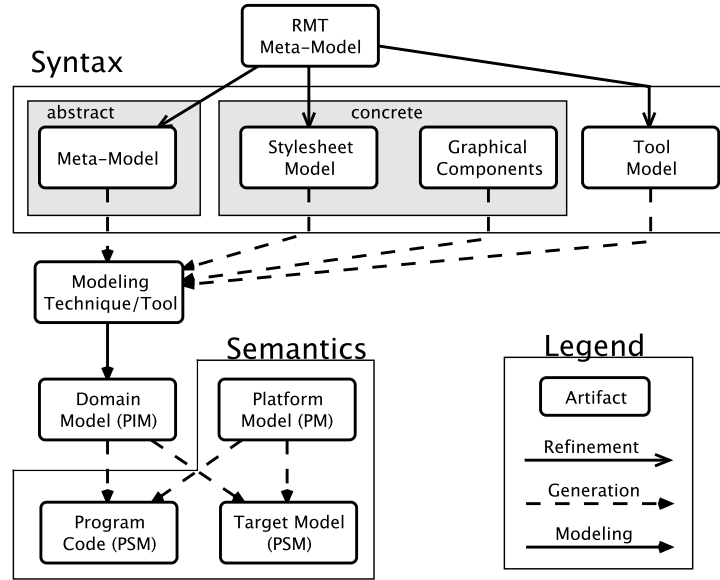


Figure 1. An abstract view on the models of a meta-modeling project.

A modeler may use the generated tool to model, store and retrieve graphical models (diagrams) in the syntax of the newly developed modeling language.³ For

³ In the following we will address these models as *domain model* or *source models*.

operational or analytic models, however, it is not enough to be able to provide graphical descriptions of the models. In these cases we need to define a clear semantics. Following the idea of the model-driven architecture (MDA) the semantic interpretation of a source model can be defined through a transformation into specific target models using a generator as shown in the lowermost part of Figure 1, which references the schematic view of Petrasch et. al. [18, p. 107].

We elaborate on this in the following section. But before we present the approach to the definition of the semantics, we stress the flexibility of the given approach, so far. The meta-modeling approach in itself offers a high degree of flexibility. By changing (augmenting, modifying or restricting) the meta-model we are able to quickly produce variations of modeling techniques, which may subsequently be compared with each other (see for instance Section 3.2). Additionally, we are able to change the representation of the modeling language by either changing the representational mapping or by editing the graphical components. Especially the latter can be done by someone without knowledge in the development details and thus create his own representation.

2.2 Transforming Source Models to Target Models

The semantics of a modeling language is defined – as semantic mapping, cf. [6] – either through formalization, through an operationalization or through the transformation into other models that already own a formal or an operational semantics. As we use the RENEW environment as basis for our approach, we transform given source models to Petri net models, i.e. our *target languages* are Petri nets formalisms. The RMT approach is not restricted to behavioral modeling languages. By choosing the proper target languages, such as Reference Nets, modeling structural properties can be performed by applying the same approach. In Figures 1 and 2 we can identify the domain specific model (source model), which is transformed into a target model (Platform Specific Model, PSM) within the application domain layer (M1). The RENEW environment together with its provided Petri net formalisms serve as Platform Model (PM, compare with Figure 1). In the context of model-driven development the source model is often described as Platform Independent Model (PIM).

The transformation process is depicted in Figure 2 as a schematic Petri net. Transitions represent actions provided by either the RMT tool set (generation, transformation, execution or analysis) or by the source model developer (modeling). Necessary artifacts for the development of the modeling language workflow are provided by the language developer using the RMT framework. These artifacts comprise the syntax meta-models, the transformer and the semantic elements provided as net components [3, Chapter 5]. Net components are Petri net snippets that are used as patterns to be mapped by a generator and combined to constitute the target models. In this sense the model transformation process can be characterized as a pattern-oriented transformation following the categorization of Petrasch et. al. [18, p. 132].

Besides supporting the agile development of graphical languages the RMT approach also provides a high level of flexibility regarding the semantic trans-

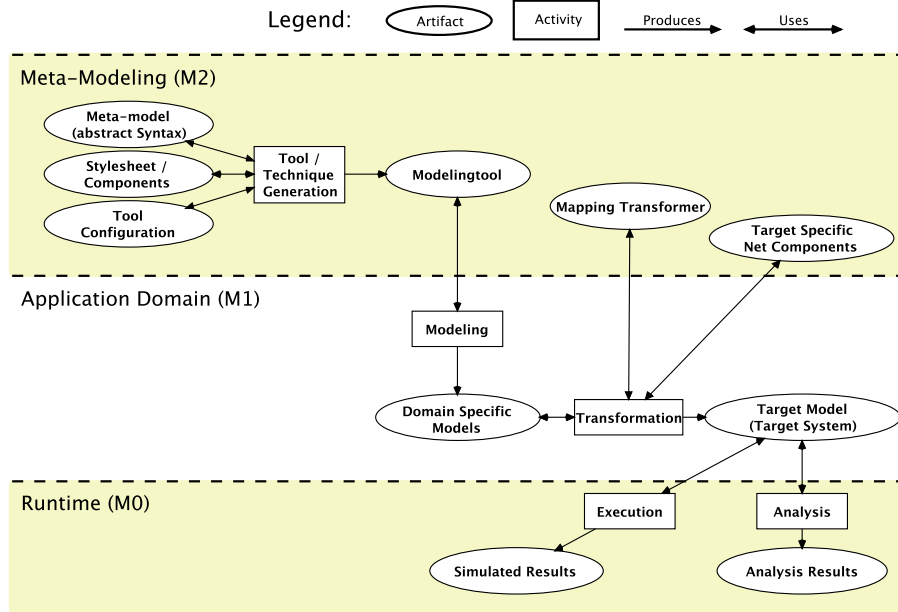


Figure 2. Artifacts and process within the RMT usage workflow.

formation. First, the semantic targets for the syntactic elements are defined as net components, which can be modified or exchanged easily. We are even able to provide several target mapping sets of net components, which can be expressed using distinct formalisms. Thus we are able to transform one source model into multiple forms of target models. For instance, we could transform a workflow description into a PT net for analytic examination and transform the same source model to a colored Petri net for simulation / execution within a real world application.

3 Developing a Prototype for BPMN

In the previous section we introduced a conceptual approach to developing modeling languages. We now show the concept in practice and demonstrate the concrete models, which are utilized in the development process. We have chosen to present, as example, the well-known modeling technique BPMN (Business Process Model and Notation [16]), in order to demonstrate the presented tool set.

In Section 3.1 we develop a (rather simple) modeling language that implements a subset of BPMN. We show, how model transformations can be used to generate Petri net models, which provide formal semantics to the abstract BPMN models. The generated Petri net models can be referred to for analyzing a BPMN process.

In a subsequent step a more specific modeling language is developed in Section 3.2. This second language – the BPMN_{AIP} formalism – enriches concepts from BPMN with domain-specific elements from the context of P*AOSE (see Section 1). The intention is to demonstrate the flexibility of the RMT approach and the appropriateness for agile, rapid and prototypical model-driven language development.

3.1 BPMN

We start with a simple subset of BPMN. Since BPMN has been described extensively in the context of modeling, meta-modeling and also in the context of Petri nets, we do not need to go into detail about the underlying semantics. A mapping of syntactic elements of BPMN to PT net components has been proposed by Dijkman et. al. [5]. Using these Petri net mappings we can focus on the aspects of agile language development instead. We concluded in Section 2.1 that a modeling language is based on the specifications of abstract and concrete (graphical) syntax.

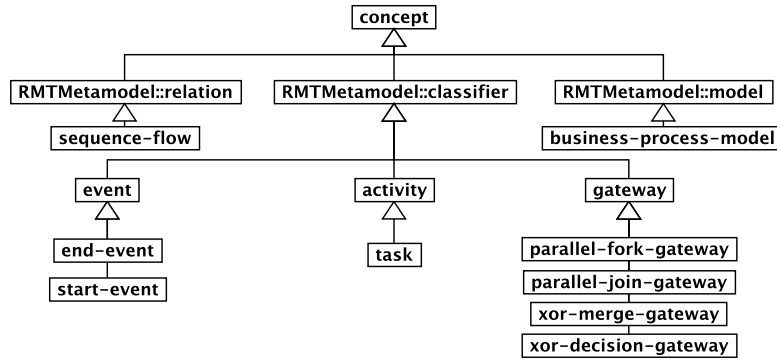


Figure 3. A meta-model for a subset of BPMN language constructs.

Figure 3 shows a meta-model for the chosen fragment of the BPMN. All concepts defined in this meta-model are instances of three basic concepts from the RMT meta-model: *model*, *classifier*, *relation*. Also shown in Figure 3 is that the three basic concepts are themselves instances of the single core concept (*concept*). The developed BPMN language defines a model type, the *business-process-model*. Also defined are *events*, *activities* and two different *gateways*, one for parallel processing and one with exclusive alternatives. These concepts can be connected through the *sequence-flow* relation. These concepts alone define the abstract syntax of the simplified BPMN formalism.

In order to complete the modeling language and generate the respective supporting modeling tool, the RMT approach requires additional information. One

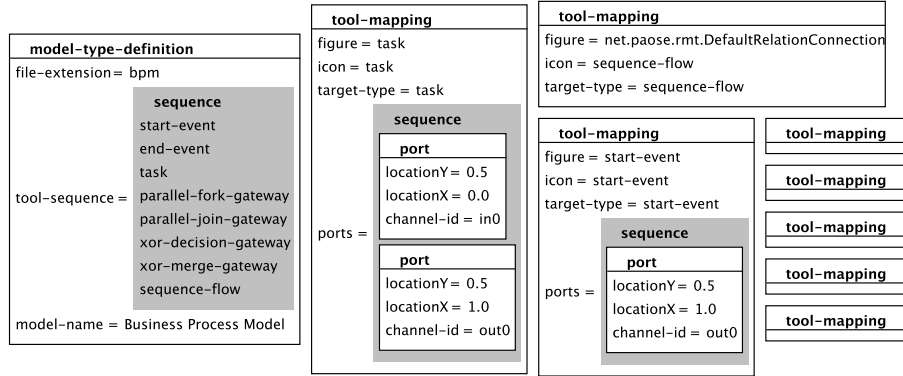


Figure 4. The tool configuration model for the BPMN modeling tool (with partly collapsed tool mappings).

is the visual representation of graphical constructs. These are developed using the built-in graphical constructs of the RENEW drawing framework. Each graphical figure is stored in a separate drawing file (template) and can be used as syntactic element for modeling later on. Another required information is a specification of properties for the modeling tool. An example of a tool configuration is shown in Figure 4. This model contains basic properties such as a model name and a file extension as well as a set of tool mappings. The latter define mappings from concepts of the meta-model (*target-type*) to graphical constructs (net components). Connectors of the constructs are specified as *ports*, relative to their position. All elements of the tool configuration are expressed in Semantic Language (SL), which can be compared to Yaml or JSON and defined using the SLEditor plugin for RENEW, which provides a UML-like representation as well as editing support for the modeler.

Figure 5 shows the graphical components representing the syntactic elements of the BPMN language alongside with the RENEW UI, which presents the loaded palette for the BPMN drawing tools. The graphical components are defined in

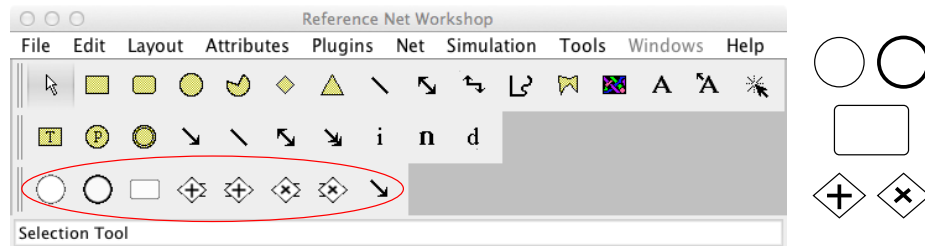


Figure 5. The RENEW UI with the tool palette providing BPMN elements.

separate template drawings. The templates define the concrete syntax for the BPMN technique. This concludes the specifications for the modeling language and enables us to generate the plugin for the modeling tool. During the generation process the RMT generator (automatically) prepares the images that are used for the tool buttons on the basis of the graphical templates. The icon images of parallel and alternative gateways where slightly modified as shown in the encircled part of Figure 5 to better distinguish the complementary constructs of split and join figures.

Using the generated BPMN plugin we are now able to model with this new technique using the RENEW editor. Figure 6 shows a ticket workflow described in BPMN. The process reflects the lifecycle of support tickets in a conventional issue tracking system. Issues are created and at some point assigned to the holder of a certain role. They can be either rejected or accepted in which case the corresponding task will be carried out by the assignee. Later on, the task may be discontinued (*unassigned*) or completed (*finish*).

With the mapping of Dijkman et. al. [5] for the transformation to Petri nets we are able to transform the given workflow to a PT net model. The generated Petri net constitutes the transformational semantics of the BPMN process in the context of the RENEW simulation environment. In consequence, the resulting model can now be executed or analyzed using for instance the RENEW simulator.

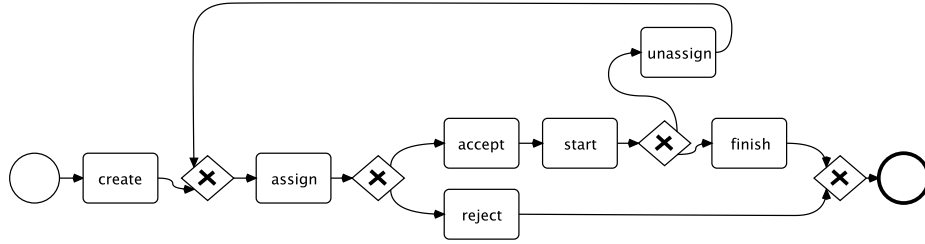


Figure 6. The lifecycle of tickets in a issue tracking system, modeled as BPMN.

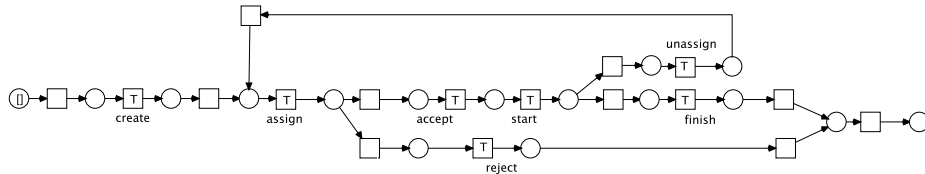


Figure 7. The target model of the lifecycle of tickets as PT net.

3.2 BPMN_{AIP}

The example presented in the preceding section describes the development of a modeling language together with a corresponding modeling tool (as RENEW plugin) following the RMT approach. Based on meta-models the approach provides a high level of flexibility in all stages during the development of modeling languages. This enables language developers to rapidly prototype specific languages, evaluate them and adapt them according to their needs. To further illustrate this flexibility, we now present a domain-specific variation of BPMN, called BPMN_{AIP}, which is used within the P*AOSE approach.

We use BPMN_{AIP} to model agent interaction protocols. In contrast to Agent Interaction Protocol Diagrams (a variation of Sequence Diagrams [3, Chapter 13]), the BPMN_{AIP} formalism allows to shift the focus to the internal agent processes. The presented agent-specific extensions have been proposed by Haustermann [7] in order to augment a subset of BPMN for the use within the P*AOSE approach. With the RMT approach it is possible to refine the BPMN language in an agile process and develop a corresponding modeling language (BPMN_{AIP}), which satisfies the demands of a given domain-specific context.

BPMN_{AIP} extends the BPMN subset in the previous section by incoming (drawn as white envelopes) and outgoing (black envelopes) message events and special tasks for agent-specific operations. The *dc-exchange-task* represents the synchronous or asynchronous call of an internal service. The *kb-access-task* serves for accessing the agent's internal knowledge base. To use these constructs in the modeling tool, they have to be added to the meta-model. Figure 8 shows the extensions of the meta-model in Figure 3. With these extensions the modeling tool can already be used with the added constructs. The generated modeling tool uses a standard representation and standard task bar tool buttons to allow running early tests if none are provided by the developer. In order to define a customized concrete syntax, analogously to the previous example, a representation template drawn with the RENEW tool, a button icon generated from the template image and a tool mapping entry in the tool configuration as shown in Figure 4 is sufficient.

In addition to the agent-specific constructs, BPMN_{AIP} also has a domain-specific semantics. The semantics is based on the agent framework that is applied in the P*AOSE approach, which uses Petri nets to implement agents and the agents' behavior. Therefore the semantic net components for the target model are tailored for the fitness within the used framework.

In order to obtain another semantics it is possible to provide a different set of net components. The RMT framework is able to handle different transformation engines and multiple net component sets. For the BPMN_{AIP} formalism the MULAN net components by Cabac are used [3, Chapter 5].

Figure 9 shows an adaptation of the ticket service example using BPMN_{AIP}. The management of the ticket status is now provided by an agent, the Ticket Agent, which can delegate tasks to other agents. In this example the task to export some drawing to an image is assigned to an Export Agent (as described by Cabac et. al. [1]), which is informed about the assignment with a message.

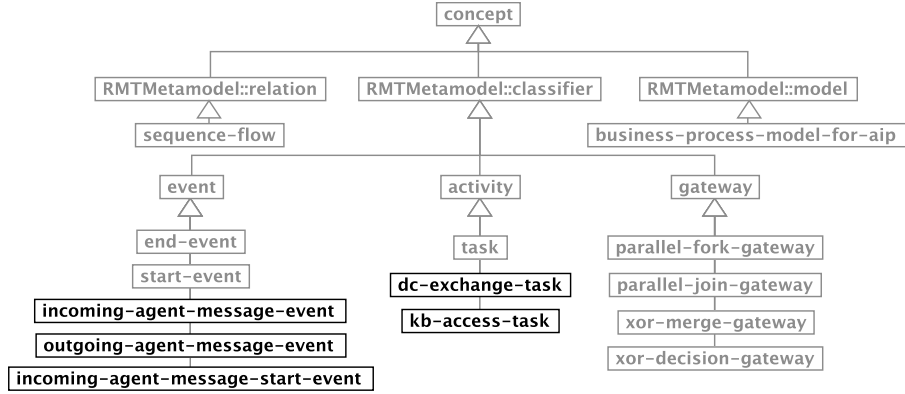


Figure 8. BPMN_{AIP} extensions to the BPMN meta-model (Figure 3).

This message results in an instantiation of the process depicted in Figure 9b. The Export Agent checks his knowledge base if it can export the drawing and delegates the task to an internal service, if possible. The Ticket Agent changes the status of the ticket according to the messages he receives as answer from the Export Agent.

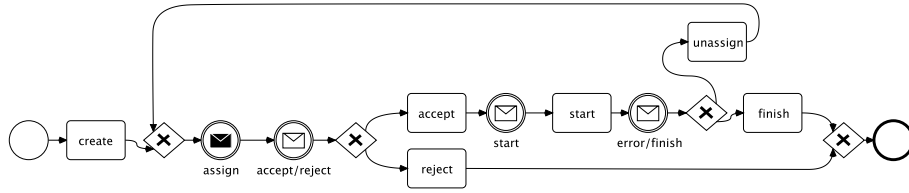


Figure 9a. The Ticket Agent.

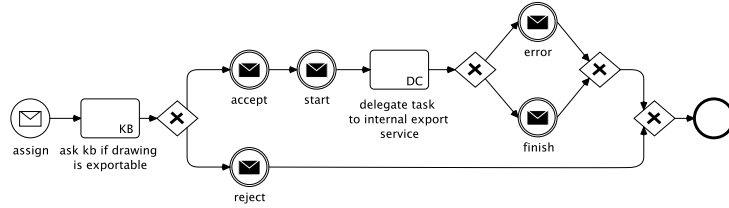


Figure 9b. The Export Agent.

With the described semantics the two agent processes in Figure 9 are transformed by the RMT-based BPMN_{AIP} modeling tool into the Petri nets shown in Figure 10. These nets are protocol net skeletons, which have to be completed by additional implementation details in order to be runnable with the used frame-

work. The figure illustrates the structure of the generated nets. A part of the net is zoomed in to exemplarily show the details of the net components. The zoomed part refers to the internal delegation of the task and the answer to the Ticket Agent.

4 Related Work

In this publication we motivate the rapid and prototypical development of domain-specific modeling languages. There are a number of related publications on prototyping domain-specific languages (DSL), each focussing on different aspects or application domains. Blunk et. al. [2] see the best gain for prototyping DSL as an extension of a general purpose programming language. Sadilek et. al. [20] stress the increasing demand for supporting agile approaches to the development of DSML. They “argue that for prototyping a DSML on the platform independent level, its semantics should not only be described in a transformational but also in an operational fashion” [20, p. 63]. However, they use the Query View Transformation (QVT) language to implement operational semantics of Petri nets, rather than exploiting the operational semantics to formalize the semantics of a second modeling language, as done here. Rouvoy et. al. [19] specialize on the domain of architecture description languages (ADLs) and develop a modular framework for prototyping ADLs based on the Scala language. The presented method emphasizes the high degree of automation through generative methods, the automated generation of modeling tools and also the automated transformation of abstract models to Petri nets.

Nytun et. al. [15] provide a categorization for different approaches to automated tool generation in the context of meta-modeling and DSML. The authors examine various meta-modeling approaches according to the four categories: structure, constraints, representation and behavior. With the RMT approach we cover most of these aspects by utilizing concept diagrams, representational mappings and Petri net-based target models. At the current time we do not provide any means to define constraints, but we plan to introduce constraints in the future.

With the claim of addressing general problems of defining DSML semantics our goal consists in developing a Petri net based framework through combining techniques of meta-modeling with Petri nets engineering. With the Event Coordination Notation (ECNO) Kindler [8] takes a model-driven approach, which uses Petri net models to implement local components behavior. The collaboration of components is defined in abstract coordination diagrams. The implementation is based on the wide spread Eclipse Modeling Framework (EMF). In combination with the EMF, the graphical modeling framework (GMF) can be used to automatically generate specific modeling tools from meta-models. The idea of generating domain-specific tools from models was adopted for this work, but we try to take a minimalistic approach instead of overcharging the tool with features, thus increasing complexity. The intrinsic complexity is a point of criticism concerning meta-modeling frameworks [19, p. 14].

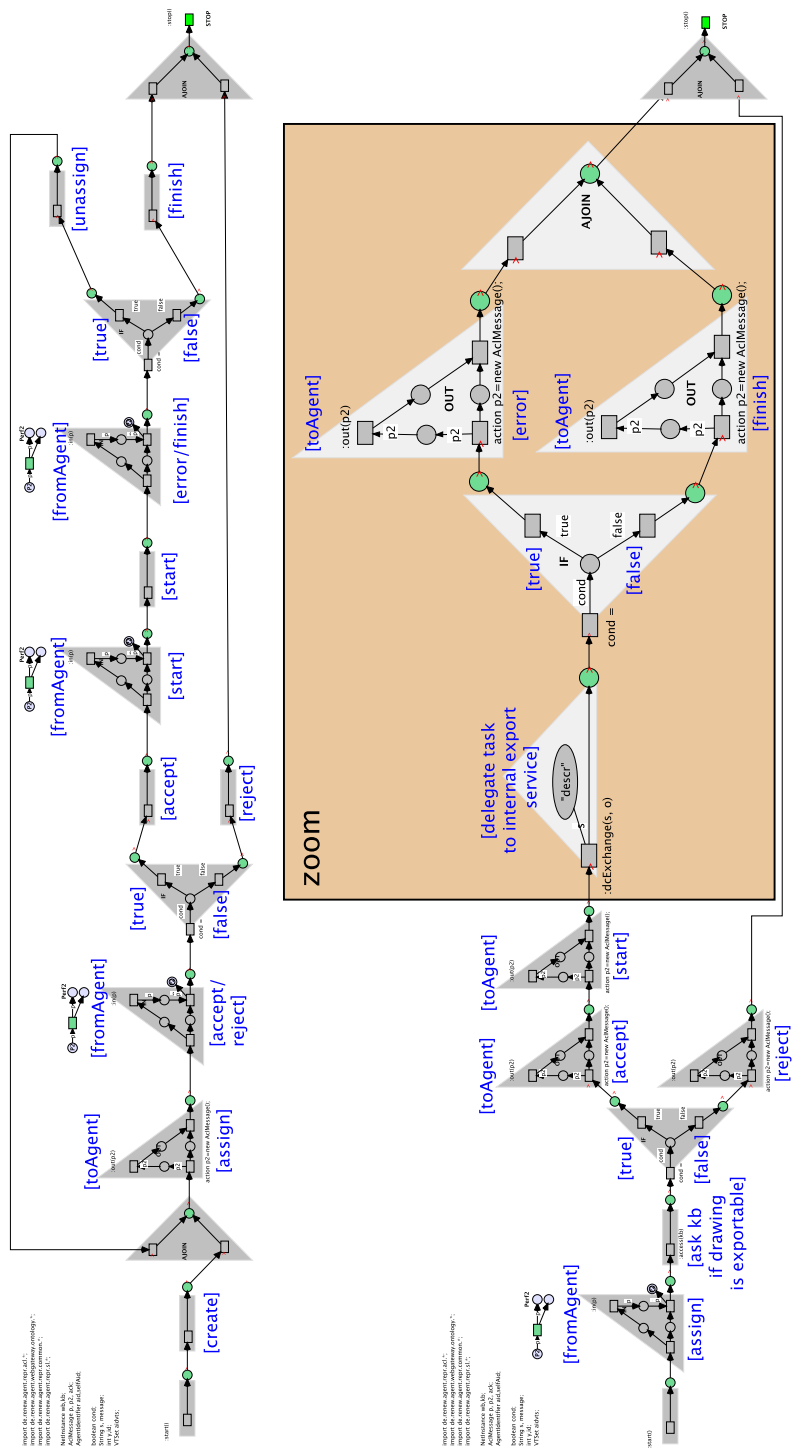


Figure 10. Generated protocol net skeletons.

Dijkman et. al. [5] show a mapping of BPMN constructs to Petri nets and elaborate on the semantics of such transformations. On the basis of that work there exists a tool for converting BPMN models to PNML and a tool for converting BPMN to YAWL. With the flexible tool presented in this work the languages can be quickly adopted and the concerns about problems in evaluating BPMN models using Petri net semantics can be empirically investigated. Lohmann et. al. [12] provide a basis for analyzing different business process modeling languages with respect to their realizability using Petri nets semantics for BPEL, BPMN, EPC, YAWL. This can be a good starting point for further research using the presented tool. The RMT framework has been applied by Möllers [14] for the development of a modeling tool for the design and execution of Deployment Diagrams.

5 Conclusion

In this contribution we present the RMT approach, which enables us to develop modeling languages and modeling tools by applying concepts of model-driven development. The key aspects of this approach are the use of meta-models for automatic tool generation and transformation of models, exploiting the formal semantics of Petri nets. Based on our continuously developed graphical modeling tool and Petri net simulation environment RENEW (see [11]) we provide the technical realization of the RMT approach. The RMT framework provides the means to describe modeling languages building on the concepts of software language engineering (cf. Section 2). The abstract syntax, concrete syntax and tool configurations are provided as model-based specifications of the desired modeling languages and tool behavior. The semantics is defined as transformation-based operational semantics using Petri net formalisms as target models. With this environment we are able to provide the representation directly within our graphical framework, leading to appropriate language constructs, which can be designed for special purposes that fit the needs and expectations of its users. With the RMT approach the users are able to develop and adapt their own languages/modeling techniques, define constructs based on graphical representations and finally generate modeling tools, which empower them to draw models in domain-specific languages.

Depending on the chosen and intended formalism, we could even go one step further. We are able to simulate the transformed models directly, if there exists an operational semantics that can be mapped to the formalisms we already have implemented within the RENEW context. For experimental environments where users want to define a special purpose language that suits exactly their current needs we can therefore provide a powerful tool set.

While the prototypical development of languages is already quite fast, we now have to address the question of sustainable meta-modeling-based tools. We have already successfully applied the tool several times within our P*AOSE approach. In this context we expect that further new modeling languages can be developed in a prototyping approach. In the future we wish to provide the means

to support hierarchical modeling within the RMT framework. With the Nets-within-Nets paradigm [21] the concepts to support hierarchical target models already exist. Since the whole P*AOSE approach is Petri net-based, the direct support by simulation of target models within RENEW is implicitly given. The prototyping approach of languages empowers us to evaluate several languages in order to improve specific frameworks that are already at hand.

Acknowledgment We thank Dr. Daniel Moldt and the TGI group of the Department of Informatics, University of Hamburg for the support, constructive criticism and fruitful discussions.

References

1. Betz, T., Cabac, L., Duvinneau, M., Wagner, T., Wester-Ebbinghaus, M.: Software engineering with Petri nets: a Web service and agent perspective. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)* pp. 41–61 (2014), http://link.springer.com/chapter/10.1007/978-3-662-45730-6_3
2. Blunk, A., Fischer, J.: Prototyping domain specific languages as extensions of a general purpose language. In: Haugen, O., Reed, R., Gotzhein, R. (eds.) *System Analysis and Modeling: Theory and Practice*, *Lecture Notes in Computer Science*, vol. 7744, pp. 72–87. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-36757-1_5
3. Cabac, L.: *Modeling Petri Net-Based Multi-Agent Applications, Agent Technology – Theory and Applications*, vol. 5. Logos Verlag, Berlin (2010)
4. Cabac, L., Mosteller, D., Wester-Ebbinghaus, M.: Modeling organizational structures and agent knowledge for Mulan applications. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC)* pp. 62–82 (2014), http://link.springer.com/chapter/10.1007/978-3-662-45730-6_4
5. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and analysis of business process models in BPMN. *Information and Software Technology* 50(12), 1281 – 1294 (2008), <http://dx.doi.org/10.1016/j.infsof.2008.02.006>
6. Harel, D., Rumpe, B.: Meaningful modeling: what’s the semantics of "semantics"? *Computer* 37(10), 64–72 (Oct 2004)
7. Haustermann, M.: *BPMN-Modelle für petrinetzbasierte agentenorientierte Softwaresysteme auf Basis von Mulan/Capa*. Master thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg (Sep 2014)
8. Kindler, E.: *Coordinating Interactions: The Event Coordination Notation*. Tech. Rep. 05, DTU Compute - Department of Applied Mathematics and Computer Science, Technical University of Denmark (2014)
9. Kleppe, A.: *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education (2008)
10. Kummer, O.: *Referenznetze*. Logos Verlag, Berlin (2002), <http://www.logos-verlag.de/cgi-local/buch?isbn=0035>
11. Kummer, O., Wienberg, F., Duvinneau, M., Cabac, L.: *Renew – User Guide (Release 2.4.2)*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg (Jan 2015), <http://www.renew.de/>

12. Lohmann, N., Verbeek, E., Dijkman, R.: Petri net transformations for business processes - a survey. In: Jensen, K., Aalst, W. (eds.) *Transactions on Petri Nets and Other Models of Concurrency II*, Lecture Notes in Computer Science, vol. 5460, pp. 46–63. Springer Berlin Heidelberg (2009)
13. Moldt, D.: Petrinetze als Denkzeug. In: Farwer, B., Moldt, D. (eds.) *Object Petri Nets, Processes, and Object Calculi*. pp. 51–70. No. FBI-HH-B-265/05 in Report of the Department of Informatics, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg (Aug 2005)
14. Möllers, K.S.M.: Ein Ansatz zur Dynamisierung von Verteilungsdiagrammen anhand der Entwicklung eines Mulan-Werkzeugs. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg (2014)
15. Nyttun, J.P., Prinz, A., Tveit, M.: Automatic Generation of Modelling Tools. In: Rensink, A., Warmer, J. (eds.) *Model Driven Architecture - Foundations and Applications*, Lecture Notes in Computer Science, vol. 4066, pp. 268–283. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11787044_21
16. OMG, Object Management Group: Business Process Model and Notation (BPMN) – Version 2.0.2 (2013), <http://www.omg.org/spec/BPMN/2.0.2>
17. PAOSE-Website: Petri Net-Based, Agent- and Organization-oriented Software Engineering. University of Hamburg, Department of Informatics, Theoretical Foundations Group: <http://www.paose.net> (2015)
18. Petrasch, R., Meimberg, O.: *Model Driven Architecture: eine praxisorientierte Einführung in die MDA*. Heidelberg: dpunkt-Verlag (2006)
19. Rouvoy, R., Merle, P.: Rapid Prototyping of Domain-Specific Architecture Languages. In: Larsson, M., Medvidovic, N. (eds.) *International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'12)*. pp. 13–22. ACM, Bertinoro, Italie (Jun 2012), <http://hal.inria.fr/hal-00690607>
20. Sadilek, D., Wachsmuth, G.: Prototyping visual interpreters and debuggers for domain-specific modelling languages. In: Schieferdecker, I., Hartman, A. (eds.) *Model Driven Architecture - Foundations and Applications*, Lecture Notes in Computer Science, vol. 5095, pp. 63–78. Springer Berlin Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-69100-6_5
21. Valk, R.: Object Petri Nets – Using the Nets-within-Nets Paradigm. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Advances in Petri Nets: Lectures on Concurrency and Petri Nets*, Lecture Notes in Computer Science, vol. 3098, pp. 819–848. Springer-Verlag, Berlin Heidelberg New York (2004), <http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3098&page=819>

Validating DCCP Simultaneous Feature Negotiation Procedure

Somsak Vanit-Anunchai

School of Telecommunication Engineering, Institute of Engineering
Suranaree University of Technology, Muang, Nakhon Ratchasima, Thailand
Email: `somsav@sut.ac.th`

Abstract. This paper investigates the feature negotiation procedure of the Datagram Congestion Control Protocol (DCCP) in RFC 4340 using Coloured Petri Nets (CPNs). After obtaining a formal executable CPN model of DCCP feature negotiation, we analyse it using state space analysis. The experimental result reveals that simultaneous negotiation could be broken on even a simple lossless FIFO channel. In the undesired terminal states, the confirmed feature values of Client and Server do not match.

Keywords: Datagram Congestion Control Protocol, Feature Negotiation, Coloured Petri Nets, State Space Analysis

1 Introduction

In 2006, the Internet Engineering Task Force (IETF) published a set of standards for a transport protocol, the Datagram Congestion Control Protocol (DCCP) [15] comprising RFC 4336 [6]; RFC 4340 [16]; RFC 4341 [7] and RFC 4342 [9]. RFC 4336 discusses problems and disadvantage of existing transport protocols and the motive for designing a new transport protocol for unreliable datagrams. RFC 4340 specifies reliable connection management procedures; reliable negotiation of options; acknowledgement and optional mechanisms used by congestion control mechanisms. RFC 4340 also provides the extension for modular congestion control, called *Congestion Control Identification* (CCID) but the congestion control mechanisms themselves are specified in other RFCs. Currently there are three published standards, RFC 4341, CCID2: TCP-like congestion control [7], RFC 4342, CCID3: TCP-Friendly Rate Control [9] and CCID4: RFC 5622 TCP-Friendly Rate Control for Small Packets [8].

1.1 Motivation

Unlike TCP, DCCP does not impose flow control on data transfer. But state information such as the sequence number sent and received is still required in order to trace packet loss which is crucial for congestion control. From the sequence number variables, a sequence number validity window is set up [16] to defend

against attacks from hackers. Thus *connection management procedures* specified in RFC4340 are used to set up and clear the state information. Apart from the reliable connection management, both sides must choose congestion control mechanisms and agree upon the same CCID. This requires a reliable negotiation procedure called *Feature Negotiation* which is also specified in RFC4340. If both sides are not aware of reaching an agreement with different CCIDs, the situation will be very harmful and currently there is no recovery mechanisms. Hence it is vital to verify that the DCCP feature negotiation procedure works correctly. In this paper we use Coloured Petri Nets (CPNs) [12] to formally model and analyse DCCP feature negotiation procedures.

1.2 Related Work

Formal methods [1] are techniques based on mathematically defined syntax and semantics for the specification, development and verification of software and hardware systems. They remove ambiguities and are indispensable for checking correctness of high-integrity systems. Coloured Petri Net (CPN) [14] is a formal method which is widely used [2,3,5,13,17] to model and analyse concurrent and complex system. An important advantage of CPNs is its graphical notation with the abstract data types providing a high level of user friendliness. CPNs were used to verify industrial scale protocols such as the Wireless Application Protocol (WAP) [10], the Internet Open Trading Protocol (IOTP) [18], TCP [11] and DCCP [21]. [21] studied DCCP connection management operating over re-ordering channels with no loss using Coloured Petri Nets. [23] extended the work in [21] by including DCCP simultaneous open procedure (RFC 5596) and Network Address Translators (NAT) in the model. However regarding DCCP feature negotiation procedure, there are very few articles [19,20] investigating it. As far as we are aware of, DCCP feature negotiation has not been formally modelled and analysed before.

1.3 Contribution

The contribution of this paper is three fold. Firstly, as far as we are aware of this paper presents the first formal executable model of DCCP feature negotiation. Secondly the formal analysis helps us identify an error in the specification. Thirdly, investigating the state space analysis provides us insight what causes the error.

This paper is organised as follows. Section 2 provides an overview of the protocol and packet format. Section 3 briefly describes DCCP feature negotiation procedure. The description of the CPN model of DCCP feature negotiation is described in section 4, which starts with modelling assumptions and specification interpretation. Section 5 discusses analysis result and insight. Section 6 presents the conclusion of this paper and future work.

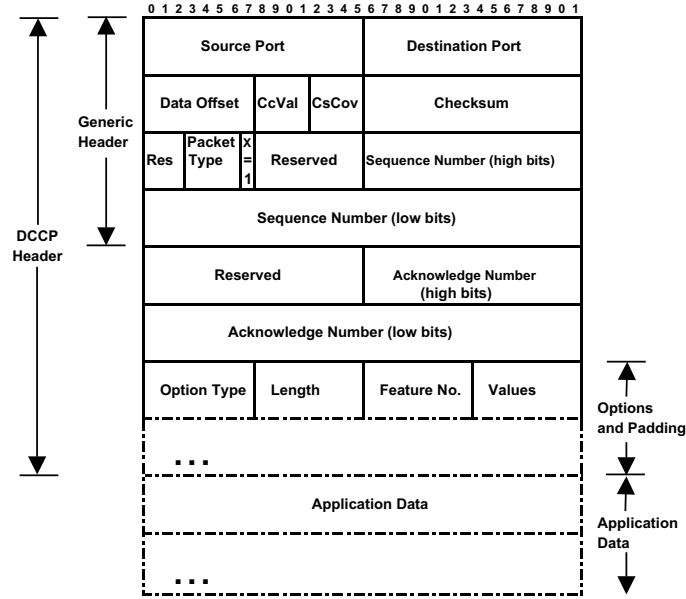


Fig. 1. DCCP packet format.

2 DCCP Overview

The Internet protocol architecture is organized into five layers known as the TCP/IP reference model. While TCP is a transport protocol that provide the reliable delivery of a byte stream, DCCP is a transport protocol for the timely but unreliable delivery of datagrams. DCCP can be viewed as an upgraded version of UDP equipped with new facilities for connection management; acknowledgement; feature negotiation and congestion control.

2.1 DCCP Packet Format

DCCPs exchange packets over the Internet Protocol between a client and a server. The protocol uses 11 packets to setup and release connections and transfer data. RFC 4340 [16] defines a DCCP packet as a sequence of 32 bit words comprising a DCCP Header and Application Data area as shown in Fig. 1. The header comprises a generic header (applicable to all packets), followed by an acknowledgement number (if any) and then the options field. The length of the Option and Application Data fields can vary.

The DCCP header contains 16 bit source and destination port numbers, and a 16 bit checksum. An 8 bit data offset indicates the length in 32-bit words from the beginning of the Header to the beginning of the Application data. CCVal, a 4 bit field, is a value used by the congestion control mechanisms [9]. Checksum

Coverage (CsCov), also a 4 bit field, specifies the part of the packet being protected by the 16 bit checksum. The four bit Packet Type field specifies the name of the packet: Request, Response, Data, DataAck, Ack, CloseReq, Close, Reset, Sync, SyncAck and Listen. Request and Data packets do not include acknowledgement numbers. The sequence numbers of Data packets and the sequence numbers and acknowledgement numbers of Ack and DataAck packets can be reduced to 24-bit short sequence numbers when setting the Extend Sequence Number (X) field to 0.

The Options field contains state information or commands for applications to negotiate various features such as the Congestion Control Identifier (CCID) and the width of the Sequence Number validity window [16].

2.2 Options Fields

The options field is a multiple of 32-bit words which may contain more than one option. Because each option consists of a multiple of 8 bits, the field may need to be padded to the word boundary. Options are classified into two groups: single byte and multi-byte. A single byte option has a value from 0 to 31 which represents an option type. An Option type is a 8-bit integer which represents the meaning of the option, such as 1 meaning mandatory, 2 meaning slow receiver [16]. The format of a multi-byte option is shown in Fig. 1. The first byte is an option type. The second byte is the length in bytes of each option including the option type field, the length and data of the option. The data comprises a number of features, the format of which will be explained in section 3.

3 Feature Negotiation Procedure

DCCP allows both the client and server to change their parameters called *features* using feature negotiation procedures. The negotiation can happen at any time but typically during connection establishment. Each entity can initiate the negotiation of two kinds of features: *local features* (L)-the initiator's features and *remote features* (R)-the other side's features. Four particular options are dedicated to feature negotiations; Change L, Confirm L, Change R and Confirm R. The option types have values of 32 to 35 respectively. The format of Confirm or Change Options including feature numbers and feature values are shown in

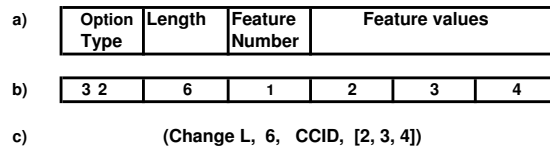


Fig. 2. Option format in DCCP header and an example of a Change L option.

Fig 2 a). Fig 2 b) shows six of 8-bit values representing a Change L option when negotiating CCID. The meaning of each 8-bit values is shown in Fig 2 c).

The feature number identifies the feature. For instance, 1 refers to CCID and 2 means short sequence numbers are allowed. The complete list of features is given in [16]. To reach agreement on a feature value, a reconciliation rule known to both sides is required. Currently RFC 4340 defines two reconciliation rules: server priority and non-negotiable.

1. The server priority rule: This rule is applied when the feature value is a fixed-length byte string. During negotiation DCCP entity keeps an ordered preference list of the feature values. The initiator sends a Change option containing its preference list. The receiver responds with the Confirm option containing an agreed value followed by its preference list. Thus the agree value will appear twice in the Confirm option. The agreed value is defined as the first element in the server's list that matches any element in the client's list. If there is no match, the agreed value remains the existing feature value.

For example, the client sends $32,6,1,2,3,4$ corresponding to Change L(32), length(6), CCID(1), the client's preference list(2,3,4). This means the client proposes to change its CCID and the preferred CCIDs are CCID#2, CCID#3 and CCID#4 respectively. The server responds $35,7,1,3,3,4,2$ corresponding to Confirm R(35), length(7), CCID (1), agreed value (3) and the server's preference list (3,4,2). According to the Client's and Server's preference lists in this example, the client must use CCID#3.

2. Non-negotiable rule: The Change and Confirm options under this rule contain only one feature value which is a byte string. After receiving the Change L from the feature local, the feature remote must accept the valid value and reply with Confirm R containing this value. If the received feature value is invalid, the feature remote must send an empty Confirm R. This non-negotiable rule must not be used with Change R and Confirm L options.

For example the client sends $32,9,3,0,0,0,0,4,0$ corresponding to Change L(32), length(9), Sequence number window (3), value of window size(1024). The server replies with $35,9,3,0,0,0,0,4,0$.

3.1 Finite State Machines

The feature negotiation procedures are represented by state diagrams. Figure 3 shows the state diagram for *feature local*. It comprises three states: STABLE; CHANGING; and UNSTABLE. The entity in the STABLE state always knows its feature value and expects the other end agrees with the same value. When the local receives Change R, it calculates a new agreed value and replies Confirm L. On the other hand the Confirm R received will be discarded.

After the entity in STABLE sends the first Change L command, it enters the CHANGING state and goes back to the STABLE state upon receiving a Confirm R or an empty Confirm R. When the local in CHANGING does not get reply from the other side, it keeps retransmit the Change L option.

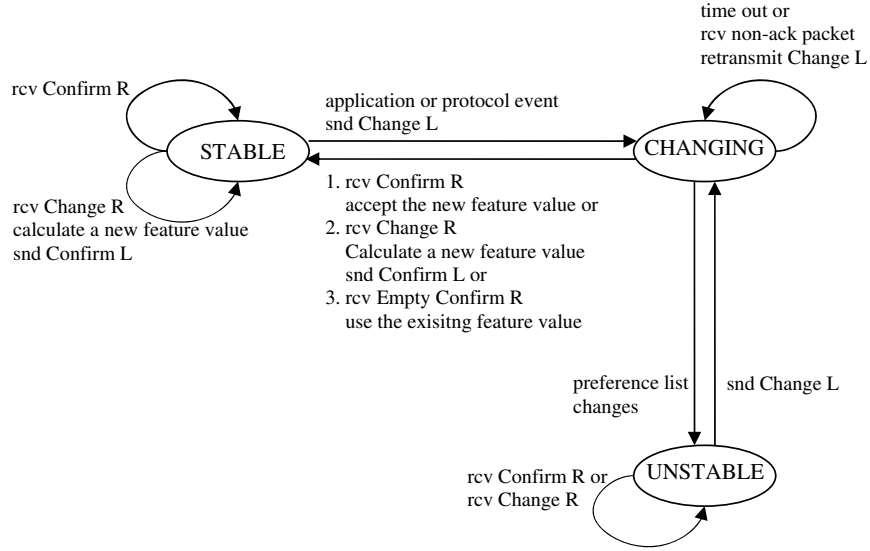


Fig. 3. DCCP feature negotiation state diagram - redrawn from [16].

When the preference list is changed by its user while the entity is in the CHANGING state, it enters the UNSTABLE state. Here it ignores the on-going negotiation but starts a new negotiation by sending a Change command with the new preference list before going back to the CHANGING state.

The state diagram for *feature remote* can be obtained by interchanging *Ls* and *Rs* in Fig. 3. Thus each entity consists of three state machines working together: connection management, feature local and feature remote. It is possible that one side initiates Change L while the other side initiates Change R of the same feature. According to Fig. 3 when the local in CHANGING receives Change R, it computes a new agree value and replies Confirm L. This situation is called *simultaneous negotiation*. The specification also allows the preferences to be changed at any time.

3.2 Important Rules of Feature Negotiation

Although the feature negotiation procedures explained in the previous section sound simple, the real situation could be very complex when packets are re-ordered and lost. Moreover the negotiation for the same feature could be simultaneously initiated by both sides and the preference lists can be changed at any time. To cope with this, the RFC specifies some rules intended to provide reliable signalling so that both sides reach agreement with the same feature value.

Non-reordered Change and Confirm Options The RFC specifies that the Change and Confirm options in packets that do not arrive in strictly increasing order must be ignored. According to the related pseudo code and algorithms, the strictly increasing order rule is only enforced for packets that contain the Change and Confirm options. An ordered packet with the Change and/or Confirm options may have a sequence number less than GSR if the later packets do not contain any Change or Confirm options.

In order to check the order of arrival, the RFC specifies another two variables: Feature Greatest Sequence Number Received (FGSR) and Feature Greatest Sequence Number Sent (FGSS). If the received packet's sequence number is less than or equal to FGSR, Change or Confirm options received must be ignored. If the acknowledgement Number is less than FGSS or the packet contains no acknowledgement, the Confirm option received must be ignored.

Because DCCP-Data with short sequence numbers is vulnerable to be attacked, any option attached to DCCP-Data that might cause the connection to be reset shall be ignore. *Thus both Change and Confirm options received in DCCP-Data must be ignored in all circumstances.* A sequence number valid packet received that contains non-reordered Change or Confirm options updates FGSR while FGSS is updated when the entity sends a Change Option during a transition from STABLE or UNSTABLE to CHANGING.

Retransmission Because the reordered options are ignored or the packet can be lost, Change options must be retransmitted when the sender does not receive a non-reordered Confirm option within a specific period. The Confirm option must be generated only when a non-reordered Change option is received. Retransmission of options may be achieved by either generating a new packet (DCCP-Ack or DCCP-Sync) or by including the appropriate option field in a packet that is about to be transmitted. Retransmission continues until a non-reordered Confirm option is received or the connection is closed down.

4 CPN Model of DCCP Feature Negotiation (DCCP-FN)

4.1 Modelling Assumptions and Specification Interpretation

We make the following assumptions regarding DCCP feature negotiation when creating our model.

1. This paper assumes the medium to be First-in First-out (FIFO) channels with no loss. There are three reasons supporting this assumption. Firstly, according to RFC 4340 the Change and Confirm Options must arrive in strictly increasing order otherwise it will be ignored. This requirement implies that actually DCCP feature negotiation protocol operates over FIFO channels. Secondly, reordered and/or lossy channels can mask out inherent errors such as unspecified receptions which could appear when protocol operates over FIFO channels with no loss. Thus protocol validation shall be started from operating over the

FIFO channels with no loss. Thirdly, the assumption of FIFO channel makes the model simpler. We can abstract away irrelevant details such as sequence number, acknowledgement number, state variables FGSS and FGSR.

2. Although we agree with [20] that the feature negotiation is not independent of the protocol state machine. To reduce the complexity of our CPN model, we assume that the feature negotiation is independent of the protocol state machine. Without loss of generality, instead of modelling three FSMs (connection management, feature local and feature remote) at each side, only one FSM (Fig. 3) (either the feature local's or the feature remote's FSM) is required. In particular we assign the feature local's FSM to Client and the feature remote's FSM to Server. This assumption makes the CPN model readable and easy to understand.

3. A DCCP packet is modelled by an option type and a list of feature values (preference list). Other fields such as packet type and sequence-acknowledgement numbers are omitted because they do not affect the operation of the feature negotiation.

4. RFC4340 allows many options to be sent in one packet and many features to be negotiated at the same time. Following an incremental approach [3], as a first step we consider the negotiation of *Congestion Control Identification* (CCID) that uses the server-priority reconciliation rule because the ability to negotiate the suitable congestion control mechanism is the main objective of DCCP.

5. Our model does not include the mandatory options, invalid options and unknown feature numbers.

6. RFC 4340 specifies that the preference list can be changed at any time. It is unclear what should be happened if the preference list is changed while the endpoint in STABLE. However according to [19], the endpoint can remain in STABLE if it changes the preference list without changing the preferred value. Thus we assume that the endpoint remains in STABLE after it changes the preference list. However we investigate the scenario when the endpoint changes the preference list without changing the preferred value.

4.2 Model Structure

Our model structure is inspired by [21, 22] who model and analyse DCCP connection management. However [21, 22] do not include the feature negotiation procedure. Our DCCP feature negotiation model comprises three hierarchical levels as shown in Fig. 4. The first level page is **Main_FN**. This page calls the second level pages named **FN_Local** and **FN_Remote**. The third level has six pages. Each one is named by a DCCP feature negotiation state. Figure 5 shows Global Declaration which defines the data associated with the model. The CPN diagram in the first level page (Fig 6) comprises two substitution transitions (represented

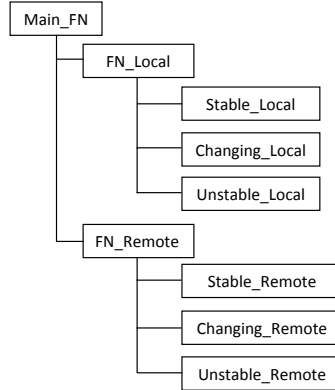


Fig. 4. The DCCP-FN hierarchy page.

```

1: (* Feature Negotiations *)
2: colset E = with e;
3: colset CCID = int with 2..255;
4: colset Confirmed_Value = CCID;
5: colset Preference_List = list CCID;
6: colset Option_Type = with ChangeL | ConfirmL
7:                       | ChangeR | ConfirmR;
8: colset Option_Field = product Option_Type
9:                       * Preference_List;
10: colset List_Option_Field = list Option_Field;
11: colset FN_State = with STABLE | CHANGING | UNSTABLE;
12: colset FN_CB = product FN_State * Confirmed_Value
13:                * Preference_List;

```

Fig. 5. Global Declaration.

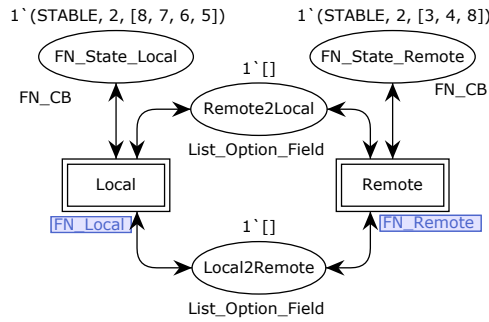


Fig. 6. The Main_FN overview page.

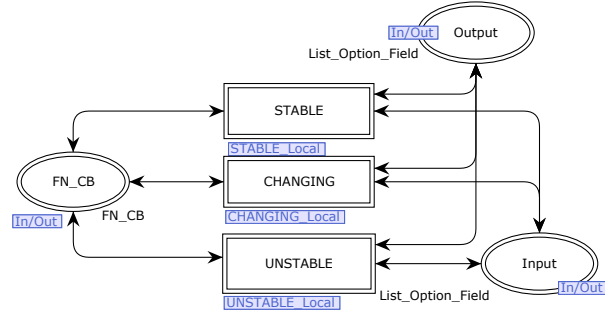


Fig. 7. The FN_Local page.

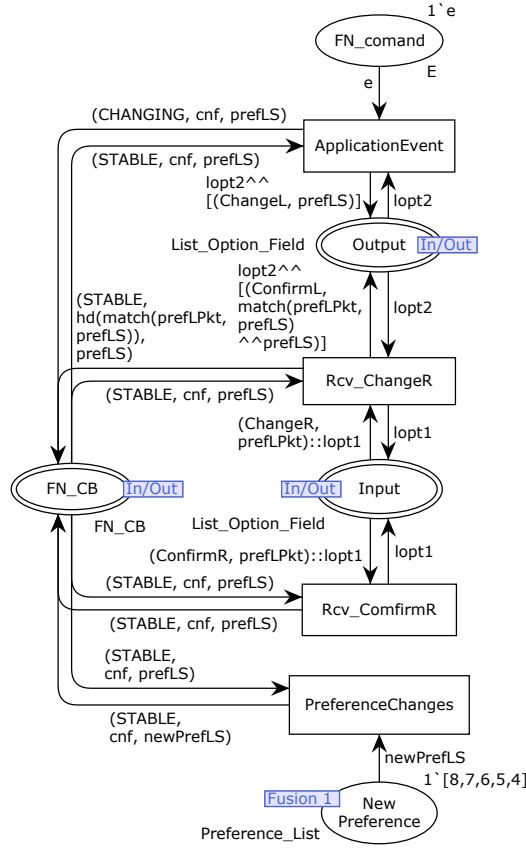


Fig. 8. The STABLE_Local page.

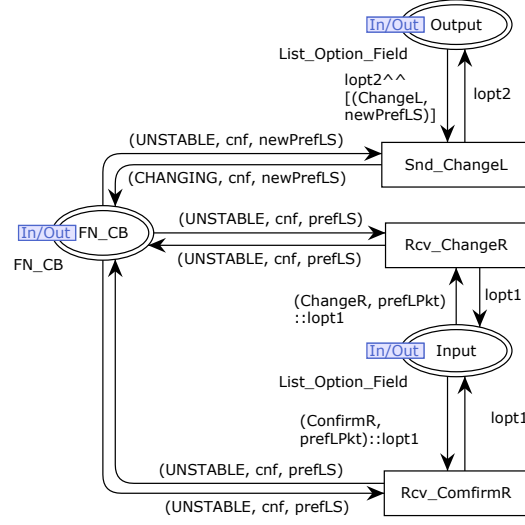


Fig. 10. The UNSTABLE_Local page.

Figure 8 captures behaviour when Local is in STABLE State. We allow DCCP entity in STABLE change its preference and remain in the STABLE state. Figure 9 and 10 model the procedures to be followed by Local when it is in CHANGING and UNSTABLE respectively.

5 Analysis of DCCP-FN CPN Model

5.1 Initial Configurations

Our DCCP feature negotiation model is analysed using CPN Tools [4,14] version 4.0 on an Intel i5-4300U 1.90GHZ with 4 GB RAM. To analyse a particular scenario, the CPN model needs to be initialised by distributing initial tokens to places `FN_State_Local` and `FN_State_Remote` (Fig. 6); places `FN_Command` and `NewPreference` in `Stable_Local` (Fig. 8) as well as places `FN_Command` and `NewPreference` in `Stable_Remote`. The channel places `Remote2Local` and `Local2Remote` initially contain an empty list. The presence of tokens 1'e in place `FN_Command` allows the entity to start the feature negotiation procedure. The analysis in this paper assumes no retransmission.

We choose to model and analyse the negotiation of the feature CCID. This feature uses the reconciliation rule: server priority. The default feature value is 2 which represents TCP-like congestion control. Although currently the standard specifies only CCID2 (RFC4341), CCID3 (RFC4342) and CCID4 (RFC5622), we make up CCID numbers in each preference list for the purpose of validating the feature negotiation procedure. Table 1 shows the values in preference lists we

Table 1. An agreed feature value before and after preference lists have been changed.

			Client (Local)	
			before	after
			[8,7,6,5]	[8,7,6,5,4]
Server (Remote)	before	[3,4,8]	8	4
	after	[4,5]	5	4

Table 2. Initial configurations of twelve possible scenarios.

Case	FN Command		Change of Preference List	
	Local	Remote	Local	Remote
1	1'e	empty	disable	disable
2	empty	1'e	disable	disable
3	1'e	1'e	disable	disable
4	1'e	empty	enable	disable
5	empty	1'e	enable	disable
6	1'e	1'e	enable	disable
7	1'e	empty	disable	enable
8	empty	1'e	disable	enable
9	1'e	1'e	disable	enable
10	1'e	empty	enable	enable
11	empty	1'e	enable	enable
12	1'e	1'e	enable	enable

used in our experiment before and after the preference has been changed. The resolved values before and after the preference changed under the server-priority reconciliation rule are shown in Table 1 as well. According to [19] the endpoint can remain in the STABLE state if it changes the preference list without changing the preferred value. Therefore at Client (Local) we keep the old preference list but adding the new feature value (4) at the end of the list.

Table 2 shows the initial configurations of twelve possible scenarios. They are classified according to which sides are allowed to initiate the negotiation and which sides change their preference lists. Our CPN model allows simultaneous negotiation and both sides can change their preference lists (Case 12).

5.2 Analysis Result

The analysis results of DCCP feature negotiation CPN model using the initial configurations described in the previous subsection are shown in Table 3. The total number of states, arcs in each case are shown in the second and third columns. Column 4, 5 and 6 show the terminal markings of each scenario. All terminal markings have both sides in STABLE and no packets left in the channels and hence there is no unspecified reception. The terminal markings are classified into 3 types. Type-I is the desired terminal state where both Client and Server reach the same feature value. Type-II is the undesired terminal state where both

Table 3. Analysis results of the CPN model

Case	nodes	arcs	Terminal Markings		
			Type I	Type II	Type III
(1)	(2)	(3)	(4)	(5)	(6)
1	4	3	1	0	0
2	4	3	1	0	0
3	20	26	1	0	0
4	19	22	2	1	0
5	10	11	2	0	0
6	106	169	2	1	0
7	10	11	2	0	0
8	19	22	2	1	0
9	106	169	2	1	0
10	50	77	3	1	0
11	52	78	3	2	0
12	553	1043	3	3	1

sides reach the different feature values but an endpoint knows that the agreed value is wrong. Type-III is also the the undesired terminal state where both sides reach the different feature values and both endpoints do not know that their feature values do not match.

5.3 Discussion

Figure 11 shows a scenario leading to a Type-II terminal state. Referring Fig. 11, after sending the first Change L Option, Client changes its preference list in UNSTABLE and sends the second Change L. When receiving the Confirm R of the first Change L in the CHANGING state, Client enters the STABLE state and then ignores the Confirm R of the second Change L. The agreed feature value in the first Confirm R is outdated and different from the feature value in Server. However when comparing the preference list in the first Confirm R option with the preference list in Client's state information, Client is able to know that the agreed value is wrong. Obviously in this case the Client should resend the Change option or reset the connection.

Figure 12 illustrates a scenario leading to an undesired Type-III terminal state. This is the center of our attention in this paper. This scenario could happen when both sides initiate the negotiation simultaneously and both sides change their preference list (Case 12). We notice that all Confirm options in Fig. 12 are discarded. It becomes one way communication with no acknowledgement. Figure 12 can be viewed as three attempts of negotiation. Two attempts are initiated simultaneously from both sides. This could be happened during DCCP simultaneous open procedure. Preference list changed in CHANGING

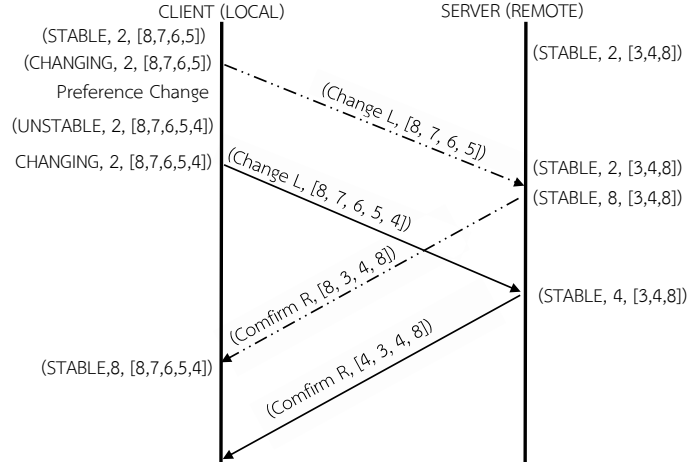


Fig. 11. A scenario leading to an undesired terminal marking Type-II.

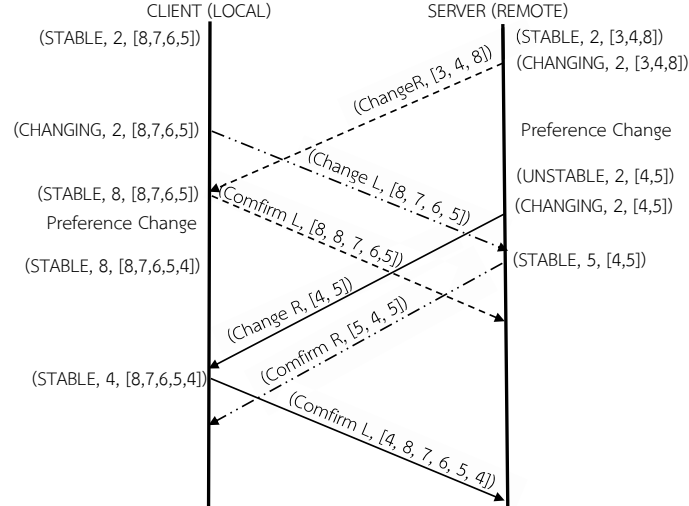


Fig. 12. A scenario leading to an undesired terminal marking Type-III.

state causes the third attempt of negotiation. All three calls do not receive any reply. The root of the problem is that the new preference list from the other side cannot pass through.

Type-III terminal state is worse than type-II because both entities are not aware that their agreed feature values are different. In our opinion the main objective of the DCCP feature negotiation protocol is to exchange the preference lists. After the preference list of the other side is known, the agreed feature value can be correctly computed. We suggest a solution when the preference list is

changed (either major or minor change), the endpoint shall send Change option to inform the other side. If the preference list is changed in the STABLE state, the endpoint shall send Change option and enter CHANGING state. Another solution could be that the endpoint does not discard Confirm option in STABLE state.

6 Conclusion and Future Work

This paper presents Coloured Petri Net model and analysis of DCCP feature negotiation procedure operating over FIFO with no loss channels. The analysis result shows that the protocol could fail to an undesired state (Type-III) where the feature values of both sides do not match and both sides are not aware of the mismatch.

Usually when the protocol operates over reordering and/or lossy channels, it is possible that the protocol could fail due to the channel imperfection. However if the protocol operates over the ideal channels (FIFO with no loss), the error indicates the flaw in the protocol itself.

The terminal state (Type-III) occurs when both sides change their preference lists during the simultaneous feature negotiation. Although the odds of this scenario is low, given the large number of potential connection in the Internet, we consider that this defect could be a serious threat.

The model development begins with a lot of assumptions. In the future we would like to relax these assumptions and refine the model. In particular we are interested to include connection management procedures together with Network Address Translators (NATs) into the model.

Acknowledgments This work is supported by Research Grant from the Thai Network Information Center Foundation and the Thailand Research Fund. The author is thankful to anonymous reviewers. Their constructive feedback has helped the author improve the quality of this paper.

References

1. F. Babich and L. Deotto. Formal Methods for the Specification and Analysis of Communication Protocols . *IEEE Communications Surveys*, 4(1):2–20, Third Quarter 2002.
2. J. Billington, M. Diaz, and G. Rozenberg (Eds.). *Application of Petri Nets to Communication Networks*, volume 1605 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 1999.
3. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer, Heidelberg, 2004.
4. CPN Tools home page. <http://cpntools.org>.

5. J. C. A. Figueiredo and L.M. Kristensen. Using Coloured Petri Nets to Investigate Behavioural and Performance Issues of TCP Protocols. In *Second Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN*, DAIMI PB-541, pages 21–40. Department of Computer Science, University of Aarhus, 11-15 October 1999.
6. S. Floyd, M. Handley, and E. Kohler. Problem Statement for the Datagram Congestion Control Protocol (DCCP), RFC 4336. Available via <http://www.rfc-editor.org/rfc/rfc4336.txt>, March 2006.
7. S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control, RFC 4341. Available via <http://www.rfc-editor.org/rfc/rfc4341.txt>, March 2006.
8. S. Floyd and E. Kohler. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 4: TCP-Friendly Rate Control for Small Packets (TFRC-SP), RFC 5622. Available via <http://www.rfc-editor.org/rfc/rfc5622.txt>, August 2009.
9. S. Floyd, E. Kohler, and J. Padhye. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC), RFC 4342. Available via <http://www.rfc-editor.org/rfc/rfc4342.txt>, March 2006.
10. S. Gordon. *Verification of the WAP Transaction Layer using Coloured Petri Nets*. PhD thesis, Institute for Telecommunications Research and Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, November 2001.
11. B. Han. *Formal Specification of the TCP Service and Verification of TCP Connection Management*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, December 2004.
12. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 2nd edition, 1997.
13. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science. Springer, Heidelberg, 1997.
14. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, Heidelberg, 2009.
15. E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proceedings of the 2006 ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'06)*, pages 27–38, Pisa, Italy, 11-15 September 2006.
16. E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol, RFC 4340. Available via <http://www.rfc-editor.org/rfc/rfc4340.txt>, March 2006.
17. L. M. Kristensen, J.B. Jørgensen, and K. Jensen. Application of Coloured Petri Nets in System Development. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets - Advanced in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 626–685. Springer, Heidelberg, 2004.
18. C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, June 2004.
19. University of Aberdeen, Electronics Research Group, School of Engineering. Background on Feature Negotiation. Available via http://www.erg.abdn.ac.uk/users/gerrit/dccp/notes/feature_negotiation/background.html.

20. University of Aberdeen, Electronics Research Group, School of Engineering. Why feature negotiation and protocol state machine are not independent. Available via http://www.erg.abdn.ac.uk/users/gerrit/dccp/notes/feature_negotiation/dependencies.html.
21. S. Vanit-Anunchai. *An Investigation of the Datagram Congestion Control Protocol's Connection Management and Synchronisation Procedures*. PhD thesis, Computer Systems Engineering Centre, School of Electrical and Information Engineering, University of South Australia, Adelaide, Australia, November 2007.
22. S. Vanit-Anunchai, J. Billington, and T. Kongprakaiwoot. Discovering Chatter and Incompleteness in the Datagram Congestion Control Protocol. In F. Wang, editor, *Proceedings of the 25th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2005)*, volume 3731 of *Lecture Notes in Computer Science*, pages 143–158, Taipei, Taiwan, 2–5 October 2005. Springer, Heidelberg.
23. Somsak Vanit-Anunchai. Analysis of Two-Layer Protocols: DCCP Simultaneous-Open and Hole Punching Procedures. In Christine Choppy and Jun Sun, editors, *1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)*, volume 31 of *OpenAccess Series in Informatics (OASICs)*, pages 3–17, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Dynamic Software Architecture for Distributed Embedded Control Systems

Tomáš Richta, Vladimír Janoušek, Radek Kočí

Brno University of Technology, Faculty of Information Technology,
IT4Innovations Centre of Excellence
Božetěchova 2, 612 66 Brno, The Czech Republic
{irichta,janousek,koci}@fit.vutbr.cz

Abstract. This paper focuses on the field of dynamically reconfigurable distributed embedded control systems construction process and presents a substantial part of the methodology aimed at this application area which is based on formal models, namely some variants of Petri Nets. Initial system specification is represented by a set of Workflow Petri Nets transformed into decomposed multi-layered Reference Petri Nets model, that is used during the generation of interpretable target system components representation. The main objective of presented approach is the introduction of dynamic reconfigurability features into the target system implementation reflecting changes in system specification during its run-time. Reconfigurability is achieved by the system decomposition into smaller interpretable pieces of computation that are installed on and performed by the underlying infrastructure. Introduced approach brings several layers of reconfigurability through a set of specific translation rules applied in different layers and scenarios for pseudo-code generation and by the possibility of installing the resultant functional parts on system nodes using well-defined communication protocol. The heart of described architecture lies within the specification of hosting platform called Petri Nets Operating System (PNOS) that includes the Reference Petri Nets interpreter.

Keywords: Dynamic Reconfigurability, Embedded Systems, Control Systems, Model-Driven Development, Model Transformation, Model Execution, Workflow Petri Nets, Reference Petri Nets

1 Introduction and Motivation

Control systems lie on the thin border between physical and information worlds. The process of control is usually described as a loop switching between reading data from sensors and triggering a number of actuators installed within the physical environment. Above all, the process should respect all user-defined rules. Control systems could be constructed as a set of programmable logic controllers with proprietary software installation, communicating with each other, thus forming distributed embedded control system. Our work considers a target platform for this type of systems implementation to be a set of minimalistic and

low energy consumption hardware devices, e.g. ATmega, PIC, or ARM micro-controllers, equipped with wireless transmission modules. Such devices are often used in the area of Wireless Sensor Networks (WSN) systems.

Usually a hardware part of any system implementation starts with selection of proper set of devices and their installation within the physical environment, including sensors and actuators attachment. The software part of system implementation complements the hardware one with the construction of appropriate application software, that controls each system unit and represents the whole system functionality. Dynamic reconfigurability features are necessary for the ability of the system to adapt itself to changes in environment and also to provide its maintainer with a possibility to change the system behaviour, while it is in runtime, i.e. without the necessity of complete destruction and further reconstruction, or even restart. Our main goal is to describe the software part of the process, that respects our focus on formal specification and dynamic reconfigurability.

In this paper, we are going to describe some recent results of the research in the field of dynamically reconfigurable distributed embedded control systems, and basic ideas of our research that aims to introduce complete methodology for control systems construction and administration, which uses formal and human readable notation as a system functionality specification, and provides the user of resulting system with the possibility to change its behaviour within the runtime. Introduced solution to the dynamic reconfigurability problem follows the model transformation and executable model paradigms - Workflow Petri Nets[1] are used as an abstract system specification modelling language, and the MULAN-like[5] multi-layered Reference Petri Nets[3] structure for modelling the resultant system implementation. The system run-time model is constructed from the work-flow one using graph transformations and then translated into the executable form, run by our specialized target platform.

2 Related Work

Related work could be divided into the following areas - embedded and operating systems, software engineering methods applied to the area of embedded systems, the usage of higher-level or visual languages for embedded systems specification and implementation, the dynamical reconfigurability within embedded systems, reconfigurable control systems (e.g. FMSs), multi-agent approach to the reconfigurable embedded systems development, system partitioning, code generation, and reconfigurable hardware.

The usage of formal modelling control system with dynamic reconfigurability features is not a new idea. Research activities in this topic are primarily focused on direct or indirect approach. The direct approach offers specific functions or rules, allowing to modify system structure, whereas the indirect approach introduces mechanisms allowing to describe system reconfigurations. The main difference consists usually in the level of reconfigurability implemented. Direct methods use formalisms containing intrinsic features allowing to reconfigure the

system. Indirect methods use specific kind of frameworks or architectures, that make possible to change the system structure.

In our field of research the first group consists of formalisms based usually on some kind of Petri nets. Reconfigurable Petri Nets [11], presented by Guan and Lim, introduced a special place describing the reconfiguration behaviour. Net Rewriting System [12] extends the basic model of Petri Nets and offers a mechanism of dynamic changes description. This work has been improved [13] by the possibility to implement net blocks according to their interfaces. Intelligent Token Petri Nets [14] introduces tokens representing jobs. Each job reflects knowledge about the system states and changes, so that the dynamic change could be easily modelled. All the presented formalisms is able to describe the system reconfiguration behaviour, nevertheless only some of them define the modularity. Moreover, the study [15] shows, that the level of reconfigurability is dependent on the level of modularity and also that there are modular structures that are not reconfigurable.

The second group handles reconfigurations using extra mechanisms. Model-based control design method, presented by Ohashi and Shin [16], uses state transition diagrams and general graph representations. Discrete-event controller based on finite automata has been presented by Liu and Darabi [17]. For reconfiguration, this method uses mega-controller, a mechanism, which responses to external events. Real-time reconfigurable supervised control architecture has been presented by Dumitrache [18], allowing to evaluate and improve the control architecture. All the presented methods are based on an external mechanism allowing system reconfiguration. Nevertheless, most of them do not deal with validity and do not present a compact method.

So far, we have investigated formalisms and approaches to the control system development. They have one common property, they are missing complex design and development methods analogous to software engineering concepts. Of course, the methods and tools that are applied in ordinary software systems are not as simply applicable to embedded systems. Nevertheless, we can be inspired with software engineering approaches and adopt them to the embedded control systems [19]. To develop embedded control system, the developer has to consider several areas. We can distinguish five areas [19] as follows—*Hardware*, *Processes* (development processes and techniques), *Platform* (drivers, hardware abstraction, operating systems), *Middleware* (application frameworks, protocols, message passing), and *Application* (user interface, architecture, design patterns, reusing).

Presented approach is based on existing formalisms and architecture that are together used in the specific platform developed by our team. In relation to previously defined areas of software engineering for embedded control systems, we deal with *Process*, *Platform*, and *Middleware* areas in this paper. *Process* area focuses on work-flow modelling using Petri Nets, transformation of the work-flow models into the Reference Nets, and definition of the levels of abstraction. *Middleware* area focuses on multi-layered architecture inspired by the MULAN architecture. *Platform* area introduces Petri Nets Operating System (PNOS)

linked with Petri Net Virtual Machine (PNVM) that offer specific means for system reconfigurability. All mentioned elements will be described in details in next chapters.

3 Formalisms and Tools

3.1 Workflow Modelling

Work-flow modelling is very popular for its aim to precisely define the functionality requirements using intuitive and human-readable form, while offering enough precision to be interpretable by machines. For its formal and verifiable characteristics and large research background we adopted for the purposes of our research Wil van der Aalst's specification for system work-flow modelling, so called Extended Workflow Petri Nets[1]. Aalst's work is well-defined and resulting work-flow models could be used for the system processes verification and validation purposes. This way is very similar to the BPMN work-flow models, so it might be easily used by the business process modelling domain experts. For that reason we decided to use the Aalst's YAWL notation[2] and Workflow Petri Nets formalism[1] in the early beginning of system construction process. The main advantage of using Workflow Petri Nets is the possibility of system specification and its adaptation by the non-technically educated domain specialists.

3.2 Reference Nets

Second step of the system construction process consists of the transformation of Workflow Petri Nets model into the multi-layered Reference Nets model complying with the nets-within-nets concept defined by Rüdiger Valk [3] and formalized as Reference Nets by Olaf Kummer [4]. In our proposed system development methodology, Reference Nets are translated into the interpretable form, that is transferred through the network to the specific nodes, responsible of its execution. The problem of generating the code from formal specification to its runnable form is mainly based on the decomposition of the whole system model to a set of sub-models, that is usually called the partitioning problem. We use similar concept to the MULAN architecture defined by Cabac et al.[5]. This architecture divides the model into four levels of abstraction - infrastructure, agent platform, agents, and protocols. Our architecture also uses four layers - infrastructure, platform, processes and sub-process. Each of these layers is mapped from the formal specification to the target platform specification.

4 Reconfigurable Architecture

Reference Nets allows to construct the system hierarchically, in several layers of abstraction. Each element of layer at any level of abstraction could be changed by change in nets marking. Nets representing system functionality are migrating

over nets of other layers changing the system functionality. The multi-layered nature of the system and responsibilities of particular levels of system decomposition is described in more detail in our previous work[10].

The core characteristics of resulting system, its dynamic reconfigurability, is based in our solution on the ability of Reference Petri Nets interpretable representations to migrate among places of the system as tokens, similarly as in reference Nets. The new or modified Petri Net, that represents the system partial behaviour change could be sent over other Petri Nets to its destination place to change the whole system functionality. In our solution, these Petri Nets parts are maintained by the Petri Nets Operating System (PNOS) and interpreted by the Petri Nets Virtual Machine (PNVM) engine[8]. System decomposition is inspired by MULAN architecture [5]. The PNOS contains PNVM (Petri Net Virtual Machine) engine that interprets Petri Nets which are installed within the system in the form of a interpretable byte-code called Petri Nets ByteCode (PNBC). PNOS also provides the installed processes with the access to input and output of the underlying hardware that is connected to sensors and actuators, and also with the serial communication port that is connected to the wired or wireless communication module (e.g. ZigBee)[8], or Ethernet interface.

The important net (lying above processes nets) interpreted in PNOS is so called Platform net. Platform net is responsible for the interpretation of commands which are read from buffered serial line, or Ethernet. These commands allow to install, instantiate, and uninstall other Petri Nets. The Platform also allows to pass messages to the other layers, which are responsible for application-specific functionality. Since we need reconfigurability in all levels, the installation and uninstallation of functionality is implemented in each level of resulting system. Next section describes the Reference nets formalism that is used as an intermediate language for the target implementation.

5 The Development Process

System development process is described in Fig. 1. It starts with the specification of the whole system work-flow, in an hierarchical way. Work-flow model is transformed to the Reference Nets layered architecture and might be further simulated and debugged using the Renew Reference Nets tools [6]. After this stage, the final set of Reference Nets is then translated into the Petri Nets ByteCode (PNBC) that is used either for the target prototype simulation using SmallDEVS tools [7] and also to be transferred to the nodes of the system infrastructure. More detailed description of the whole PNOS architecture and functionality could be found in [8].

5.1 Model Transformations

There are two translation phases. The translation of the Workflow Petri Nets model into the Reference Petri Nets model and translation of the Reference Petri Nets model into its interpretable form. The first transformation phase takes into

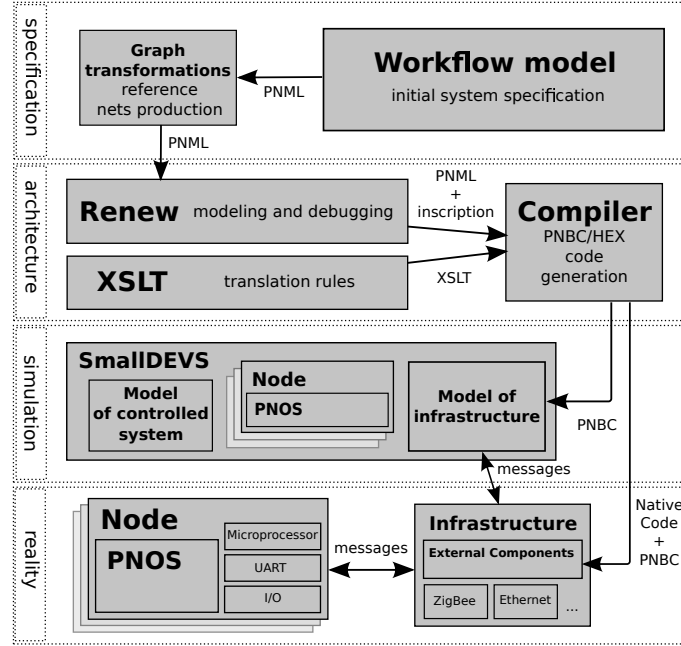


Fig. 1. System construction process

account the set of work-flow specifications described within the work-flow model of the system and produces target node representations. Such a representation should contain the basic PNOS I/O functionality, and the platform functionality, which means the ability of receiving nets specifications, nets instantiation, removing nets instances, removing nets specifications, etc. Using this functionality the node main processes should be installed. It usually consists of the description of sub-processes interactions and ordering. Then the main processes of each node are installed with translated sub-processes. The communication between resources is represented by transitions, that are not part of any other role and serve as a data transport part of the system. Particular data types should be described in the terms dictionary, that holds all the necessary information needed for nets translation, that is not included within the diagram. Regarding the work-flow model, also other specific rules for the communication protocol could be derived.

5.2 Basic Definitions

Let us introduce some basic definitions of formalisms used during the system development. As a basement the classical Petri nets definition describes the main rules of the specification formalism.

Definition 1 (Petri Net). A Petri net is a triple $PN = (P, T, F)$ where:

- P and T are disjoint finite sets of places and transitions, respectively and
- $F \subseteq (P \times T) \cup (T \times P)$ is a binary relation called the flow relation representing arcs of the net.
- $\bullet x = y | yFx$ is called input set (preset) of the element x and
- $x^\bullet = y | xFy$ is called output set (postset) of the element x , where $x \in P \cup T$.

Van der Aalst's extensions to Petri Nets add two basic conditions to the nets construction. Our modelling approach is very similar, so we can use his definition, but for the further transformation of models we need some more rules to be added. First let us introduce the simple work-flow net definition.

Definition 2 (Workflow Net). A Petri net $PN = (P, T, F)$ is a WF-net (Workflow net) if and only if [1]:

- PN has two special places: $i \in P$ and $o \in P$. Place i is a source place: $\bullet i = \emptyset$. Place o is a sink place: $o^\bullet = \emptyset$.
- If we add a transition t^* to PN which connects place o with i (i.e. $\bullet t^* = \{o\}$ and $t^{*\bullet} = \{i\}$), then the resulting Petri net is strongly connected.

Some other simplification rules added by Aalst and Hofstede extended workflow models to provide for better human-readability. Some special types of transitions representing logical operators and some special operations for manipulation with tokens were added. Transitions and places are considered to be tasks and conditions. Each EWF-net consists of tasks (either composite or atomic) and conditions which can be interpreted as places. Tasks in elementary form are atomic units of work, and in compound form modularize an execution order of a set of tasks. In contrast to Petri nets, it is possible to connect “transition-like objects” like composite and atomic tasks directly to each other without using a “place-like object” (i.e., conditions) in-between[2].

Definition 3 (Extended Workflow Net). An extended work-flow net (EWF-net) is a tuple $EWf = (C, i, o, T, F, S, name, split, join, rem, nofi)$ such that [2]:

- C is a set of conditions,
- $i \in C$ is the input condition,
- $o \in C$ is the output condition,
- T is set of tasks,
- $F \subseteq (C \setminus \{o\} \times T) \cup (T \times C \setminus \{i\}) \cup (T \times T)$ is the flow relation,
- every node in net graph $(C \cup T, F)$ is on a directed path from i to o ,
- $split : T \rightarrow \{AND, XOR, OR\}$ specifies the split behaviour of each task,
- $join : T \rightarrow \{AND, XOR, OR\}$ specifies the join behaviour of each task,
- $rem : T \rightarrow \mathbb{P}(T \cup C \setminus \{i, o\})$ specifies the additional tokens to be removed by emptying a part of the work-flow, and
- $nofi : T \rightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ specifies the multiplicity of each task (minimum, maximum, threshold for continuation, and dynamic/static creation of instances).

Our approach follows the previous definitions and adds some more rules to enable the extended work-flow models with communication features to satisfy the developer ability to combine multiple work-flow specifications.

Definition 4 (Extended Communicating Workflow Net). *We call Extended Communicating Workflow net $ECWF = (EWF, I, O, F^C)$ a EWF net that has following properties:*

- EWF is an extended work-flow net,
- I is a set of $ECWF$ input places, where $\forall p_I \in I : \bullet p_I = \emptyset \wedge p_I \neq i$,
- O is a set of $ECWF$ output places, where $\forall p_O \in O : p_O^\bullet = \emptyset \wedge p_O \neq o$,
- F^C is a communication flow $F^C \subseteq (I \times T) \cup (T \times O)$,
- $I \cup P_{EWF} = \emptyset \wedge O \cup P_{EWF} = \emptyset$.

To specify complete work-flow model a definition of Workflow Specification was introduced by Aalst and Hofstede. We adopted this definition and added some slight change to one of the rules.

Definition 5 (Workflow Specification). *A Workflow Specification S is a n -tuple (Q, top, T°, map) such that:*

- Q is a set of $ECWF$ -nets,
- $top \in Q$ is the top level work-flow[2],
- $T^\circ = \cup_{N \in Q} T_N$ is the set of all tasks[2],
- $\forall N_1, N_2 \in Q, N_1 \neq N_2 \Rightarrow (C_{N_1} \cup T_{N_1}) \cap (C_{N_2} \cup T_{N_2}) = \emptyset$, i.e., no name clashes[2],
- $map : T^\circ \rightarrow Q \setminus \{top\}$ is a surjective injective (bijective) function which maps each composite task onto a EWF net[2], and
- the relation $\{(N_1, N_2) \in Q \times Q \mid \exists t \in dom(map_{N_1}) map_{N_1}(t) = N_2\}$ is a tree[2].

And also some special types of tasks representing composite and multi-instance tasks were added by Aalst and Hofstede.

Definition 6. *Whenever we introduce a work-flow specification $S = (Q, top, T^\circ, map)$, we assume $T^A, T^C, T^{SI}, T^{MI}, C^\circ$ to be defined as follows [2]:*

- $T^A = \{t \in T^\circ \mid t \notin dom(map)\}$ is the set of atomic tasks,
- $T^C = \{t \in T^\circ \mid t \in dom(map)\}$ is the set of composite tasks,
- $T^{SI} = \{t \in T^\circ \mid \forall N \in Q, t \in dom(nofi_N)\}$ is the set of single instance tasks,
- $T^{MI} = \{t \in T^\circ \mid \exists N \in Q, t \in dom(nofi_N)\}$ is the set of (potentially) multiple instance tasks, and
- $C^\circ = \cup_{N \in Q} C_N^{ext}$ is the extended set of all conditions.

Final definition describes the Workflow System consisting of set of Extended Communicating Workflow Specifications and communication transitions.

Definition 7 (Workflow System). *Let us call Workflow System the triple $WS = (\hat{S}, T^{WS}, F^{WS})$, where:*

- \hat{S} is non-empty finite set of extended communicating work-flow specifications,

- T^{WS} is a finite set of communication transitions,
- $F^{WS} \subseteq (O^{WS} \times T^{WS}) \times (T^{WS} \times I^{WS})$ is a system communication flow relation, where $O^{WS} = \bigcup_{O_{Si} i \in \langle 1, \dots, n \rangle}$ is a set of all extended communicating work-flow specifications output places and, $I^{WS} = \bigcup_{I_{Si} i \in \langle 1, \dots, n \rangle}$ is a set of all extended communicating work-flow specifications input places.

Target system representation for the first phase of system model transformation is constructed as a set of Reference Nets based on Valk's nets-within-nets paradigm that is formalized as an Elementary Object System which consists of elementary net systems (EN System) $EN = (B, E, F, C)$, which is defined as finite set of places B , finite set of transitions E , disjoint from B , a flow relation $F \subseteq (B \times E) \cup (E \times B)$ and an initial marking $C \subseteq B$ [3].

Definition 8 (Elementary Object System). An elementary object system is a n -tuple $EOS = (SN, \widehat{ON}, Rho, type, \widehat{M})$ where [3]:

- $SN = (P, T, W)$ is a Petri net, called system net of EOS,
- $\widehat{ON} = \{ON_1, \dots, ON_n\} (n \geq 1)$ is a finite set of EN systems, called object systems of EOS, denoted by $ON_i = (B_i, E_i, F_i, m_{0i})$, which is either elementary net system or a system net of embedded EOS,
- $Rho = (\rho, \sigma)$ is the interaction relation, consisting of a system/object interaction relation $\rho \subseteq T \times E$ where $E := \bigcup \{E_i | 1 \leq i \leq n\}$ and symmetric object/object interaction relation $\sigma \subseteq (E \times E) \setminus id_E$,
- $type : W \rightarrow 2^{\{1, \dots, n\}} \cup \mathbb{N}$ is the arc type function, and
- \widehat{M} is a marking defined in following definition.

Definition 9 (System Marking). The set $Obj := \{(ON_i, m_i) | 1 \leq i \leq n, m_i \in R(ON_i)\}$ is the set of objects of the elementary object system. An object-marking (O -marking) is a mapping $\widehat{M} : P \rightarrow 2^{Obj} \cup \mathbb{N}$ such that $\widehat{M}(p) \cap Obj \neq \emptyset \Rightarrow \widehat{M}(p) \cap \mathbb{N} = \emptyset$ for all $p \in P$.

Next paragraphs are going to describe both transformation process phases. The first one is the transformation of the work-flow model into the operational nets-within-nets model, second one the transformation of the nets-within-nets model into its interpretable form, reflecting the target PNOS platform.

5.3 From Workflow Nets to Reference Nets

We decided to describe our methods on the sample home automation example. The whole system functionality is described in the form of work-flow model in our approach represented by the Workflow System depicted in Fig. 2. There are following elements within the work-flow models - places, transitions, and logical transitions[1], sub-process transitions[1], connecting arcs, and system nodes borders. Places could be named, when there is a name on the place it is further considered as an variable name. Transitions could be also named. The named transition represents calling some particular atomic function of the underlying PNOS.

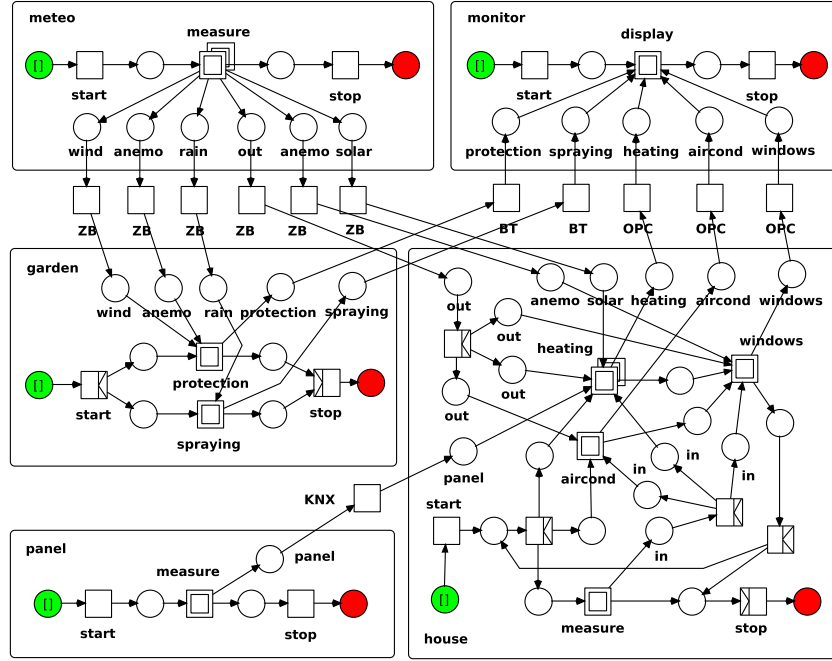


Fig. 2. Workflow System net

Logical transitions are: AND-split, AND-join, OR-split, OR-join, and AND/OR-split, they simplify the model to be easily readable for the non-technically educated domain experts. Sub-process transitions represent condensed parts of the system, that are described in another diagram, e.g. in Fig. 3.

Generating the Infrastructure layer Work-flow model of the intended system is translated into multi-layered Reference Nets model. Each layer of the Reference Nets model is generated separately using different production rules. First part of the system, that should be generated from the original model is the top level Infrastructure layer net, that describes the communication among all nodes of the system and could be used as a sort of deployment diagram. Infrastructure layer is a basic layer of the Reference Nets model and serves for the validation purposes and also as a description of the distribution of target system structure. Basically the main purpose of Infrastructure layer lies in description of the system nodes and their communication.

Within the Infrastructure layer, each node is represented as a place in which the particular Platform layer net is located. If there is any communication between nodes, this communication is represented as a transition between corresponding nodes. For example model described in Fig. 2 should be translated into the Infrastructure net described in Fig. 4. This layer is produced by the following set of rules.

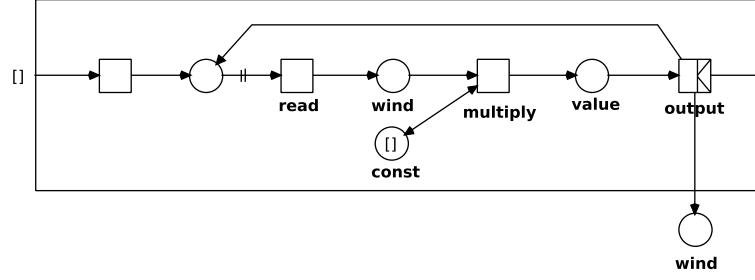


Fig. 3. Measure subprocess

Let $WS = (\hat{S}, P^{WS}, T^{WS}, F^{WS})$ be a Workflow System which has to be transformed and $SN = (P^I, T^I, W^I)$ a system net representing the Infrastructure layer of the target elementary object system should be generated using Algorithm 1.

Algorithm 1

(* demonstrates infrastructure net construction *)

1. $P^I = T^I = W^I \leftarrow \emptyset$
2. for each work-flow specification produce place in the system net, $\forall s \in \hat{S} : P^I = P^I \cup \{p_{name(s)}\}$
3. for every set of the communication transitions with the same name, place one transition to the system net, $\forall \xi(t) \in \chi(T^{WS}) = [\chi(T_i^{WS})]_{i \in \{1, \dots, n\}} : T^I = T^I \cup \{t_{name(\xi(t))}\}$, where $name(\xi(t_i)) = name(\xi(t_j)) (i \neq j)$
4. connect all communication transitions to the corresponding places with double-sided arcs, $\forall p_I \in P^I, \forall t_I \in T^I : p_I^I \in \bullet t_I^I \wedge p_I^I \in t_I^I \bullet$, where $t^{WS} \in T^{WS} : \forall p_i^{WS} \text{ in } C_i : p_i^{WS} \in \bullet t^{WS} \vee p_i^{WS} \in t^{WS} \bullet$
5. annotate all arcs with arbitrary names
6. place inscriptions to the transitions that invoke the $: output$ up-link in the source node and places the result to the $: input$ up-link of all the target nodes

Each node of the system, placed logically within the Infrastructure net place is considered to run on some piece of hardware installed with the PNOS. Because PNOS also consists of the PNVM it is able to interpret Reference Nets translated into the PNBC pseudo-code. Basic layer of the system, that must be installed on all nodes of the system is Platform layer, that brings a set of basic meta-operations that enables the node with other Reference Nets manipulation means - like loading, unloading nets, passing values, etc. This layer is described in Fig. 5. After the Platform layer was installed on the basic PNOS and become interpreted by the PNVM kernel, it is possible to send to it some other nets to define or modify the node behaviour. Basic types of such nets are Processes and Sub-processes of the target system.

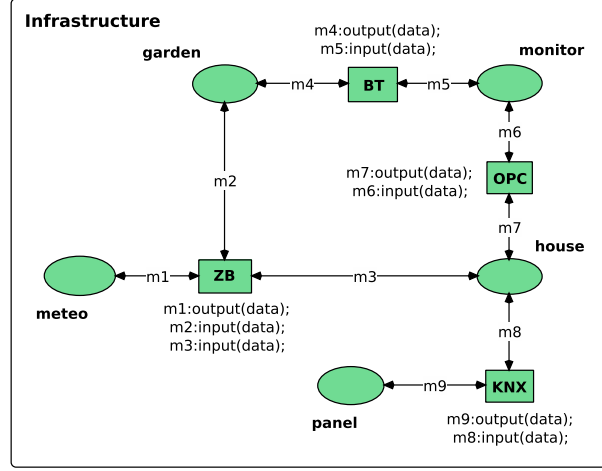


Fig. 4. System Infrastructure net

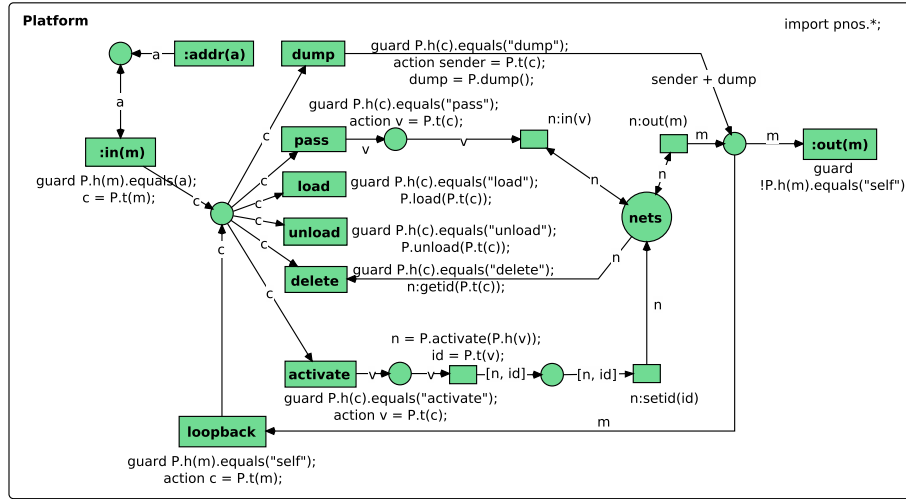


Fig. 5. Platform net

Generating the Process layer The translation of Processes layer also has its own set of production rules. When translating the work-flow model, there is at least one process net generated for each Workflow Specification within the the system model. Main process net consists of the set of meta-operations, that enable the main process to receive and run new nets definitions, and to pass the received values to running subnets. Input place is used for receiving the data by : *input* up-link. Output place serves as an buffer for the : *output* up-link. Nets place then stores all sub-process nets. During the main process life-cycle, each sub-process net is taken from the nets place, it is started, or served with parameters and started. Started net is then put back to nets place, where it resides, until the result is produced. When the result is ready, the net is taken from the temporary place again, the output result is taken, and the net is then stored again back to the nets place, or it could be stopped. The result of the net is then propagated according to the logic specified in the main process net. The example of translating the *garden* node main process net is shown in Fig. 6.

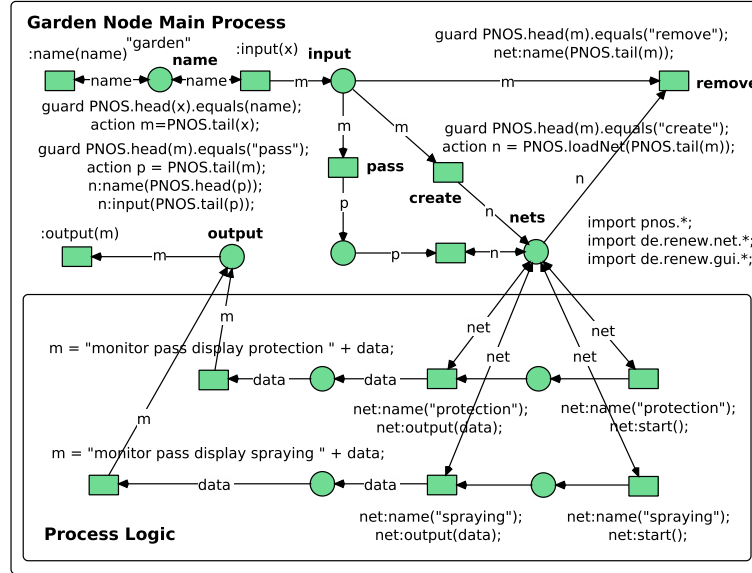


Fig. 6. Garden Main Process net

All the process nets should be produced according to the following rules. Let $S_i = (Q, top, T^\diamond, map)$ be a Workflow Specification to be transformed and $ON_i = (P_i^P, T_i^P, W_i^P)$ a net of the Processes layer of the target system. For the translation following Algorithm 2 should be used.

Algorithm 2

(* demonstrates process nets construction *)

1. $P_i^P = T_i^P = W_i^P \leftarrow \emptyset$
2. add nets, input and output places, $P_i^P = P_i^P \cup \{p_{nets}, p_{in}, p_{out}\}$
3. add the platform meta-operations, $T_i^P = T_i^P \cup \{t_{name}, t_{pass}, t_{create}, t_{remove}\}$
4. for each sub-process in swim-lane construct the first transition that takes the subnet from the nets place and invokes the : *start* up-link and a transition that triggers the : *output* up-link, $\forall t_{S_i} \in T^{top} : T_i^P = T_i^P \cup \{t_{i(start)}^P, t_{i(out)}^P\}$, where $\bullet t_{i(start)}^P = p_{nets} = t_{i(start)}^{\bullet} \wedge \bullet t_{i(out)}^P = p_{nets} = t_{i(out)}^{\bullet}$
5. connect both transitions with synchronization place and corresponding arcs, $\forall t^C \in T_i^C$, where $\exists p \in P, t \in T_i \subset T \setminus \bigcup \{T_i\} : p \in t^C \bullet \wedge p \in t^{\bullet} : P_i^P = P_i^P \cup \{p_i^P\}$, where $t_{i(start)}^{P\bullet} = p_i^P = \bullet t_{i(out)}^P$
6. add one more place for each output communication to store the results of the sub-process, $P_i^P = P_i^P \cup \{p_i^P\} : t_{i(start)}^{P\bullet} = p_i^P$
7. if the output is to be sent to another node add the transition that constructs the message and puts the resulting message into the output sink, $T_i^P = T_i^P \cup \{t_i^P\} : \bullet t_i^P = p_i^P \wedge t_i^{P\bullet} = p_{out}$
8. translate special transitions according to the rules defined by Aalst [1]
9. omit input places
10. copy left places, $\forall c \in C : P_i^P = P_i^P \cup c_c^P$
11. copy left transitions, $\forall t \in T : T_i^P = T_i^P \cup t_t^P$

Generating the Sub-process layer Within the house work-flow model, there is a measure sub-process used in *meteo* and *house* modules. This sub-process should be translated to the Sub-process layer using Algorithm 3.

Algorithm 3

(* demonstrates sub-process nets construction *)

1. for all sub-process places produce corresponding places, $\forall c \in C : P_i^P = P_i^P \cup c_c^P$
2. for all sub-process transitions produce corresponding transitions, $\forall t \in T : T_i^P = T_i^P \cup t_t^P$
3. translate special transitions according to rules defined by Aalst [1]
4. if there's a loop, switch the do-while-do loop to the while-loop and add the while condition place to the beginning of loop and add the : *stop* transition to enable removing the condition, search for the transitions inscriptions within the dictionary - transition producing the values and transitions consuming the values

Resulting sub-process net is described in Fig. 7.

5.4 From Reference Nets to Petri Nets Byte Code

Following part of the development process comprises of target system code generation. In our approach, each layer of the system should be compiled to target code independently. All generated levels communicate with each other using up-links and down-links.

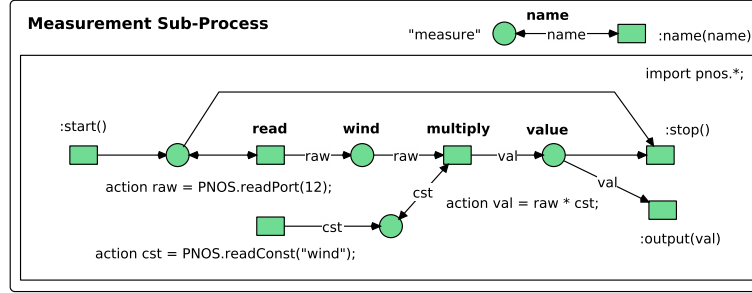


Fig. 7. Measure Sub-process net

The only part of the system, which is implemented natively, is the PNOS kernel, including PNVM [8]. The example of byte-code follows. It represents the measure net (depicted in Fig. 7). In fact, it is a human-readable version of the byte-code. In this representation, numbers are represented as text and also some spaces and line breaks are added. This means that the contents of the code memory is a bit more condensed. Each byte of the code is either an instruction for PNVM, or data.

```

(Nmeasure
(measure/wind)
(cond/wind/cst/value/name)
(Ustart() () (P1(B1) (V1)))
(Ustop() () (O1(B1) (V1)))
(Uoutput(val) () (P4(B1) (V1)))
(Uname(name) () (P5(B1) (V1)))
(I 05(B1) (S1)))
(Tread(cond/raw)
(P1(B1) (V1))
(A(: (V2) (r(S2))))
(O2(B1) (V2)))
(Tconst(cst)
(A(: (V1) (r(S2))))
(O2(B1) (V1)))
(Tmultiply(raw/cst/val)
(P2(B1) (V1))
(P3(B1) (V2))
(A(: (V2) (/(* (V1) (V2)) (I10000))))
(O4(B1) (V2)))

```

The important feature of the system is its reconfigurability. It is based on operations of the operating system that are designated for manipulations with nets (in the form of PNBC) and their instances. Nets could be sent to a node as a part of the command for its installation. The command is executed by Platform net. Using other commands, the platform can instantiate a net, pass a command

to it, destroy a net instance and unload a net template - see Fig. 5. The PNOS Platform functionality is described in more detail in [8], [9], [10].

6 Installation and Reconfiguration

The main operating principle of resulting system could be described on the tasks of system construction - installation, and its reconfiguration. The installation of the system starts with placing proper nodes to the target environment. Each node should be installed with the PNOS, PNVM and basic platform layer. The physical communication between nodes using different wired or wireless communication technologies should be established. In our running example the scenario should start with installing the processes for each Workflow Specification and then sending particular sub-processes nets to relevant nodes.

```
meteo load measure-wind
meteo create mw1 measure-wind
meteo load measure-anemo
meteo create ma1 measure-anemo
...
meteo start
meteo pass mw1 start
meteo pass ma1 start
...
```

The other important part of system functionality is its reconfiguration. It should be performed on each defined level of the system architecture. Basically, the node firmware including the PNOS and PNVM could be reprogrammed and rebuilt and then sent over the air to the particular node. The Platform net could be modified and also sent to the particular node, but usually we do not expect this layer to be modified often. The next level of reconfiguration is the processes layer. All processes of the node could be changed and then passed to its platform to change the behaviour of the node. Finally all the sub-processes nets could be modified and sent to particular nodes processes that reinstall them within the nets place. The example of the reconfiguration process follows.

```
meteo pass mw1 stop
meteo destroy mw1
meteo unload measure-wind
meteo load measure-wind
meteo create mw1 measure-wind
meteo pass mw1 start
...
```

There is a plan in future to add the pause and resume operations to the platform, to be able to pause any particular net instance, change its template and resume then. For that it is necessary to invent, how to represent the pausing and resuming conditions in Petri Nets, that is not part of this material.

7 Conclusion

We described the basics of model transformation and execution-based methodology of distributed embedded control system development. Among the main methods it uses Petri Nets models transformations and target system prototype code generation. Development process starts with the work-flow model of the system specification defined according to the rules of Van der Aalst's Workflow Specifications. Work-flow model of the system describes the functionality from user's or domain specialist's point of view. Using our methods, the work-flow model is further transformed to the multi-layered architecture based set of Reference Petri Nets. Each layer of the system is then translated to the specific target representation called Petri Nets ByteCode (PNBC), which is interpreted by the Petri Nets Virtual Machine (PNVM), that is a part of the Petri Nets Operating System (PNOS), that is installed on all nodes of the system. Targeted dynamical system reconfigurability is achieved by the possibility of PNBC net templates and instances replacement with its new versions. After the replacement, PNVM interpretation engine starts to perform a new version of partial functionality of the system. That makes the dynamic reconfigurability possible.

Acknowledgement

This work has been supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by BUT FIT grant FIT-11-1, and by the Ministry of Education, Youth and Sports under the contract MSM 0021630528.

References

1. Van der Aalst, W.M.P., Van Hee, K.M. 2002. Workflow Management: Models, Methods, and Systems. IT press, Cambridge, MA.
2. Van der Aalst, W.M.P., Ter Hofstede, A.H.M. 2005. YAWL: yet another workflow language. *Inf. Syst.* 30, 4 (June 2005), p. 245-275.
3. Valk, R. 1998. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. *Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN '98)*, Jörgen Desel and Manuel Silva (Eds.). Springer-Verlag, London, UK, p. 1-25.
4. Kummer, O. 2001. Introduction to Petri nets and reference nets. *SozionikAktuell* 1:2001 / Rolf von Lüde, Daniel Moldt, Rüdiger Valk (Hrsg.).
5. Cabac, L., Duvigneau, M., Moldt, D., Rölke, H. 2005. Modeling dynamic architectures using nets-within-nets. *Proceedings of the 26th international conference on Applications and Theory of Petri Nets (ICATPN'05)*, Gianfranco Ciardo and Philippe Darondeau (Eds.). Springer-Verlag, Berlin, Heidelberg, p. 148-167.
6. Kummer, O., Wienberg, F., Duvigneau, M., Köhler, M., Moldt, D., and Rölke, H. 2003. Renew - the Reference Net Workshop. *Tool Demonstrations. Eric Veerbeek (ed.). 24th International Conference on Application and Theory of Petri Nets (ATPN 2003)*.

7. Janoušek, V., Kironský, E. 2009. Interactive evolutionary modelling and simulation of discrete-event systems using prototypical objects. *International Journal of Autonomic Computing*. London: Inderscience Publishers, 2009, Vol. 1, Iss. 2, p. 104-120. ISSN 1741-8569.
8. Richta, T., Janoušek, V. 2013. Operating System for Petri Nets-Specified Reconfigurable Embedded Systems. *Proceedings of Computer Aided Systems Theory - EUROCAST 2013*, published as LNCS 8111. Berlin Heidelberg: Springer Verlag, 2013, p. 444-451. ISBN 978-3-642-53855-1.
9. Richta, T., Janoušek, V., Kočí, R. 2012. Code Generation For Petri Nets-Specified Reconfigurable Distributed Control Systems, *Proceedings of 15th International Conference on Mechatronics - Mechatronika 2012*, Prague, CZ, FEL Č, 2012, p. 263-269. ISBN 978-80-01-04985-3.
10. Richta, T., Janoušek, V., Kočí, R. 2013. Petri nets-based development of dynamically reconfigurable embedded systems. In Daniel Moldt and Heiko Rölke, editors, *Petri Nets and Software Engineering. International Workshop, PNSE'13*, Milano, Italy, June 24-25, 2013. *Proceedings*, volume 989 of *CEUR Workshop Proceedings*, pages 203-217. CEUR-WS.org, 2013.
11. Guan, S., U., Lim, S., S. 2004. Modeling adaptable multimedia and self-modifying protocol execution. *Future Gener. Comput. Syst.*, Vol. 20, Iss. 1, p. 123-143.
12. Llorens, M., Oliver, J. 2004. Structural and dynamic changes in concurrent systems: Reconfigurable Petri Nets. *IEEE Transactions on Automation Science and Engineering*, Vol. 53, Iss. 9, p. 1147-1158.
13. Li, J., Dai, X., Meng, Z. 2009. Automatic reconfiguration of Petri net controllers for reconfigurable manufacturing systems with an improved net rewriting system based approach. *IEEE Transactions on Automation Science and Engineering*, Vol. 6, Iss. 1, p. 156-167.
14. Wu, N., Q., Zhou, M., C. 2011. Intelligent token Petri nets for modelling and control of reconfigurable automated manufacturing systems with dynamic changes. *Transactions of the Institute of Measurement and Control*, Vol. 33, Iss. 1, p. 9-29.
15. Almeida, E., E., Luntz, J., E., Tibury, D., M. 2007. Event-condition-action systems for reconfigurable logic control. *IEEE Transactions on Automation Science and Engineering*, Vol. 4, Iss. 2, p. 167-181.
16. Ohashi, K., Shin, K., G. 2011. Model-based control for reconfigurable manufacturing systems. *Proceedings of IEEE International Conference on Robotics and Automation*, p. 553-558.
17. Liu, J., Darabi, H. 2004. Control reconfiguration of discrete event systems controllers with partial observation. *IEEE Transactions on Systems, Man, and Cybernetics, Part B, Cybernetics*, Vol. 34, Iss. 6, p. 2262-2272.
18. Dumitrache, I., Caramihai, S. I., Stănescu, A., M. 2000. Intelligent agent-based control systems in manufacturing. *Proceedings of IEEE International Symposium on Intelligent Control*, p. 369-374.
19. Oshana, R., Kraelig, M. 2013. *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications*, Newnes.

Part III

Short Presentations

Reengineering the Editor of the GreatSPN Framework

Elvio Gilberto Amparore

Università di Torino, Dipartimento di Informatica, Italy
amparore@di.unito.it

Abstract. This paper describes the technical challenges around the modernization process of the GreatSPN framework[15], one of the first Petri net frameworks started in the eighties, in particular in the reengineering of its Graphical User Interface and in its general user-friendliness, to account for its large set of functionalities¹.

Keywords: GreatSPN, GUI, Dataflow architecture.

1 Objectives and contributions

It is common opinion that formalisms like Petri nets or Timed Automata are very powerful and yet simple to understand thanks to their graphical representation. A graphic schema is usually simple to specify and grasp. However, graphical formalisms depends on the availability of good graphical editors to be able to gain the full advantages.

Back in 1985, the University of Torino developed the (probably) first documented software package for the analysis of *stochastic Petri nets*, under the name of *Graphical Editor and Analyzer for Timed and Stochastic Petri nets* (GreatSPN) [15]. The framework consisted in a set of tools for the analysis of *Generalized Stochastic Petri nets* (GSPN) [1], and it was supported by a graphical editor, described in [16], for interactively design, validate and evaluate GSPN models. The graphical editor, developed initially on a Sun 3 machine with UNIX BSD 4.2, was based on the X11/Motif toolkit. The simplicity of graphically designing models boosted the usage of GSPNs in various fields, from performance evaluation, telecommunications, biology and more. Other tools/GUIs have then followed, providing nowadays a large base of Petri net tools.

Today, the GreatSPN framework provides a vast collection of solvers developed in a time span of 30 years, that includes solvers optimized for low memory consumption (for the computers of the late '80s), modern model checkers based on decision diagram techniques [3], a stochastic model checker, ODE/SDE² solvers, Markov decision process optimizers, DSPN [23] solvers, simulators, and others. While the set of solvers of GreatSPN is very large, the obsolescence of technologies like Motif hurt the usability of the GUI over the years. After having evaluated other GUIs, the group arrived to the decision of renewing the interface of GreatSPN. The modular nature of the framework itself allows to easily replace solvers, modules and the GUI itself, while maintaining

¹ This work has been funded by the *Compagnia di San Paolo*, as a part of the AMALFI (Advanced Methodologies for the Analysis and management of the Future Internet) project.

² Ordinary and Stochastic Differential Equations.

the other modules fully working and unchanged. In the end, the modular framework structure proved to be easy to maintain and to develop over such a long timeframe.

This paper describes the reengineered GUI of GreatSPN, with its recent enhancements. The GUI is written in Java, and it is therefore portable to multiple platforms. Enhancements include, among all, support for drawing colored Petri nets and hybrid Petri nets, token game, batch measure specification and processing, and support for multiple solvers/model checkers. The paper describes the overall tool workflow, from the modeling and the verification phase, that allows to edit models, simulate their behaviors, inspect their structural properties, up to the evaluation phase, where performance indexes are computed with numerical solvers and/or simulators, and the computed results are visualized interactively to the user. A prototype of the GUI was briefly described in a short paper [5], centered around its use for stochastic model checking.

The GUI supports multiple formalisms: Generalized Stochastic Petri Nets (GSPN), GSPN with colors (*Stochastic Well-Formed net*, or SWN), Hybrid Petri nets, and Deterministic Timed Automata (DTA). In addition, the application presents a number of unique features, like multipage projects, solution batches, support for template variables in models, \LaTeX labels and high quality vector graphics. This new GUI is described here with a focus on various recent additions: parametric measure specification, the support for SWN and hybrid Petri nets, and the integration inside the framework.

The modernization process actually required a process of re-engineering of the GUI around the workflow of GreatSPN. During this process, many limitations of the existing GUI have been removed, like the absence of a SWN token game, the missing support for model checkers, no capacity for drawing Timed Automata, and other. The main contributions provided by the modernized GreatSPN GUI are its improved usability, while keeping the compatibility with the large framework, and its support to multiple formalisms and solvers, which expands the tool usability. Other formalisms and other solvers may be added using the modular tool structure.

Section 2 describes the architecture of the GreatSPN framework. Section 3 and 4 introduce the application interface, and describe briefly the modeling capabilities of the editor. Section 5 shows how the user can simulate the designed GSPNs with the token game, and visualize their structural properties like the minimal P/T semi-flows. Section 6 describes how the designed models can be verified quantitatively with the set of supported solvers. Section 7 shows a simple use case of the tool that illustrates how the GUI can help the user in the process of modeling and analysis. The paper concludes with a comparison of other commonly used GUIs in section 8 and with the section 9 with a brief discussion on the future of GreatSPN.

2 Architecture of GreatSPN

GreatSPN is a large framework made by several interacting components, that has grown over the time to incorporate various Petri net-related features. The framework itself is not made by a large, monolithic tool. Instead, many independent tools interact by sharing data through files in standardized formats, resulting in a *dataflow architecture* approach. Each tool is responsible for reading its own input, written in one or more files, performing the computation, and writing the outputs in other files. The framework actu-

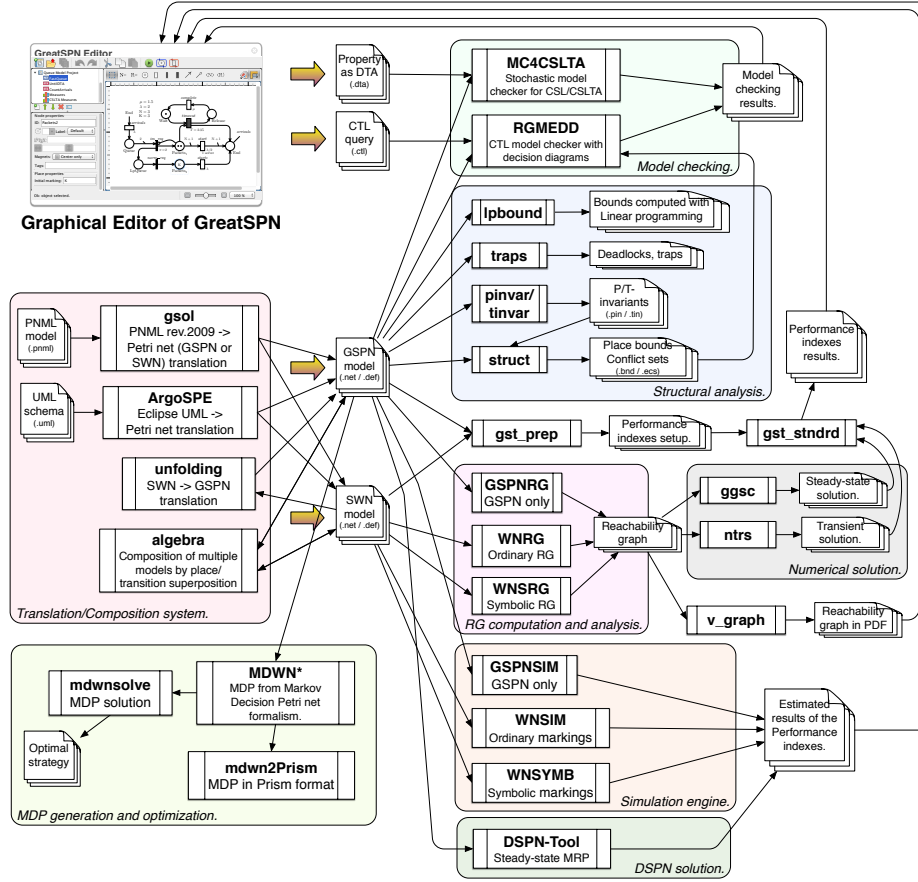


Fig. 1. (Partial) Architecture of the GreatSPN framework, as it is today.

ally contains more than 60 binaries. The advantage of this software architecture is that it allows to easily modify/replace single components, while keeping the rest of the framework unchanged, as long as the input/output formats are observed. While this software architecture is not very modern, it has proven to be very solid and maintainable, such that in the framework many software modules written in its 30 years of development co-exists, without causing too many troubles.

A simplified schema of the GreatSPN framework is shown in Fig. 1, that reports a selection of the various features of GreatSPN. Tools are written in bold, and are grouped in logical modules, according to their function, that span from numerical solutions, structural analysis, MDP support, conversion between multiple formalism, and so on. The graphical editor is the center for drawing the models and their properties. It is responsible for the invocation of various command line tools and for the visualization of the results. Actually, many but not all the command line tools are available from the GUI.

The workflow of GreatSPN was conceived, back in its original design, to be made in three main phases: first the user (the “modeler”) designs the Petri net in a textual or graphical way; secondly, structural properties are computed (minimal P/T semi-flows, place bounds, conflict sets, ...) to understand if the model is designed properly and may be solved under numerical analysis or simulation; then the user specifies the measures of interest on the model and calls a command line solvers to do the computation. Several solvers are provided, for different types of models and with different characteristics.

2.1 Reengineering requirements.

In order to create a new GUI that replaces the old one, it must satisfy a set of requirements and constraints imposed by the framework itself. First of all, the GUI is responsible of these tasks:

1. Help the user in the process of drawing Petri net models and other graphical models.
2. Allow the user to call the tools provided by the GreatSPN framework for the structural analysis of Petri net models, to discover potential structural mistakes made in the drawn models.
3. Simplify the process of specifying and computing performance measures, by calling the command-line solvers and providing an understandable visualization of the computed results.

The design choices done to satisfy requirement 1 are explained in sections 3 and 4. Requirements 2 and 3 involve the interaction between the GUI and the command line tools. Command line tools expect precise file formats in input and produce other files as output, since these tools are designed to be non-interactive. Therefore, the tool interaction with the solvers require an explicit serialization of the data for the computation and a deserialization of the results. Many different files are involved in any computation: a partial list of these files is shown in Table 2.1.

Extension	Content of the file
.PNPRO	Petri net Project (XML format). Main format of the editor.
.net	Input format of Petri net models for command line solves.
.def	Input performance indexes.
.ctl	Qualitative queries in CTL language.
.dta	Deterministic timed automata for CSL ^{TA} model checker.
.pin, .tin	P/T invariants.
.sta	Computed statistics.
.throu	Transition throughputs.
.tpd	Token distributions in places.
.ecs	Extended conflict sets.
.bnd	Upper/lower bounds of tokens in places.
.grg	Reachability graph.

Table 1. File extensions of the input/output files used by the editor and the solvers).

The complete solution process of a Petri net may require the invocation from one to twelve different command line tools, depending on the target measures to be computed. In addition, the reengineered workflow has been improved to support parametric models, i.e. models defined to depend on multiple integer/real parameters, whose values are specified at solution time and not at design time. Parameters are passed as command line arguments, and all command line tools have been modified to support them.

Another design requirement of the GUI is to support new modeling formalisms and functionalities that are not present in the current toolchain. To represent and store these informations, the tool uses a new XML file format for the models. This choice avoids modifying the input file formats of the toolchain. Any command line tool invocation serializes the drawn model in appropriate input formats when needed, leaving the main file format just for the editor. In this way, it is easy to change the editor format without breaking the compatibility with the command line tools of the GreatSPN framework.

Requirements 2 and 3 involve the reconstruction of the modeler workflow of GreatSPN inside the GUI. Examples of how this workflow is implemented graphically are given in sections 5–7.

2.2 Code structure of the new GUI.

The GUI consists of about 55K lines of code written in the Java™ language, plus an optional command line L^AT_EX engine that runs in the background to format the text labels of the models. The application is cross platform and runs on Windows, MacOSX and Linux. Java package structure is shown in Table 2.2.

Packages	Description
gui	Core GUI structure, main window cycle.
gui.net	Visualization/editing of abstract graphs (Petri nets, automata).
gui.play	Interactive token game.
gui.semiflows	Visualization of minimal P/T semi-flows.
gui.measures	Editing of measures and visualization of computed results.
domain	Data structures.
domain.project	File management, undo/redo facility.
domain.grammar	Unified ANTLRv4 grammar for expressions and measures.
domain.io	Serialization/deserialization in net/def, XML and APNN formats.
domain.values	Expression evaluation engine.
domain.elements.gspn	GSPN elements.
domain.elements.dta	DTA elements.
domain.play	Token game logic.
domain.semiflows	Computation of minimal P/T semi-flows.
domain.measures	Measure specification and tool invocations.
domain.unfolding	Unfolding of colored Petri nets into uncolored ones.

Table 2. Code structure of the Java application.

The core structure of the design view of the GUI is essentially an editor for abstract graphs of nodes and edges. The version described in this paper supports two graph

formalisms: Petri nets and automata. New formalisms can be added by deriving the corresponding base classes in the Java codebase. Adding a new formalism is done by deriving the base classes for the model, the node elements and the edge elements, and by providing the Java panels to edit properties. For instance, the DTA formalism, implemented in the `domain.elemens.dta` package, involves about 2K lines of code: two Java classes for the DTA locations and edges (the graph nodes), a class for the DTA model in a project, and the property panels for the location and edges. Of course, other part of the application that use DTAs also involve some additional logic. To abstract different syntax of properties, measures, expressions, provided by various solvers, the GUI has a uniform C-like language for expressions. When an expression needs to be passed to a solver, it is converted to the specific syntax expected by the tool. Abstracting expression languages of different solvers allows to support multiple solvers without having to re-specify expressions and measures for different tools. Overall, the complete GreatSPN framework amounts to about 500K lines of code, mostly made by C/C++ programs.

3 Drawing Petri net models

The core feature of the editor is the drawing of Petri net models, centered around the GSPN, the SWN and the Hybrid Petri net formalisms. Figure 2 shows the main application window, taken while editing a colored Petri net model. In the upper-left panel, there is the list of open projects. The editor is designed around the idea of *multi-page* projects. Each project correspond to a file, and is made by several pages. In the current version of the editor, pages can be of three types: Petri net models, DTA models or table of measures. In the lower-left panel of the main window there is the property panel, that shows the editable properties of the selected objects. The central canvas contains the editor of the selected project page, that is in this case a SWN model.

Petri nets are drawn with the usual graphical notation. Transitions may be immediate (thin black bars), exponential (white rectangles) or general (black rectangles). Names, arc multiplicities, transition delays, weights and priorities are drawn as small movable labels near the corresponding Petri net elements. Arcs may be “broken”, meaning that only the beginning and the end of the arrows are shown. Color definitions are drawn in textual form, as in the upper right part of the window where two color classes, a composite color domain and two color variables are declared. The editor also supports fluid places and fluid transitions (not shown in the example of Fig. 2), and place partitions for Kronecker-based solutions [11]. The editing process supports all the common operations of modern interactive editors, like undo/redo of every actions, cut/copy/paste of objects, drag selection of objects with the mouse, single and multiple editing of selected objects, etc. Great care has been put to the graphical quality of the resulting Petri net models, to allow for high quality visualization of the net. The interface is designed to avoid modal dialog windows as much as possible, to streamline the use of the GUI.

Figure 3 shows some of the extended features of the Petri net editor. Name labels for elements (places, transitions, constants, etc) may appear in three user-selected modes:

- The label shown is the alphanumeric object identifier, as-is;
- A \LaTeX string is used, allowing for more readable models that better express their meanings, like in the simple reaction network of Fig. 3(A) where alphanumeric

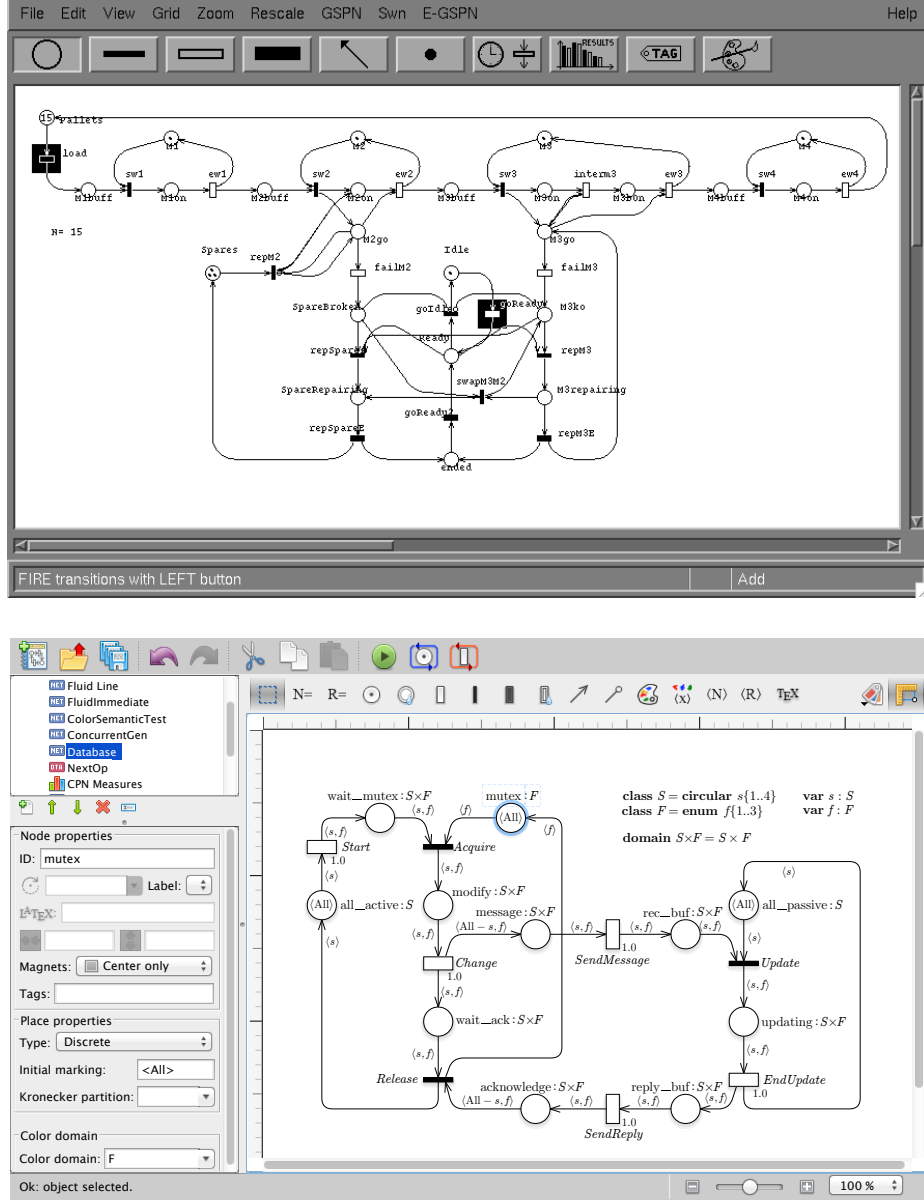


Fig. 2. The old and the new graphical user interfaces of GreatSPN. Screen-capture of the former is taken during the interactive token-game, while the SVG capture of the latter shows the design view with a colored Petri net model.

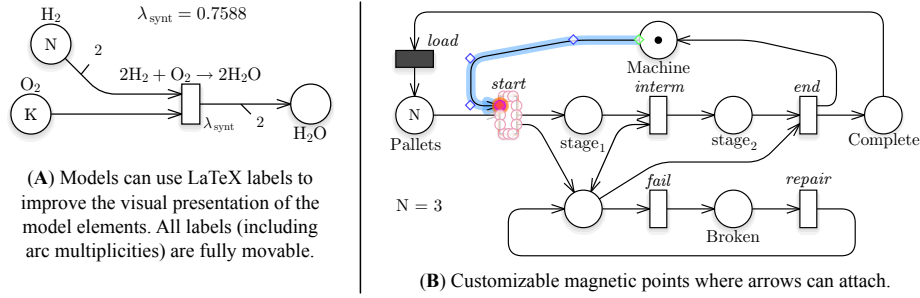


Fig. 3. Some features of the Petri net editor.

transition names are replaced with the represented chemical reaction, and place names represent the chemical species.

- The label is the object identifier automatically formatted in \LaTeX with a function that tries to convert common patterns (like `a1pha` \rightarrow α , or `stage1` \rightarrow stage_1).

Arc arrows point to the center of the attached transitions/places. If this behavior is not satisfactory, the editor provides a set of customizable “magnetic points” drawn on the element perimeters, where the arrows may attach. This behavior is shown in Fig. 3(B), figure that has been taken while dragging the arc arrow with the mouse on the “start” transition that has “3 magnets per side”. All elements in the model are vector based, which result in high print quality. Printing and PDF exportation of the models are also possible, using the printing facilities of the operating system.

Figure 4 shows a colored model, drawn using the SWN formalism, as it appears in the editor window. Support for SWN has been recently added to the GUI. The model has three objects, located in the upper left part, that represents object declarations. The $\langle N \rangle$ objects declares a parametric integer constant, whose value is decided at verification time. The ‘class’ declaration defines a color class for the places in the Petri net, as usual in the SWN formalism. Places belonging to this color class are labeled with the place name followed by a colon and the color class name. The third declaration is a color variable named x of color class *Philo*. All expressions are parsed and verified syntactically and semantically on-the-fly, and appear in red if there is some error.

4 Drawing CSL^{TA} DTAs

The second type of models that can be drawn with the editor are *Deterministic Timed Automata* (DTA), a type of timed automata for the CSL^{TA} stochastic logic [19]. CSL^{TA} works by measuring stochastic GSPN behaviors using a DTA. A DTA is an automaton that reads the language of GSPN firing sequences (also called *paths*), and separates accepted and rejected paths. The formal semantic of the DTA can be found in [19] (single clock), and in [14] (with multiple clocks). In few words, the logic provides a stochastic operator: $s_0 \models P_{\bowtie \lambda}(A)$ that is satisfied iff the overall probability of the set of GSPN paths starting in state s_0 and accepted by the DTA A , is $\bowtie \lambda$.

Figure 5 shows three CSL^{TA} DTAs, drawn with the notation described in [4]. The first DTA describes the CSL [7] path property: $\Phi_1 \text{ Until}^{[\alpha, \beta]} \Phi_2$. Locations are drawn

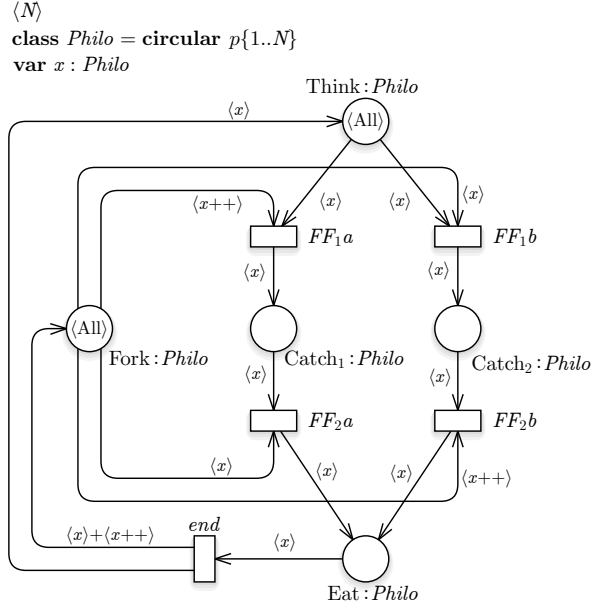


Fig. 4. Model of the N dining philosophers drawn in the SWN formalism.

as rounded rectangles, and the state proposition that the GSPN must satisfy while the DTA is in a location is written below the location rectangle, in bold. Initial locations are represented with an entering arrow, and final locations are drawn with a double border. The editor also allows *final rejecting locations*, not included in the original definition, but used in [2]. There are two kinds of edges, *boundary*, drawn dashed, and *inner*, drawn solid. Boundary edges are triggered as soon as the clock condition is satisfied, and are labeled with a \sharp . Inner edges specify the set of GSPN actions with which they are synchronized. Each edge also specify a set of clock constraints, and an optional set of clock resets.

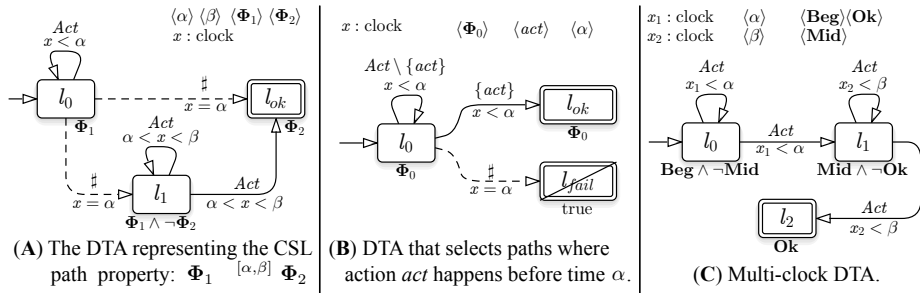


Fig. 5. Some example of DTA models drawn with the editor.

DTAs are parametric, and are not bound a priori to a specific GSPN model. Instead, all state propositions, real clock boundaries and action names are declared as *template variables* (depicted as $\langle var \rangle$). When the DTA is used for computing measures of a GSPN, these parameters are instantiated to boolean conditions, real values and transition names of the GSPN, as we shall see in section 5.1.

Clocks are declared as part of the DTA. The DTAs (A) and (B) of Fig. 5 have a single clock, while the DTA (C) has two clocks. Currently, only single-clock DTAs can be verified numerically. The DTA (B) accepts all the GSPN where a transition *act* fires before time α , while remaining in states that satisfy the condition Φ_0 .

5 Interactive simulation and inspection of structural properties

The behavior of Petri nets can be experimented interactively inside the GUI. This is known as the “token game” or “interactive simulation”, and works as follows. The editor shows the initial marking of the GSPN, and highlights the set of enable transitions of the model. By click on one of the enabled transitions, the editor responds by firing the tokens from the input to the output places, showing the behavior of the model. The reached marking is then shown, and the user can continue firing new transitions.

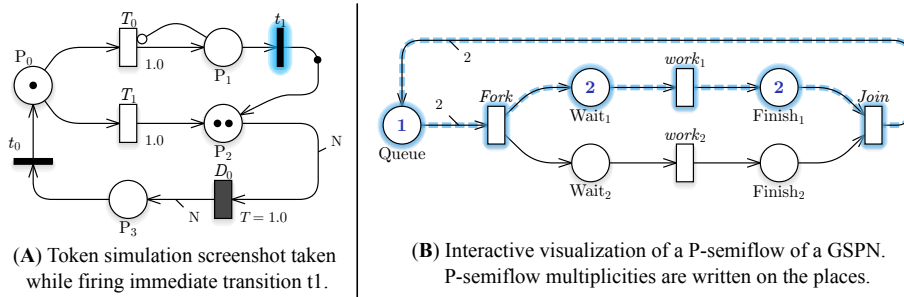


Fig. 6. Interface of the interactive GSPN simulation and semi-flows visualization.

Figure 6(A) shows this interactive simulation on a GSPN model, taken during the firing of transition t_1 . Tokens removed from input places and added to output places are drawn with a short animation. Token game works in untimed and timed mode. In untimed mode, no age/duration of the events is considered, and no track is kept for the advance of time. In timed mode instead, a time is present, and the time for the transition fire is taken into account. For DSPN models with non-exponential transitions, timing constraints are resolved. The user may also enable a *semi-automatic firing* mode, where interaction is required only if there is a choice between multiple concurrently enabled transitions, or a *random firing* mode where the editor picks the next transition randomly (and the next time in timed simulation), thus simulating without the user interaction. SWN models are supported in interactive mode: instead of black dots, colored tokens are shown as color names. When firing a colored transitions, a list of enabling bounds of the color variables is shown to the user, who can pick the one to fire.

Additionally, the user may visualize the minimal P and T semi-flows that covers a GSPN model, as shown in Fig. 6(B). The user selects the minimal semi-flow that wants to visualize from a list, and the editor highlights the involved places and transitions. Semi-flows are computed with the modified Farkas method of Martinez and Silva [24]. With these tools, the user may inspect the behavior and the structural properties of the Petri net while modeling, which is useful to verify that the model is drawn correctly.

5.1 Interactive CSL^{TA} simulation

An interactive simulation of the path probability operator of the CSL^{TA} logic is, roughly speaking, a system where GSPN firings are checked by the DTA. Each GSPN transition firing has to be matched by a corresponding DTA edge, otherwise the path is rejected. In addition, boundary edges of the DTA (labeled with a \sharp and drawn as dashed arrows) are autonomous and are taken as soon as their timing conditions are met. Before starting the simulation, the template variables of the models are shown as a list of text boxes, that must be filled by the user with appropriate values. Values assigned to the parametric variables of the DTA are shown above the DTA. In the central panel of the window, the GSPN and the DTA are shown side by side.

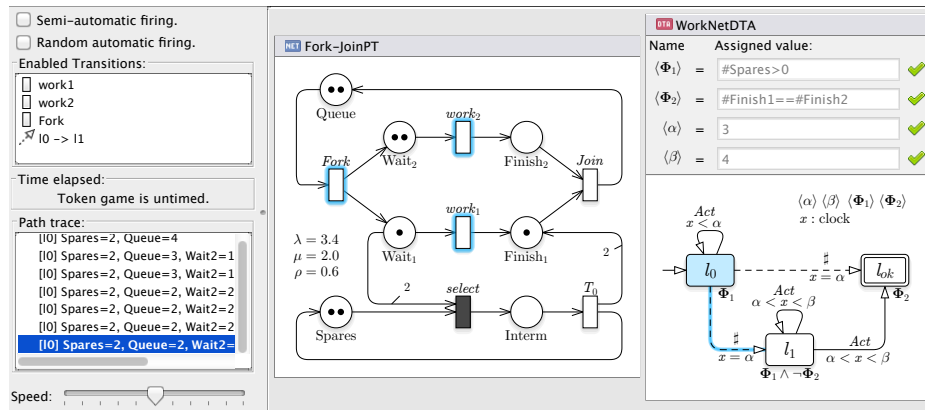


Fig. 7. Interface of the interactive CSL^{TA} model checking simulation.

Figure 7 shows the GUI window for the joint simulation of a GSPN model and a DTA. The list of enabled GSPN transitions and autonomous DTA edges is shown in the upper-left corner. In the lower left corner there is the state of the path trace chosen interactively by the user, starting from the initial marking. Values assigned to the parametric variables are validated while typing, and their correctness is signaled with a green tick mark on the right of the corresponding text boxes. When all values are assigned, the user may press the play button and the joint simulation starts in the initial state of the GSPN and in the initial location of the DTA. Each time the user selects a GSPN transition to fire, a DTA inner edge has to be chosen afterwards to match the GSPN firing. Boundary edges of the DTA may also be independently enabled

(clock condition is evaluated in a timed simulation, and ignored in an untimed one). The simulation ends when the DTA reaches a final location, or when no DTA edge can match a GSPN firing.

The screenshot shows a GUI for specifying and computing a batch of measures. It is organized into several sections:

- Target model:** A dropdown menu set to "Database CPN".
- Solver:** A dropdown menu set to "GreatSPN".
- Template parameters:** A section for defining the template parameters. It includes a "Name" field with the value "<n>", a "ranges" dropdown, and input fields for "from" (1), "to" (8), and "step" (1). Each of these fields has a green checkmark icon to its right. There are also up and down arrow buttons.
- Solver parameters:** A section for defining solver parameters. It includes an "Epsilon" field set to "1.0e-7", a "Max iterations" field set to "10000" with a green checkmark, a "Solver mode" dropdown set to "SWN Ordinary", and a "CTMC solution is computed in:" section with three radio buttons: "Steady state" (selected), "Simulation", and "Transient". There is also an "at:" field set to "1.0" with a green checkmark.
- Measures:** A table with three rows, each representing a different measure. The first row is labeled "1°" and has a radio button, a dropdown set to "ALL", and the description "All place distributions and transition throughputs." with a "Compute" button. The second row is labeled "2°" and has a radio button, a dropdown set to "RG", and the description "Plot of the Reachability Graph with vanishing markings." with a "Compute" button. The third row is labeled "3°" and has a radio button, a dropdown set to "PERF", and the description "E{ #Queue }" with a green checkmark and an equals sign, followed by a "Compute" button.

At the bottom right of the interface, there are two buttons: "View log..." and "Compute All".

Fig. 8. The interface for specifying and computing a batch of measures.

6 Computing measures

The GUI integrates an interface for specifying, computing and visualizing measures on Petri net models. A project may contain multiple *measure pages*, and each page specifies:

- The target Petri net model;
- The selected numerical solver, from a list of supported solvers;
- The instantiation of the parameters of model, if any;
- Solver-specific parameters and flags;
- A table of target measures that will be computed.

The GUI is currently integrated with three solvers. The first is the GreatSPN toolchain, that can evaluate standard performance measures (mean number of tokens in a place, transition throughputs, etc...) on GSPN/SWN models using an extensively tested implementation. Index can be computed in steady-state or in transient with a numerical solver, or by using a simulator. The second solver is the MC4CSL^{TA} stochastic model checker, that can evaluate standard performance measures for GSPN and DSPN models [6], as well as CSL and CSL^{TA} queries. The third solver is RGMEDD [3], the symbolic CTL model checker of GreatSPN [8].

Figure 8 shows a measure page editor for a GSPN model with one parametric marking parameter $\langle n \rangle$ and with three measures (at the bottom). The GSPN model is evaluated multiple times for different values of n , from 1 to 10 (template parameters section), with increments of 1. The numerical solution is computed by invoking the command-line solvers with the specified solver parameters (solution in steady-state, maximum number of iterations, use the ordinary SWN solution, etc..). The table of measures lists what will be computed. Entry ALL specifies that all basic GSPN measures will be computed, which are the distributions of tokens in each place, and the transition throughputs. RG and TRG specify that the (Tangible) Reachability Graph will be generated by the GreatSPN tools, and a graphical representation will be drawn. Queries in a given language (CTL, CSL, CSL^{TA}, Performance measures) can be specified textually. A PERF query expresses a performance measure on places and transitions, using the syntax of the measure language of GreatSPN.

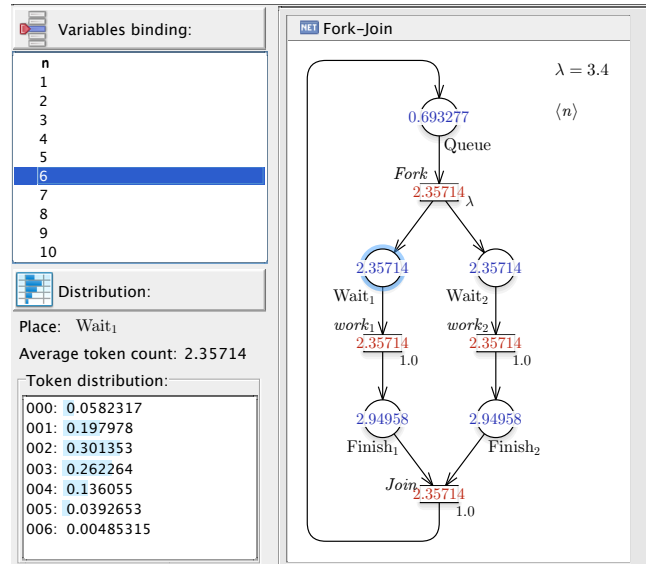


Fig. 9. The interface that shows the ALL results computed on a parametric GSPN in steady-state.

When the user clicks on the “Compute” button, the GUI calls the command-line numerical solvers, and shows the output to the user. To use the command line tools directly, the user can export the GSPN/DTA models as separate files. Currently supported formats are the GreatSPN format and the APNN format [20].

Computed solutions are shown interactively to the user. Figure 9 shows the interface that is used to show the results of the ALL measure, computed with parameter $\langle n \rangle$ that ranges from 1 to 10. Places and transitions show their expected number of tokens and throughputs, respectively, for the value selected by the user (in this case $n = 6$). When the user selects a place, its token distribution is shown in the bottom-left corner.

Template parameters can be bound to a single value, a list of values or a range of values. If the performance measures are computed in transient, it is possible to specify that the transient time t is template variable, thus computing a sequence of solutions at different time steps. This allows the user to setup a batch of parametric tests with a certain degree of flexibility.

7 Use case

We now present a simple use case to show how the tool functionalities can be used to support model analysis with standard performance measures as well as with CSL^{TA} queries.

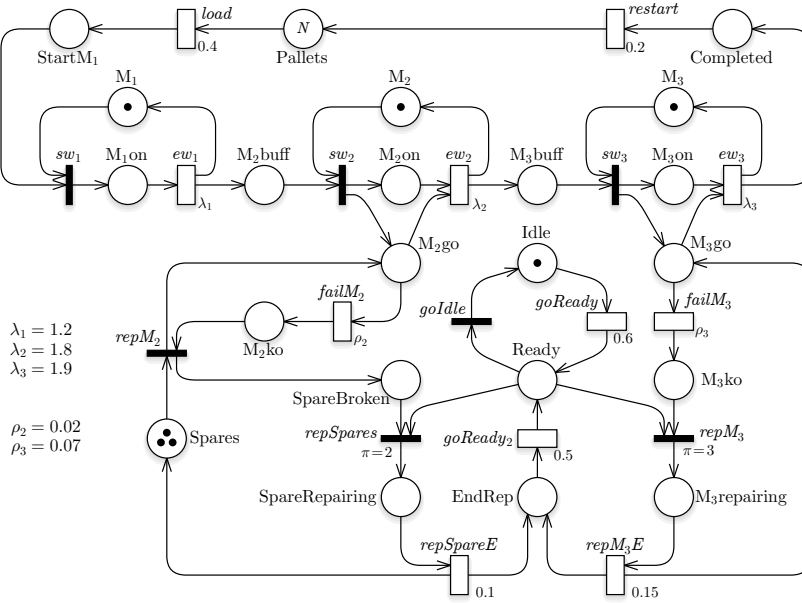


Fig. 10. The GSPN of a Flexible Manufacturing System (FMS).

Figure 10 shows an instance of a *Flexible Manufacturing System* (FMS) taken from [9], modeled with the GUI and exported as PDF. The model represents a system where N pallets are treated in a sequence of three machines, M_1 , M_2 and M_3 . Each machine can treat one pallet at a time. Machine 2 and 3 are subject to breakages, and a repairman continuously checks the machine for repairs. Machine 2 has a set of spare parts that can be used to replace the broken parts, without losing work time. Machine 3 instead always requires a stop to do the repair. The model is parametric in the number N of circulating pallets.

Figure 11 depicts two DTAs drawn with the GUI that express two path properties on the FMS model. The first accepts the system behaviors where three completion events

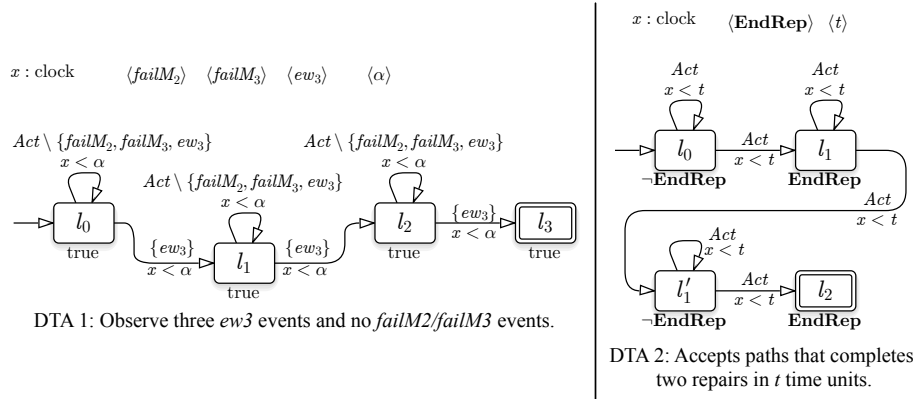


Fig. 11. The Two DTAs used in the analysis of the FMS model.

ew_3 are observed before time α , having not seen any failure of the machines M_2 and M_3 . The second DTA accepts the paths where at least two repairs have been completed in t time units.

To carry on the analysis from the editor, it is sufficient to create a new set of measures for the FMS model, that are parametric in the number N of pallets and on the time t of the DTA. Figure 12 shows the results of the analysis of the FMS model by computing the steady state solution and the model checking of the two DTAs. Datas are computed using the GUI, and then exported from the GUI in Excel Open XML format to plot the diagrams.

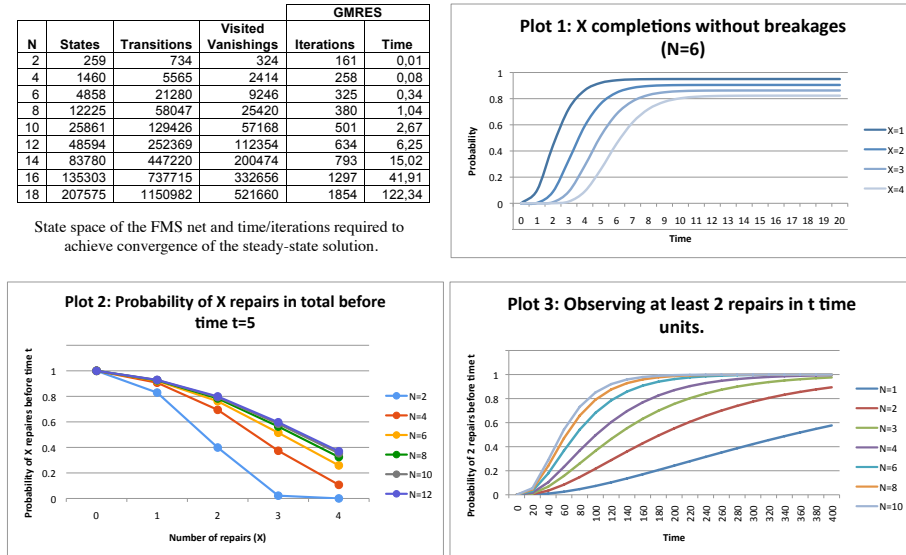


Fig. 12. Results of the FMS model analysis, visualized in Excel.

The table in the upper-left corner shows the size of the FMS model state space, the number of transitions, the number of vanishing states visited by the numerical solver, and the iterations/time needed to compute the steady state solution on a 2.4Ghz Intel machine with accuracy of 10^{-7} .

Plot 1 represents the result of DTA 1 on the FMS with $N=6$ pallets for 1,2,3,4 completions (the DTA of Figure 11 represents the case $X=3$). The plot shows the probability of having enough time to do X completions, and for large values of t converges to the probability of observing a failure. Plot 2 shows the results of DTA 1 by varying the number of circulating pallets N with a fixed time $t = 5$. The x-axis plots the number of completions, while the y-axis shows the overall probability of completing X tasks before time t . A larger number of pallets increases the throughput of the system, resulting in an increased probability. Finally, Plot 3 shows the results of DTA 2, i.e. the probability of observing two repairs in t time units. Time is plotted on the x-axis and probability on the y-axis, for various numbers of circulating pallets. Since the machine may break when it is under usage, the probability increases for higher values of N .

8 Related work

While the GreatSPN framework with the new interface provides a solid base for editing, verifying and computing quantitative/qualitative measures of GSPN/SWN models, there are other tools that provide similar features. Before reimplementing a new GUI, we have explored various alternatives. An (incomplete) list is given.

Möbius : The aims of the Möbius tool [18], developed at the University of Illinois, are similar to those of GreatSPN. It supports multiple formalisms, multiple solvers, and provides a complete analysis workflow, from design to verification to the numerical solution. It supports analysis of discrete and continuous time Markov chains, Markov regenerative processes and a powerful simulator. However, the central formalism is the SAN, not the stochastic Petri net, so it is not directly suitable for GreatSPN (even if SAN nets can be converted to GSPN). In addition, no SWN and no stochastic model checking is available.

Snoopy : The tool Snoopy [21] is a proprietary software developed at Cottbus TU. It provides a unified editor for Petri net models, with support for hierarchical composition and multiple solvers. Snoopy supports hybrid Petri nets (HPN), colored Petri nets, as well as other extensions. The main solver is Marcie, based on MTBDD/MTIDD (decision diagram variants) techniques.

Coloane : Coloane [17] provides support for both Petri net and Timed Automata, but is currently not focused on stochastic formalisms. It is designed to provide a GUI around the standard PNML format [12], an XML-based exchange format for Petri net models.

CPNtool : CPN is a powerful toolkit for the design and evaluation of colored Petri nets [22]. The formalism adopted by CPN includes a color algebra, expressions in ML. The tool is supported by a flexible simulation engine. The specific mix of ML code

and Petri net graphics is very compact and powerful, but unfortunately is far from what GreatSPN solvers expect. In addition, the tool has some portability concerns. Therefore, a conversion between CPNtool models and GreatSPN models appears difficult.

TimeNet A successor of the DSPN-express tool developed at TU Berlin, TimeNET is a modern tool for editing stochastic and colored Petri nets. It is still being developed, and it has been recently updated with heuristic optimization techniques [13].

The specific characteristic of the GreatSPN models, and the vast number of solvers lead to the decision of designing a specific GUI for it. Additionally, some features like the DTA specification and the support to the CSL^{TA} stochastic logic are, to the best of our knowledge, a unique feature of the GreatSPN GUI, and are not found on other tools. The tool is available at <http://www.di.unito.it/~greatspn/index.html>, in the “New Java GUI” section, either as a part of GreatSPN or as a standalone version. A virtual machine with all the tools installed is also available, at request.

9 Conclusions and Future works

This paper presents an in-depth analysis of a new graphical user interface for the GreatSPN framework. The GUI is designed around a complete workflow for the modeling of Petri nets and DTAs, and includes graphical interactive analysis, specification of measures, computation and interactive visualization of the results, and an integration with multiple solvers/simulators/model checkers/optimizer/translators including a CSL^{TA} stochastic model checker and GreatSPN. A small use case has been also presented, to show the effectiveness of the GUI modeling capabilities and analysis with measures from the user point of view.

Since the tool architecture is scalable and customizable, we plan to extend the tool in various directions. First of all, the Petri net formalism can be augmented to support other extensions, like compositional formalism. Similarly, DTAs can be extended to cover more complete statistical control automata, like Linear Hybrid Automata [10]. Solvers and file formats of other framework can also be considered, like PNML, and there is an ongoing work to support solvers[11] of the APNN-Toolbox of TU-Dortmund.

References

1. Ajmone Marsan, M., Conte, G., Balbo, G.: A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. *ACM Transactions on Computer Systems* 2, 93–122 (May 1984), <http://doi.acm.org/10.1145/190.191>
2. Amparore, E., Donatelli, S.: Improving and assessing the efficiency of the MC4CSL^{TA} model checker. In: *EPEW 2013* (2013)
3. Amparore, E.G., Beccuti, M., Donatelli, S.: (Stochastic) model checking in GreatSPN. In: *Application and Theory of Petri Nets, LNCS*, vol. 8489, pp. 354–363. Springer (2014)
4. Amparore, E.G.: State, action, and path properties in Markov chains. Ph.D. thesis, Dipartimento di Informatica, Università di Torino, Italy (2013)
5. Amparore, E.G.: A New GreatSPN GUI for GSPN Editing and CSLTA Model Checking. In: Norman, G., Sanders, W. (eds.) *Quantitative Evaluation of Systems, Lecture Notes in Computer Science*, vol. 8657, pp. 170–173. Springer International Publishing (2014)

6. Amparore, E.G., Donatelli, S.: Revisiting the Iterative Solution of Markov Regenerative Processes. *Numerical Solution of Markov Chains (NSMC)* (2010)
7. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. *ACM Trans. Comput. Logic* 1(1), 162–170 (2000)
8. Babar, J., Beccuti, M., Donatelli, S., Miner, A.S.: GreatSPN Enhanced with Decision Diagram Data Structures. In: Lilius, J., Penczek, W. (eds.) *Petri Nets*. LNCS, vol. 6128, pp. 308–317. Springer (2010)
9. Balbo, G., Beccuti, M., De Pierro, M., Franceschinis, G.: First Passage Time Computation in Tagged GSPNs with Queue Places. *The Computer Journal* (2010), first published online July 22, 2010.
10. Ballarini, P., Djafri, H., Duflot, M., Haddad, S., Pekergin, N.: HASL: An expressive language for statistical verification of stochastic models. In: *Proceedings of VALUETOOLS'11*. pp. 306–315. Cachan, France (May 2011)
11. Bause F. and Buchholz P. and Kemper P.: Hierarchically combined queueing petri nets. In: *In Proc. 11th Int. Conf. on Analysis and Optimization of Systems, Discrete Event Systems, Sophie-Antipolis*. pp. 176–182. Sophie-Antipolis, France (1994)
12. Billington, J., Christensen, S., Van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The petri net markup language: Concepts, technology, and tools. In: *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*. pp. 483–505. ICATPN'03, Springer-Verlag, Berlin, Heidelberg (2003)
13. Bodenstein, C., Zimmermann, A.: Timenet optimization environment: Batch simulation and heuristic optimization of scpn with timenet 4.2. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. pp. 129–133. VALUETOOLS '14, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2014)
14. Chen, T., Han, T., Katoen, J.P., Mereacre, A.: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications. *Logic in Computer Science, Symposium on* 0, 309–318 (2009)
15. Chiola, G.: A software package for the analysis of generalized stochastic petri net models. In: *International Workshop on Timed Petri Nets*. pp. 136–143. IEEE Computer Society, Washington, DC, USA (1985)
16. Chiola, G.: A Graphical Petri Net Tool for Performance Analysis. In: *Third Int. Workshop on Modeling Techniques and Performance Evaluation*. pp. 323–333. Paris, France (1987)
17. Coloane webpage. <https://coloane.lip6.fr/>
18. Courtney, T., Daly, D., Derisavi, S., Gaonkar, S., Griffith, M., Lam, V., Sanders, W.: The Mobius modeling environment: recent developments. In: *International Conference on Quantitative Evaluation of Systems (QEST)*. pp. 328–329 (2004)
19. Donatelli, S., Haddad, S., Sproston, J.: Model checking timed and stochastic properties with CSL^{TA}. *IEEE Trans. Softw. Eng.* 35(2), 224–240 (2009)
20. F. Bause, P.K., Kritzing, P.: Abstract Petri nets notation. In: *Petri Net Newsletter*. vol. 49, pp. 9–27 (1995)
21. Heiner, M., Herajy, M., Liu, F., Rohr, C., Schwarick, M.: Snoopy: A unifying petri net tool. In: *Application and Theory of Petri Nets*, LNCS, vol. 7347 (2012)
22. Jensen, K., Kristensen, L.M.: *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer Publishing Company, Incorporated, 1st edn. (2009)
23. M. Ajmone Marsan, Chiola, G.: On Petri nets with deterministic and exponentially distributed firing times. In: *Advances in Petri Nets*. vol. 266/1987, pp. 132–145. Springer Berlin / Heidelberg (1987)
24. Martinez, J., Silva, M.: A simple and fast algorithm to obtain all invariants of a generalized Petri net. In: *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*. pp. 301–310. Springer-Verlag, London, UK, UK (1982)

Improving Performance of Complex Workflows: Investigating Moving Net Execution to the Cloud

Sofiane Bendoukha and Thomas Wagner

University of Hamburg, Department of Informatics
<http://www.informatik.uni-hamburg.de/TGI/>

Abstract. In this paper we propose and discuss mechanisms and implementation issues for moving the execution of computation- and time-consuming workflows into the Cloud. These complex workflows are specified by Petri nets, more precisely reference nets using the RENEW tool. We believe that Cloud technology is a suitable solution to (i) overcome the lack of resources on-premises and to (ii) improve the performance of the whole system based on quality of service (QoS) constraints. As execution target for simulations, tests have been performed on an OpenStack Cloud. Furthermore, the integration and interfaces between workflows, Cloud computing and agent concepts are also addressed.

Keywords: Petri nets; Cloud Computing; Workflows; Multi-agent Systems; Reference Nets; PAOSE.

1 Introduction

Several long-running and high-throughput applications can be designed as complex workflows, which describe the order and relationships between the different activities and related data (input, output). In such scenarios, these tasks often need to be mapped to distributed resources, possibly due to a lack of on-premise resources or failures. Recently, Cloud computing has attracted more interest from both the industry and academic community. Cloud computing is a recent computing paradigm. It has its origin in distributed computing, parallel, utility and grid computing. The National Institute of Standards and Technology (NIST) defines Cloud computing as: “A *model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*” [19].

In fact, Cloud technology provides an environment that allows to dynamically allocate resources for the execution of workflow tasks following an on-demand and pay-as-you-go model. In this work, we aim to take advantage of these resources to improve the performance of the applications. These applications are, in our case, specified as Petri nets using the **RE**ference **NE**ts **W**orkshop (RENEW) editor. In order to make this possible, we need to provide mechanisms and strategies that are based on the integration of workflow concepts and Cloud technology

(and later the agent paradigm). There are different ways to address this. On the one hand, Cloud for workflow uses Cloud resources to execute complex workflows and especially scientific workflows [12] [13]. Such works are more resource-centric and focus on the computational tasks. On the other hand, Clouds need structured and mature workflow concepts and high-level languages to handle issues like managing complex task and data dependencies. It should be noted that we only consider moving the execution of entire nets or systems of nets into the Cloud. Further distribution aspects of the simulation/execution¹ are outside of the scope of this paper.

The research described in this paper focuses more on performance issues, which can be considerably improved by using Cloud resources. We present our approach to provide techniques and tools to move the execution of complex workflows modelled in Petri nets to the Cloud. The migration to the Cloud is based mainly on user requirements. Thus Quality of Service (QoS) parameters are specified in advance. We emphasise response time and cost constraints, but this can be easily extended to other QoS parameters such as service availability. Modelling and execution of Petri net models is performed exclusively through the RENEW editor. Furthermore, we discuss different realisation possibilities. We examine three different types of interfaces, which define how input and output to the Cloud calls are defined. Simple interfaces provide only basic functionality to initiate Cloud workflows and receive results. Simulation interfaces are used to run extensive simulations of workflows in a Cloud environment. Lastly, advanced interfaces feature advanced mechanisms to process input and output data for the Cloud. The main avenue of thought for the advanced interfaces is to utilise autonomous software agents and their characteristics.

This paper is structured as follows. In Section 2, we present the conceptual and technical background as well as related work. Section 3 introduces the approach and methodology for moving net simulations to the Cloud. Section 4 proposes the different kinds of interfaces. Finally Section 5 discusses the approach and Section 6 concludes the paper and presents future work.

2 Background and Related Work

In this section we are going to discuss the conceptual and technical background for this work. For the specification of workflows Petri nets and especially *reference nets* are employed. Related work is also presented.

2.1 Reference Nets

Reference nets were introduced in 2002 by Olaf Kummer (see [14]). Reference nets are modelled and simulated using the RENEW editor and simulation tool [16]. Both are described in the RENEW manual². In reference nets, tokens can be

¹ We use the terms simulation and execution interchangeably. If a distinction has to be made it will be clear from the context.

² The latest version of RENEW, documentation and articles are available on (<http://www.renew.de>)

anonymous, basic data types or references to Java objects or other reference nets. The referenced objects can be of any class in the Java programming language.

Firing a transition can also create a new instance of a subnet in such a way that a reference to the new net will be put into a place as a token. This allows for a specific, hierarchical nesting of networks, which is helpful for building complex systems in this formalism. The creation of instances is similar to object instances in object-oriented programming languages and the usage of references allows to construct reference net systems, whose structures are not fixed at build time.

2.2 Renew

As mentioned above, we use RENEW for the modelling of workflows. RENEW is a graphical tool for creating, editing and simulating reference nets. It combines the ‘nets within nets’ paradigm of reference nets with the implementation power of Java. The RENEW plug-in architecture, which was developed and introduced in [22], allows the extension of RENEW with additional functionality through the use of interfaces between RENEW components without changing the core of RENEW. Additional functionality (e.g. additional net formalisms, simulation and verification tools, interface extensions) can be added to RENEW by providing the Java classes and nets for the new plug-in. Many such plugins have already been developed, which makes RENEW a versatile and extensive Petri net tool.

2.3 Agents

We also utilise software agents for advanced features regarding the interface to the Cloud execution (see Section 4.3). We use the MULAN (**M**ulti **A**gent **N**ets [21]) reference architecture and its implementation CAPA (**C**oncurrent **A**gent **P**latform **A**rchitecture [10]). Both have been created and implemented using RENEW and the majority of the executable code are in fact reference nets. Agents are executed in a distributed environment and generally communicate via standardised asynchronous messages. They can feature intelligence, reactive and proactive behaviour, and autonomy. These kinds of properties are utilised for the Cloud execution.

2.4 Related Work

Originally, WfMS were not conceived to be used in Cloud-like environments. With the growth of Cloud computing, several traditional WfMS improved their kernel and are now able to provide interfaces to communicate with external Cloud services. The prevalent (scientific) WfMS are: Taverna[18], Pegasus[9], Triana[23], Askalon[11], Kepler[2] and the General Workflow Execution Service (GWES)[1]. The originality of these systems is that they run on parallel and distributed computing systems in order to reach a high level of performance and get access to wide range of external resources. The Pegasus system allows scientists to execute workflows in different resources including clusters, Grids and Clouds.

This has been adapted later to execute scientific workflows in the Cloud (within an Amazon EC2 Instance) [17]. Compared to our work, the *migration* to the Cloud is almost the same, the difference lies at the modelling level, where we use Reference nets as modelling technique. GEWES is an interesting project that makes use of high-level Petri Nets (HLPN) for the description of workflows. The GWES coordinates the composition and execution process of workflows in arbitrary distributed systems, such as SOA, Cluster, Grid, or Cloud environments. In the workflow specifications, transitions represent tasks and tokens represent data flowing through the workflow.

There have also been many more efforts to infuse Cloud and distribution aspects into general workflow management. The ADEPT project [8, 20] focuses on flexible and adaptive workflow management but also deals with distribution and migration aspects to avoid performance bottlenecks in the network. Another interesting combination of Clouds and workflows is the OpenTosca project [5]. It utilises management plans implemented as workflows to configure Cloud applications for organisations. [24] also deals with configuration issues but focuses explicitly on the configuration of interorganisational business processes in the Cloud. The issues addressed by these and more publications represent advanced features of workflows in Clouds. They are outside the scope of this paper. Some of these issues are, however, considered future work.

3 Renew in the Cloud

RENEW *in the Cloud* designs the process of simulating Petri net models not locally (i.e. on-premises) but in the Cloud. There are different reasons why we are moving the simulation to other execution environments but the main reason is to seek gains in performance. Especially (Petri net) models that contain complex and time consuming tasks are of interest here. In our approach the design/modelling step is performed at the user's side since it does not require computing or storage capabilities. After this, the models are pushed to the Cloud provider. The Cloud provider should be able to provide instances, that support Petri net simulations. Therefore, Cloud instances need to be *provisioned* by external Petri net editors and simulators. Since our chosen editor is RENEW it will be installed and configured before starting the simulation. The whole process consists of the following steps:

1. modelling the workflow
2. configuring the Cloud instance
3. starting/connecting to the Cloud instance
4. uploading the required nets
5. executing the simulation and getting the results

Technically, our work is based on the Vagrant tool³, which permits us to create reproducible development environments. According to the Vagrant homepage, Vagrant *"is a tool for building complete development environments. With*

³ <https://www.vagrantup.com/>

an easy-to-use workflow and focus on automation". There are three ways to use Vagrant: with a virtual machine, a Cloud provider, or with VMware.

For creating a Vagrant machine the vagrant tool first needs to be installed as well as the VirtualBox. For both Vagrant and VirtualBox, the installation is possible on the three famous operating systems: Linux, Windows and Mac. Next, a configuration file called *Vagrantfile* is mandatory to configure a Vagrant machine. It is a Ruby file used to configure Vagrant and to describe virtual machines required for a project as well as how to configure and provision these machines. Finally, the guest Vagrant host can be started using the command *vagrant up*.

3.1 First Prototype (with VirtualBox)

To run RENEW and all required software on the host machine, configuration using a Vagrantfile is needed. The latter permits to provision the host machine(s) with additional softwares (in our case RENEW). Since RENEW requires Java 6 or later, this portion of code shows instructions that should be added.

Figure. 2 shows the steps to follow for the execution (simulation) of a workflow (Petri net). First of all, the workflow is modelled using RENEW and generates .rnw files. It should be noted that, for now, we focus solely on the simple *execution* of workflow nets in the Cloud. Workflow *management* aspects are currently considered in the background. For example, human interaction with the workflow, e.g. a user executing a task, is currently only simulated by the system. Later on it is possible to incorporate a workflow management system in the Cloud which would support these kinds of aspects. Workflow management within RENEW implemented as a reference net (agent) system, which would be executed in the Cloud, is already possible [25]. For now, the vagrant machine is equipped with a RENEW version without a graphical user interface, i.e that we are obligated to run the simulation with the command line. The correspondent console command is *startsimulation*. The syntax of the command is:

startsimulation <net system> <primary net> [-i]

- *net system*: The compiled net files (.sns files, Shadow Net System).
- *primary net*: The name of the net, of which a net instance shall be opened when the simulation starts. Using the regular GUI, this equals the selecting of a net before starting the simulation.
- *-i*: This must be set before starting the simulation (only for this prototype). Concretely, we use -r, which means to run the whole simulation without steps. More information about this command can be found in [15, p.106].

For testing purposes we created a simple net (primary net) that contains a single transition that prints a string on the screen. Since the reference net formalism allows using java code, this is done simply by the instruction `System.out.println("message")` (see Figure. 3). Once the required files are prepared (.rnw and .sns), they are sent to the Vagrant machine. The nets are either copied to the synced directory with the Vagrant machine or with *scp*. To start the simulation on the guest machine, there are three possibilities: (i) by a command

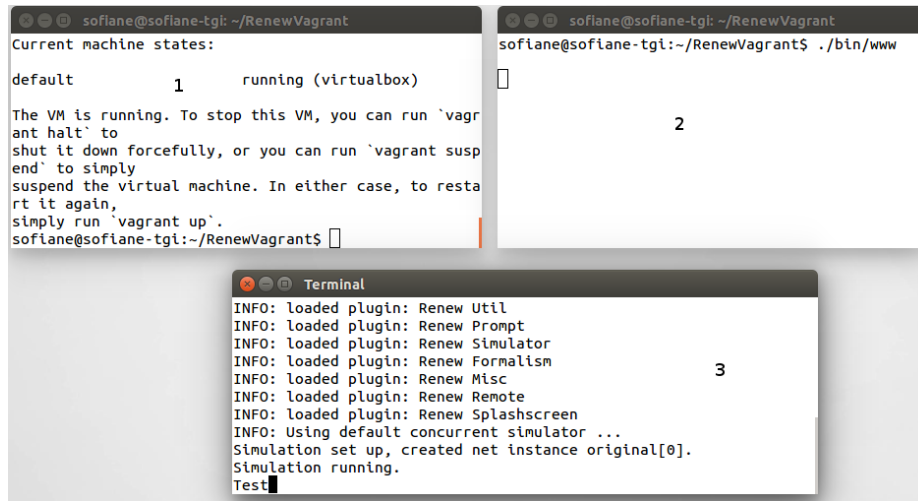


Fig. 1. Run Simulation in a Vagrant Machine

line (using `nohup` and `ssh`) (ii) through a web Gui (using NodeJS) (iii) from a reference net directly (inscribed to transitions). Figure 1, shows the process of starting a vagrant machine and launching RENEW and the simulation. Executing the command in **2**, launches a new terminal and starts RENEW and simulate the net on the Vagrant machine. (1) The Vagrant machine should be up and running (2) The web server (NodeJS) is started (3) RENEW is launched and a simulation is started with the required nets.

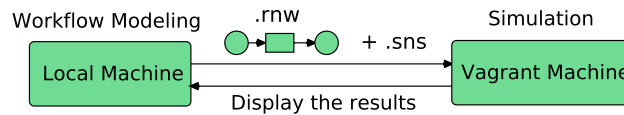


Fig. 2. Remote Simulation with Vagrant

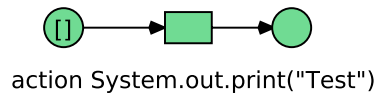


Fig. 3. The Original Net (.rnw)

3.2 Second Prototype (with OpenStack)

The second version of the implementation is based on a concrete Cloud environment. The instances are not launched in a virtual machine at the host machine, but in a Cloud (see Figure. 5). We mentioned before that *Vagrant* uses specific providers. The default one is *VirtualBox*⁴. Other built-in providers are VMWare⁵, Docker⁶ and Hyper-V⁷. When executing *vagrant up* we will have a virtual machine created on the local host. If we require only one VM then it is enough to work locally. Nevertheless, when the number of VMs grows we will face an overload due to a lack of resources. The natural solution is to look for external resources which, in our case, are available in a Cloud. Due to financial and technical constraints, in our testbed we use an open source Cloud framework called *OpenStack*⁸. *OpenStack* is an open source software for creating private and public Clouds. It is installed on a CentOS Linux operating system. Thanks to the plug-in architecture that *Vagrant* is based on, we are able to connect to different Cloud providers and launch our instances. This is performed by a plug-in called *vagrant-openstack-provider*⁹. This plug-in permits to control and provision machines within an OpenStack Cloud. Other features are for instance: Create and boot OpenStack instances, SSH into the instances and suspend and resume instances. The principles for running RENEW simulation in the Cloud are almost the same as presented in the previous section. We still need to upload the required nets (.rnw and .sns) to the VM. The difference is at the configuration level, which is realised by the *Vagrantfile*. A minimal configuration consists of the following:

```
require 'vagrant-openstack-provider'
Vagrant.configure('2') do |config|
  config.vm.box = 'openstack'
  config.ssh.username = 'stack'
  config.vm.provider :openstack do |os|
    os.openstack\__auth\__url = 'http://keystone-server.net/v2.0/tokens'
    os.username = 'openstackUser'
    os.password = 'openstackPassword'
    os.tenant\__name = 'myTenant'
    os.flavor = 'm1.small'
    os.image = 'ubuntu'
    os.floating\__ip\__pool = 'publicNetwork'
  end
end
```

⁴ www.virtualbox.org

⁵ www.vmware.com

⁶ www.docker.com

⁷ www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx

⁸ www.openstack.org

⁹ <https://github.com/ggiamarchi/vagrant-openstack-provider>

The configuration presented above concerns only the credentials and the image used to boot the instances. The important next step is to configure these instances to be able to handle RENEW simulations. The configuration is performed exactly in the same way as working with virtual machines (VirtualBox). Configuring instances plays an important role and directly affects the performance of the system. Although, for testing purpose, we worked on a private OpenStack Cloud, our implementation can be integrated within commercial Cloud providers like Amazon¹⁰, Windows Azure¹¹ or HP¹². With providers, Cloud consumers can configure their instances based on a pay-as-you-go model. Resources provided by commercial Cloud providers are not free, which can negatively affect the choice of the Cloud consumers. With respect to the application requirements, there are different types of instances which depend on the Cloud provider. Instance types describe the compute, memory and storage capacity of the instances that Cloud consumers use for hosting (computing) their applications. Therefore, the requirements for the applications should be clearly specified as QoS parameters. This issue has been already addressed in [3]. QoS parameters can be specified as inputs to the transitions. For example, with OpenStack these are called by names such as “*m1.large*” or “*m1.tiny*”. Figure. 4 shows the characteristics of T2 instances.

Model	vCPU	CPU Credits / hour	Mem (GiB)	
t2.micro	1	6	1	E
t2.small	1	12	2	E
t2.medium	2	24	4	E

Fig. 4. Amazon T2 Instance Characteristics

¹⁰ <http://aws.amazon.com/>

¹¹ <http://azure.microsoft.com>

¹² <http://www.hpcloud.com/>

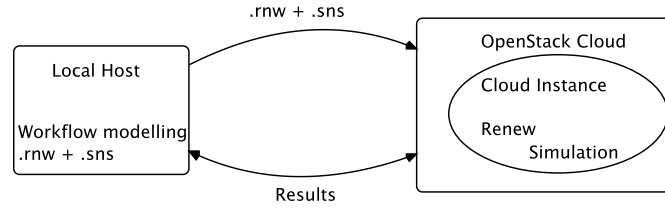


Fig. 5. RENEW Simulation in the OpenStack Cloud

4 Interface

In the previous section we have described how to enable RENEW simulations in a Cloud environment. The basic technical realisation bundles up and simply executes a workflow net system and its shadow net representation. In order to practically utilise this execution we need to define an interface for it. We have examined possible interfaces that can be grouped into three categories: Simple, Simulation, Advanced. These categories will be discussed in Sections 4.1 through 4.3.

Prototypes for the simple interfaces already exist. More features for these interfaces as well as the simulation and advanced interfaces are currently under development. They will be discussed on a conceptual level here.

Note that how exactly a simulation as a Cloud functionality is called has already been discussed in the previous section. Generally it can be called either via the console, a web interface or directly within a (local) running net system. If an interface restricts these possibilities it will be shortly addressed.

4.1 Simple Interfaces

Simple interfaces offer basic, yet versatile functionality that can later be utilised in more complex settings. The input for simple interfaces remains the workflow system and its shadow net representation. The output options vary, but share that any results obtained are returned as simple data or objects. Simple interfaces do not support any kind of intelligence or autonomy. They are simply called when needed and report back the predefined results.

Console Interface This interface uses either the internal RENEW console or the general system console as the output medium. Consequently it already directly works with reference nets. By simply inscribing a `System.out.println(textVariable)` to any transition of the net system being executed in the Cloud the String representation of the object `textVariable` is printed on the console. Figure. 1 already shows a working prototype using such a console interface.

For very simple use cases (e.g. testing a certain outcome of the net system) this is already sufficient, but in most cases any obtained result should automatically be made available to the caller in a more utilisable way. This can be realised

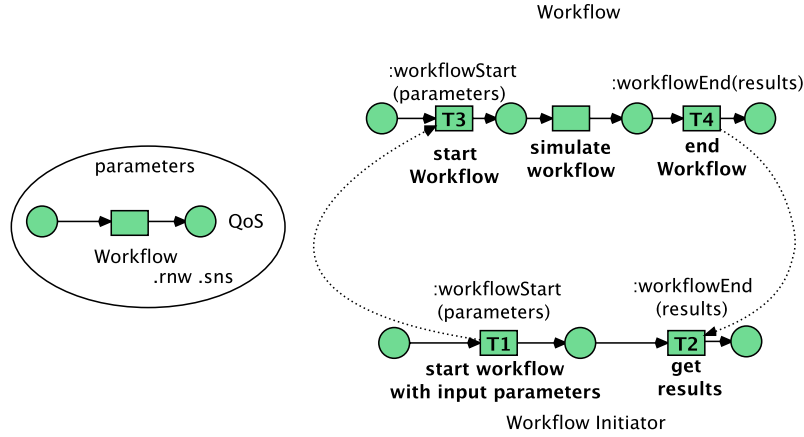


Fig. 6. Synchronous Channels Interface

by reading any output in the console and combining these outputs into a result object that is passed back when the execution has been completed. This way more complex use cases and computations can also be supported even with this very simple interface. One problem with this approach is that it is not standardised or regulated by the modelling approach. This is a general problem that will be discussed in Section 5.

Synchronous Channel Interface Realising the interface through synchronous channels is another way of providing a simple interface. Synchronous channels, in general, are a mechanism to allow data and object transfer between net instances. They were first introduced in [7] and are fundamental to the reference net formalism. Within the Cloud context synchronous channels allow for data objects created and modified during the execution of a workflow to be transferred back to its initiator or even directly into other running (local or remote) net systems. Consequently, the full potential is realised when the Cloud call is incorporated into a net system. There are a number of ways in which synchronous channels can be incorporated into an interface for Cloud-based workflows. The simplest way is to explicitly inscribe an output channel to a transition in the net. When this transition fires the synchronous channel is called and the specified data object transferred to the Cloud call initiator. By extending this to multiple transitions we can realise a kind of continuous feedback for the initiator. Whenever a transition inscribed with the synchronous channel would fire a result would be send to the initiator.

Figure 6 illustrates the approach mentioned above. There are two main nets: *Workflow Initiator* and *Workflow*. The Workflow Initiator manages the workflow locally and is responsible for the communication with the Cloud provider. On the other side, the Workflow is executed in the Cloud. After modelling the workflows, the model is saved in RENEW (`.rnw`) and Shadow net (`.sns`) files. These files are

required for the Workflow to be executed. The communication between both nets is possible through synchronous channels. For instance, T1 and T4 are for sending data; T2 and T3 are for receiving data. Furthermore, all the parameters can be put into a place instead of synchronous channels.

There are two main problems when using synchronous channels. First of all, similar to the console interface, this interface is not structured. Careless modellers may set output channels to incorrect transitions so that results may not be valid. Another issue is related to the continuous update mechanism. If (possibly partial) results are transferred back to the initiator at multiple times, it may be difficult to work with these results. Depending on the net a modeller would have to explicitly build against that specific interface in order to aggregate the results into a valid composition. For this simple interface it would be cumbersome and inefficient. This is one of the issues addressed by the advanced interfaces described in Section 4.3.

Up until now, synchronous channels have only been discussed for output scenarios. Incorporating synchronous channels for the input of the Cloud-based workflows is also possible. In the simplest option this would only be used to incorporate initial input data. This would not change the basic functionality all too much, as initial data can easily be supplied via the console or simply as the initial marking of the workflows. It would make it easier though to change the initial marking. If called from a running net system the Cloud workflow could be initiated with runtime information. A synchronous input channel would simply pass the data object directly into the workflow in the Cloud. Without synchronous channels a new net system with the specified initial marking would have to be created or the console call would have to be tailored to the runtime information.

It is also possible to transfer data into the running Cloud workflow. This would require the initiator to be able to maintain a connection with the Cloud system. This is mostly feasible when the Cloud call is initiated by a running net system which would continue with its own execution and provide additional data to the Cloud net system at some later point. Certain transitions in the Cloud net system could then be inscribed with an input channel over which this additional data could be received. Ensuring the correct connection and synchronisation between local and Cloud net systems is the main challenge in this context. This is currently considered future work and outside of the scope of this paper.

Using synchronous channels in the proposed ways has some disadvantages though. Without any restrictions to modelling the placement of input and output in the net would affect any verification of workflow correctness or other Petri net properties. This is discussed further in Section 5.

4.2 Simulation Interfaces

The simulation interfaces are not so much interfaces, as they are a utilisation of RENEW in a Cloud environment. Instead of executing a net system remotely once for some direct usage these interfaces execute the net system a large number of times. The information about these simulation runs is then reported back to

the initiator. This constitutes the output of these interfaces. The input consists, beside the net system and shadow net representation, of simulation parameters (e.g. number of simulation runs). The advantage of running these simulations in a Cloud environment is that it frees up the modellers local machine.

Result Simulation One possibility is to run a set of simulations and have the system report back the results of each run. With the same initial marking different simulations may still produce different results. This could be due to race conditions, non-deterministic behaviour, etc. With these results the modeller could validate assumptions about the net system or determine possible error sources.

This kind of simulation could be extended by enabling variable initial markings. Simulating a net system with differing parameters might influence the results and help modellers even more.

Timed Simulation Another possibility is to run a set of simulations and compare the time it takes to complete them. This kind of simulation is more useful for testing the performance or new features in the runtime environment, in our case RENEW. Running the simulation with new features enabled and comparing the results obtained without them can yield information about new algorithms.

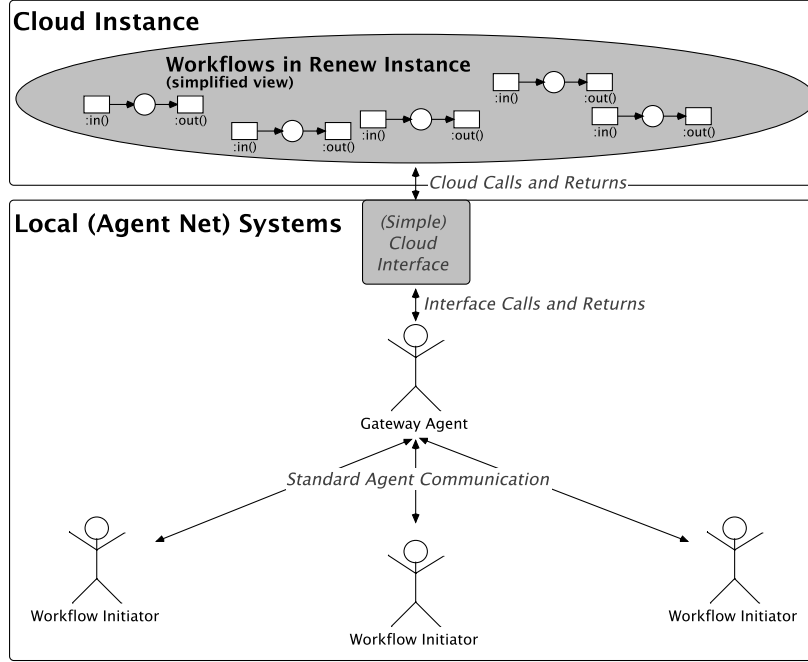
Focussing more on the performance of the net system it might be of interest to the modeller to determine the impact of different initial markings. Varying over the initial marking of the net system could then help modellers determine performance bottlenecks. When using (reference) Petri nets for processes in practical software engineering within the PAOSE (PETRI NET-BASED, AGENT-AND ORGANIZATION-ORIENTED SOFTWARE ENGINEERING [6]) development approach for example, such simulations and their results become especially useful and interesting.

4.3 Advanced Interfaces

The advanced interfaces go beyond simple call interfaces like the ones discussed in Section 4.1. They utilise these simple interfaces but add another layer of abstraction to them. This leads to additional characteristics like certain degrees of intelligence and autonomy. They can also feature mechanisms to manage and store known net systems so that they may even serve as a kind of directory service. They can also aggregate results, enforce quality of service concerns or choose the best from a set of results. Consequently, no general statements about input and output can be made.

Agent Interface Using agents for an advanced interface to the Cloud execution of Petri net systems has a number of intrinsic advantages. Agents possess autonomy and a certain degree of intelligence. Reactive and proactive agent behaviour can also be utilised.

In an advanced interface an agent would serve as a kind of gateway between the local net systems and the Cloud net systems. For the MULAN and CAPA

**Fig. 7.** Agent Interface Illustration

agents we utilise this would expand upon the ideas introduced by the WebGateway agent [4] towards Cloud calls. The WebGateway agent serves as a kind of bridge between the net execution of a RENEW environment and the web environment. Agents in RENEW can then offer their functionality as web services and also access remote web services.

For the Cloud context agents would serve in a similar fashion. The idea is illustrated in Figure 7. Some agents would be responsible for the net systems. They would take on the role of the initiator. They could act autonomously or be controlled by a human user via some kind of user interface.

These agents would control and/or create the workflows which should be executed in the Cloud. They would send requests and data to the gateway agent¹³. The gateway agent would then use a simple interface (see above) in its internal functionality to initiate the workflow in the Cloud on behalf of the other agents. Any result obtained in the Cloud would be send back to the gateway agent which would then forward it to the other agents.

¹³ Alternatively the workflows could be stored in a database known to all agents. In that case the initiator agents would simply send requests and identifiers of the workflows to the gateway agent.

At this point the characteristics and advantages of software agents can be utilised. In the following we will cover some ideas of how, starting from the relatively simple approach described above, this can be done.

The gateway agent can aggregate the results of the Cloud calls into more informative composite results. Partial results could be incorporated into the workflows with standardised instructions for the gateway agents to combine them after the execution has been completed. The gateway agent can also instantiate the workflow multiple times and choose the best (or fastest) result. Of course, the gateway agent has to be equipped with mechanisms to aggregate or assess results in these fashions. This is, however, simply a question for the technical implementation and not a conceptual one. Aggregation of results is especially interesting for simulation purposes. The gateway agent could automatically create composite results for modellers to inspect. It could also automatically vary over the initial parameters based on the initial results (e.g. to validate results or test certain outlier data).

The gateway agent can also react to errors or other problems occurring during the execution in the Cloud. If the Cloud execution returns an error the gateway agent can retry the instantiation. If the error was caused by the call it can also adapt the call (e.g. if input parameters had incorrect types like a string representation of an integer value). This would happen transparently to the initiator of the call which would only have to be involved if the gateway agent was unable to find a solution to the problem.

Using proactive behaviour the agent can also support the execution of Cloud workflows. For example, it could restart workflows if the returned result strongly deviated from expected results. Or it could prepare or even already initiate recurring net executions.

The gateway agent can also handle quality of service (QoS) concerns. As stated in 3, QoS are specified as parameters either in transitions or places. The second scenario is the more appropriate since it uses synchronous channels. In this situation, in addition to the workflow model (and its related files .rnw and .sns) modellers also include QoS parameters. In this work, we focus on time and budget, but modellers can include other constraints. The gateway agent can consequently play another role, which is Cloud brokering. By brokering we mean that the agent looks for the suitable Cloud provider to execute the workflow based on its requirements. This can be useful when working with multiple Clouds.

One disadvantage of using a gateway agent for the Cloud is that it centralises the communication. This decouples the communication aspects from the individual agents, but gives the system a single point of failure. Only one agent in the system, the gateway agent, possesses the functionality and mechanisms to invoke Cloud systems. This makes other agents simpler and possibly more efficient to execute, but if the gateway agent fails communication with the Cloud is lost. This could be remedied by implementing a solution with multiple gateway agents and distributing the functionality. If one gateway agent failed others could take its place.

Entity Interface The term entity describes a hybrid construct between an agent and a workflow. Depending on the runtime needs they can act as an agent (e.g. for communication), a workflow (e.g. for task deployment and execution) or something between the two (e.g. as a mobile process). Entities and modelling with them is currently ongoing research. The Cloud context enhances the capabilities of entities in many regards.

From the interface point of view an entity possesses all the characteristics of agents and has access to the entire functionality described in the previous paragraph for an advanced interface provided through an agent. But this interface is extended even more because of the additional possibilities gained through the workflow properties of an entity. Entities are, in one perspective, a (workflow) process. This automatically entails a certain behaviour-centric structure and purpose to the modelling.

By structuring the calls and instantiations of the Cloud net systems as a process itself the modeller is directly supported. While anything can be achieved through regular, less-rigidly structured modelling, restricting the modeller into such a process perspective is still beneficial. Considering process order, task subdivisions, processing of partial results and other aspects of a process are direct requirements in this perspective. Consequently they are obligatory to the modeller here. But that means that these aspects, which range from helpful to essential, can also not be ignored or omitted. This is what the entities add on a conceptual level to the advanced interface of agents.

5 Discussion

One issue that was raised in Section 4 concerned the restrictions on modelling and the placement of input and output in a net for the interfaces. If that placement is unrestricted it may be error-prone and puts the responsibility solely on the modeller without any support. An effort could be made to restrict input and output to the initial and exit places of the workflow. This would ensure only full results are returned to the caller and make it easier to verify workflow net properties. However, there are cases in which partial results (e.g. status updates) *during* the execution of a workflow net are desirable. The restriction would preclude this. A compromise would be to allow simple status reports from anywhere in the net (e.g. via the console), but only complete results from the final place or transition of the workflow (e.g. via synchronous channel). Only these complete results would then be made available for further operations in the workflow initiator.

Without any restrictions it would also be impossible to make any statements about the correctness of the executed workflows. For practical purposes allowing input into already running workflows and arbitrary input/output locations might be helpful to some use-cases. But from a verification and validation point-of-view these mechanisms are problematic. Incorporating concepts like workflow correctness into the Cloud calls and interfaces is currently ongoing work but outside the scope of this paper.

The question of restrictions raises another interesting point. This paper is focused on the execution of workflow nets. Arbitrary workflow net systems can be executed in the Cloud. That includes scientific and interorganisational workflows.

From a technical standpoint though, it is possible to execute any net system in the Cloud. The only precondition is that a plugin for the net formalism in question is provided for the RENEW instance running in the Cloud. RENEW plugins for many formalisms already exist (e.g. P/T nets, nets supporting time annotations) and more can be added.

When allowing arbitrary net systems without restrictions to the interface or without any structured modelling these arbitrary net systems might pose challenging to modellers in terms of efficiency and manageability. For this reason it is advisable to use structured modelling paradigms, like agents or entities, for the Cloud net systems as well. In the following paragraphs we will examine how this would affect the advanced interfaces described in the previous section.

By executing the agent interface within the Cloud (as opposed to outside the Cloud as described in Section 4.3) the communication can be simplified. In this scenario the net system executed in the Cloud is a CAPA agent platform with a running gateway agent. The gateway agent is accessible for other agents via the standardised FIPA compliant asynchronous message communication supported in CAPA. This would “move” the interface from the local execution into the Cloud, since to other agents it does not matter where the gateway is executed. They communicate with him in the same way as any other local or remote agent. This would lead to efficiency gains as the gateway agent could access resources in the Cloud environment directly. The technical capabilities of the gateway agent would also be improved. Other properties of the interface would largely remain the same.

The entity interface would benefit in the same way as the agent interface. In addition it would also affect the modelling abstraction of the entity, as it could be considered a (workflow) process in the Cloud executing other (workflow) processes. This is especially interesting in the interorganisational workflow setting which we are researching for entities. The entity in the Cloud could be considered as the overall interorganisational workflow while the workflows it controls are the subworkflows for each involved organisation.

6 Conclusion

In this paper we presented our approach for moving net executions to the Cloud. The paper described the technical aspects, implementation and methodology. From a technical point of view it is possible to execute any net system supported by RENEW in the Cloud. However, for the purpose of this paper we focused on workflows. For this context the notion of Cloud interfaces was introduced. These interfaces can be classified as simple, simulation and advanced depending on how the communication and the transfer of data are performed. Furthermore, we discussed the integration of agent concepts in order to provide gateways to the Cloud.

Direct future work is related to agents and especially the entity concept. This paper described how agent and the entity concepts can realise advanced interfaces for the Cloud net systems. The other direction is currently also being researched. In general, opening up the capabilities of entities to Cloud functions is already beneficial in of itself. But agents and especially entities can also feature very complex behaviour. In fact, some processes of entities can be regarded as fully-fledged subsystems. Relocating these subsystems to the Cloud can improve the performance of entity systems greatly.

Concerning workflow complexity, we are also currently working on a concrete scientific workflow application. This application is related to the remote sensing domain, especially image processing of satellite imagery. Most of the work has been achieved: we have implemented an image processing tool that allows modelling and execution of remote sensing applications specified by reference nets. The next natural step is to execute those workflows in the Cloud based on the results presented in this paper. Furthermore, this work should be evaluated in terms of performance. This concerns running several simulations in parallel (in different virtual machines) in the Cloud.

In conclusion, the realisation of RENEW in a Cloud opens up a number of advantages w.r.t. performance, availability, flexibility, etc. Some of these have already been discussed in this paper. Other will become more noticeable with the ongoing work. The continued incorporation of the Cloud aspects with complex workflow and agent systems is just one of the possible avenues of thought, albeit the most promising one currently.

References

1. Martin Alt, Sergei Gorlatch, Andreas Hoheisel, and Hans-Werner Pohl. A Grid Workflow Language Using High-Level Petri Nets. In Roman Wyrzykowski, Jack Dongarra, Norbert Meyer, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3911 of *LNCS*, pages 715–722. Springer-Verlag, 2006.
2. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 423–424, June 2004.
3. Sofiane Bendoukha and Lawrence Cabac. Cloud transition for QoS modeling of inter-organizational workflows. In Daniel Moldt, editor, *Modeling and Business Environments MODBE'13, Milano, Italia, June 2013. Proceedings*, volume 989 of *CEUR Workshop Proceedings*, pages 355–356. CEUR-WS.org, June 2013.
4. Tobias Betz, Lawrence Cabac, Michael Duvinneau, Thomas Wagner, and Matthias Wester-Ebbinghaus. Software Engineering with Petri Nets: A Web Service and Agent Perspective. In Maciej Koutny, Serge Haddad, and Alex Yakovlev, editors, *Transactions on Petri Nets and Other Models of Concurrency IX*, Lecture Notes in Computer Science, pages 41–61. Springer Berlin Heidelberg, 2014.
5. Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*, volume 8274 of *LNCS*, pages 692–695. Springer Berlin Heidelberg, December 2013.

6. Lawrence Cabac. Multi-agent system: A guiding metaphor for the organization of software development projects. In Paolo Petta, editor, *Proceedings of the Fifth German Conference on Multiagent System Technologies*, volume 4687 of *Lecture Notes in Computer Science*, pages 1–12, Leipzig, Germany, 2007. Springer-Verlag.
7. Søren Christensen and N. D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In Valette, R., editor, *Lecture Notes in Computer Science; Application and Theory of Petri Nets 1994, Proceedings 15th International Conference, Zaragoza, Spain*, volume 815, pages 159–178. Springer-Verlag, 1994.
8. Peter Dadam and Manfred Reichert. The adept project: A decade of research and development for robust and flexible process support - challenges and achievements. *Computer Science - Research and Development*, 23(2):81–97, 2009.
9. Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Sci. Program.*, 13(3):219–237, July 2005.
10. Michael Duvigneau. Bereitstellung einer Agentenplattform für petrinetzbasierte Agenten. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2002.
11. T. Fahringer, R. Prodan, Rubing Duan, F. Nerieri, S. Podlipnig, Jun Qin, M. Siddiqui, Hong-Linh Truong, A. Villazon, and M. Wiczorek. Askalon: a grid application development and computing environment. In *Grid Computing, 2005. The 6th IEEE/ACM International Workshop on*, pages 10 pp.–, Nov 2005.
12. Christina Hoffa, Gaurang Mehta, Timothy Freeman, Ewa Deelman, Kate Keahey, G. Bruce Berriman, and John Good. On the use of cloud computing for scientific workflows. In *eScience*, pages 640–645. IEEE Computer Society, 2008.
13. G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66, dec. 2009.
14. Olaf Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
15. Olaf Kummer, Frank Wienberg, Michael Duvigneau, and Lawrence Cabac. *Renew – User Guide (Release 2.4)*. University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg, April 2013. Available at: <http://www.renew.de/>.
16. Olaf Kummer, Frank Wienberg, Michael Duvigneau, Michael Köhler, Daniel Moldt, and Heiko Rölke. Renew – the Reference Net Workshop. In Eric Veerbeek, editor, *Tool Demonstrations. 24th International Conference on Application and Theory of Petri Nets (ATPN 2003). International Conference on Business Process Management (BPM 2003)*, pages 99–102, June 2003.
17. A. Nagavaram, G. Agrawal, M.A. Freitas, K.H. Telu, G. Mehta, R.G. Mayani, and E. Deelman. A cloud-based dynamic workflow for mass spectrometry data analysis. In *E-Science (e-Science), 2011 IEEE 7th International Conference on*, pages 47–54, Dec 2011.
18. Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:2004, 2004.
19. Tim Grance Peter Mell. The nist definition of cloud computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, 2011. <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>.
20. Manfred Reichert, Thomas Bauer, and Peter Dadam. Flexibility for distributed workflows. In Minhong Wang and Sun Zhaohao, editors, *Handbook of Research on*

- Complex Dynamic Process Management: Techniques for Adaptability in Turbulent Environments*, pages 137–171. IGI Global, Hershey, New York, July 2009.
21. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
 22. Jörn Schumacher. Eine Plugin-Architektur für Renew – Konzepte, Methoden, Umsetzung. Diploma thesis, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, October 2003.
 23. Ian Taylor, Matthew Shields, Ian Wang, and Omer Rana. Triana applications within grid computing and peer to peer environments. *Journal of Grid Computing*, 1(2):199–217, 2003.
 24. W.M.P. van der Aalst. Business process configuration in the cloud: How to support and analyze multi-tenant processes? In *Web Services (ECOWS), 2011 Ninth IEEE European Conference on*, pages 3–10, Sept 2011.
 25. Thomas Wagner. A centralized Petri net- and agent-based workflow management system. In Michael Duvigneau and Daniel Moldt, editors, *Proceedings of the Fifth International Workshop on Modeling of Objects, Components and Agents, MOCA'09, Hamburg*, number FBI-HH-B-290/09 in Bericht, pages 29–44, Vogt-Kölln Str. 30, D-22527 Hamburg, September 2009. University of Hamburg, Department of Informatics.

Modelling the Behaviour of Management Operations in Cloud-based Applications^{*}

Antonio Brogi, Andrea Canciani, Jacopo Soldani, and PengWei Wang

Department of Computer Science, University of Pisa, Italy

Abstract. How to flexibly manage complex applications over heterogeneous clouds is one of the emerging problems in the cloud era. The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) aims at solving this problem by providing a language to describe and manage complex cloud applications in a portable, vendor-agnostic way. TOSCA permits to define an application as an orchestration of nodes, whose types can specify states, requirements, capabilities and management operations — but not how they interact each another.

In this paper we first propose how to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities. We then illustrate how such behaviour can be naturally modelled, in a compositional way, by means of open Petri nets. The proposed modelling permits to automate different analyses, such as determining whether a deployment plan is valid, which are its effects, or which plans allow to reach certain system configurations.

1 Introduction

Available cloud technologies permit to run on-demand distributed software systems at a fraction of the cost which was necessary just a few years ago. On the other hand, how to flexibly deploy and manage such applications over heterogeneous clouds is one of the emerging problems in the cloud era.

In this perspective, OASIS recently released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA [19,20]), a standard to support the automation of the deployment and management of complex cloud-based applications. TOSCA provides a modelling language to specify, in a portable and vendor-agnostic way, a cloud application and its deployment and management. An application can be specified in TOSCA by instantiating component types, by connecting a component's requirements to the capabilities of other components, and by orchestrating components' operations into plans defining the deployment and management of the whole application.

Unfortunately, the current specification of TOSCA [19] does not permit to describe the behaviour of the management operations of an application. Namely, it is not possible to describe the order in which the management operations of

^{*} This work has been partly supported by the EU-FP7-ICT-610531 project SeaClouds.

a component must be invoked, nor how those operations depend on the requirements and affect the capabilities of that component. As a consequence, the verification of whether a plan to deploy an application is valid must be performed manually, with a time-consuming and error-prone process.

In this paper, we first propose a way to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities. We define how to specify the management protocol of a TOSCA component by means of finite state machines whose states and transitions are associated with conditions on (some of) the component's requirements and capabilities. Intuitively speaking, those conditions define the consistency of component's states and constrain the executability of component's operations to the satisfaction of requirements.

We then illustrate how the management protocols of TOSCA components can be naturally modelled, in a compositional way, by means of open Petri nets [2,14]. This allows us to obtain the management protocol of an arbitrarily complex cloud application by combining the management protocols of its components. The proposed modelling permits to automate different analyses, such as determining whether a deployment plan is valid, which are its effects, or which plans allow to reach certain system configurations.

The rest of the paper is organized as follows. Sect. 2 introduces the needed background (TOSCA and open Petri nets), while Sect. 3 illustrates a scenario motivating the need for an explicit, machine-readable representation of management protocols. Sect. 4 describes how TOSCA can be extended to specify the behaviour of management operations, how such behaviour can be naturally and compositionally modelled by means of open Petri nets, and how the proposed modelling permits to automate different types of analysis. Related work is discussed in Sect. 5, while some concluding remarks are drawn in Sect. 6.

2 Background

2.1 TOSCA

TOSCA [19] is an emerging standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. In this perspective, TOSCA provides an XML-based modelling language which allows to specify the structure of a cloud application as a typed topology graph, and deployment/management tasks as plans. More precisely, each cloud application is represented as a **ServiceTemplate** (Fig. 1), which consists of a **TopologyTemplate** and (optionally) of management **Plans**.

The **TopologyTemplate** is a typed directed graph that describes the topological structure of the composite cloud application. Its nodes (**NodeTemplates**) model the application components, while its edges (**RelationshipTemplates**) model the relations between those application components. **NodeTemplates** and **RelationshipTemplates** are typed by means of **NodeTypes** and **RelationshipTypes**, respectively. A **NodeType** defines (i) the observable properties of an application component C , (ii) the possible states of its instances, (iii) the requirements

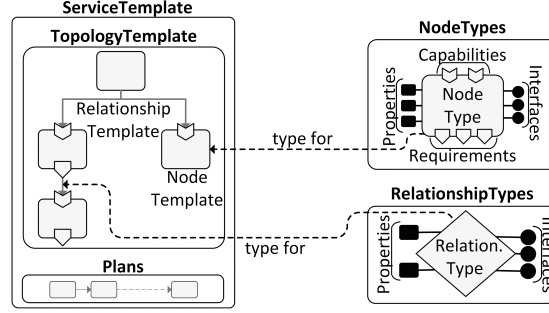


Fig. 1. TOSCA ServiceTemplate.

needed by C , (iv) the capabilities offered by C to satisfy other components' requirements, and (v) the management operations of C . **RelationshipTypes** describe the properties of relationships occurring among components. Syntactically, properties are described by **PropertiesDefinitions**, states by **InstanceStates**, requirements by **RequirementDefinitions** (of certain **RequirementTypes**), capabilities by **CapabilityDefinitions** (of certain **CapabilityTypes**), and operations by **Interfaces** and **Operations**.

On the other hand, **Plans** enable the description of application deployment and/or management aspects. Each **Plan** is a workflow that orchestrates the operations offered by the application components (i.e., **NodeTemplates**) to address (part of) the management of the whole cloud application¹.

2.2 (Open) Petri nets

Before providing a formal definition of open Petri nets (Def. 2), we recall the definition of Petri nets just to introduce the employed notation. We instead omit to recall other very basic notions about Petri nets (e.g., marking of a net, firing of transitions, etc.) as they are well-known and easy to find in literature [18].

Definition 1. A Petri net is a tuple $\mathcal{P} = \langle P, T, \bullet, \bullet \rangle$ where P is a set of places, T is a set of transitions (with $P \cap T = \emptyset$), and $\bullet, \bullet : T \rightarrow 2^P$ are functions assigning to each transition its input and output places.

According to [2], an open Petri net is an ordinary Petri net with a distinguished set of (open) places that are intended to represent the interface of the net towards the external environment, meaning that the environment can put or remove tokens from those places. In this paper, we will employ a subset of open Petri nets, where transitions consume at most one token from each place, and where the environment can both add/remove tokens to/from all open places.

Definition 2. An open Petri net is a pair $\mathcal{Z} = \langle \mathcal{P}, I \rangle$, where $\mathcal{P} = \langle P, T, \bullet, \bullet \rangle$ is an ordinary Petri net, and $I \subseteq P$ is the set of open places. The places in $P \setminus I$ will be referred to as internal places.

¹ A more detailed and self-contained introduction to TOSCA can be found in [7].

3 Motivating scenario

Consider a developer who wants to deploy and manage the web services *SendSMS* and *Forex* on a TOSCA-compliant cloud platform. She first describes her services in TOSCA, and then selects the third-party components (i.e. *NodeTypes*) needed to run them. For instance, she indicates that her services will run on a *Tomcat* server installed on an *Ubuntu* operating system, which in turn runs on an *AmazonEC2* virtual machine. Fig. 2 illustrates the resulting *TopologyTemplate*,

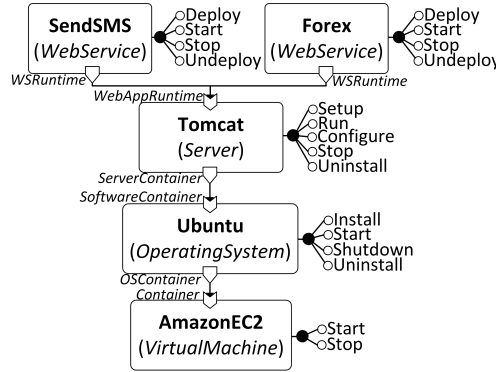


Fig. 2. Motivating scenario.

according to the Winery graphical notation [15]. For the sake of simplicity, and without loss of generality, in the following we focus only on the lifecycle interface [7] of each *NodeType* instantiated in the topology (i.e., the interface containing the operations to install, configure, start, stop, and uninstall a component).

Suppose that the developer wants to describe the automation of the deployment of the *SendSMS* and *Forex* services by writing a TOSCA *Plan*. Since TOSCA does not include any representation of the management protocols of (third-party) *NodeTypes*, developers may produce invalid *Plans*. For instance, while Fig. 3 illustrates three seemingly valid *Plans*, only the third is a valid

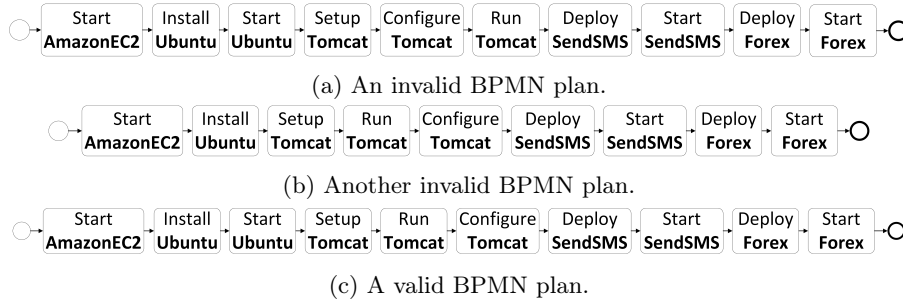


Fig. 3. Deployment Plans.

plan. The other **Plans** cannot be considered valid since (a) *Tomcat's Configure* operation cannot be executed before *Tomcat* is running, and (b) *Tomcat* cannot be installed when the *Ubuntu* operating system is not running.

While the validity of **Plans** can be manually verified, this is a time-consuming and error-prone process. In order to enable the automated verification of the validity of **Plans**, TOSCA should be extended so as to permit specifying the behaviour of and the relations among **NodeTypes**' management operations.

4 Modelling management protocols

While a TOSCA **NodeType** can be described by means of its states, requirements, capabilities, and management operations, there is currently no way to specify how management operations affect states, how operations or states depend on requirements, or which capabilities are concretely provided in a certain state.

The objective of the next section is precisely to propose a way to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities.

4.1 Cloud-based application management protocols in TOSCA

Let N be a TOSCA **NodeType**, and let us denote its states, requirements, capabilities, and management operations with S_N , R_N , C_N , and O_N , respectively.

We want to permit describing whether and how the management operations of N depend on other operations of the same node as well as on operations of the other nodes providing the capabilities that satisfy the requirements of N .

- The first type of dependencies can be easily described by specifying the relationship between states and management operations of N . More precisely, the order with which the operations of N can be executed can be described by means of a transition relation T , that specifies whether an operation o can be executed in a state s , and which state is reached by executing o in s .
- The second type of dependencies can be described by associating transitions and states with (possibly empty) sets of requirements to indicate that the corresponding capabilities are assumed to be provided. More precisely, the requirements associated with a transition t specify which are the capabilities that must be offered by other nodes to allow the execution of t . The requirements associated with a state of a **NodeType** N specify which are the capabilities that must (continue to) be offered by other nodes in order for N to (continue to) work properly.

To complete the description, each state s of a **NodeType** N also specifies the capabilities provided by N in s .

Definition 3. Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a **NodeType**, where S_N , R_N , C_N , and O_N are the sets of its states, requirements, capabilities, and management operations. $\mathcal{M}_N = \langle \bar{s}, R, C, T \rangle$ is the management protocol of N , where

- $\bar{s} \in S_N$ is the initial state,
- R is a function indicating, for each state $s \in S_N$, which conditions on requirements must hold (i.e., $R(s) \subseteq R_N$, with $R(\bar{s}) = \emptyset$)²,
- C is a function indicating which capabilities of N are concretely offered in a state $s \in S_N$ (i.e., $C(s) \subseteq C_N$, with $C(\bar{s}) = \emptyset$), and
- $T \subseteq S_N \times 2^{R_N} \times O_N \times S_N$ is a set of quadruples modelling the transition relation (i.e., $\langle s, H, o, s' \rangle \in T$ means that in state s , and if condition H holds, o is executable and leads to state s').

Syntactically, to describe \mathcal{M}_N we slightly extend the syntax for describing a TOSCA **NodeType**. Namely, we enrich the description of an **InstanceState** by introducing the nested elements **ReliesOn** and **Offers**. **ReliesOn** defines R (of Def. 3) by enabling the association between states and assumed requirements, while **Offers** defines C by indicating the capabilities offered in a state. Furthermore, we introduce the element **ManagementProtocol**, which allows to specify the **InitialState** \bar{s} of the protocol, as well as the **Transitions** defining the transition relation T .

The management protocols of the **NodeTypes** in the motivating scenario of Sect. 3 are shown in Fig. 4, where \mathcal{M}_{WS} is the management protocol for **Web-Services**, \mathcal{M}_S for **Server**, \mathcal{M}_{OS} for **OperatingSystem**, and \mathcal{M}_{VM} for **Virtual-Machine**. Consider for instance the management protocol \mathcal{M}_S of **NodeType**

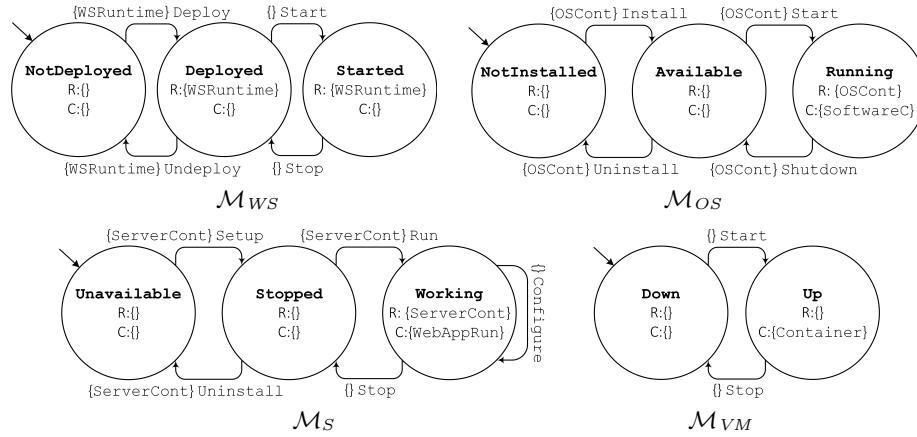


Fig. 4. Management protocols of the **NodeTypes** in our motivating scenario.

Server defining the *Tomcat* server. Its states S_N are **Unavailable** (initial state), **Stopped**, and **Working**, the only requirement in R_N is **ServerContainer**, the only capability in C_N is **WebAppRuntime**, and its management operations are **Setup**, **Uninstall**, **Run**, **Stop**, and **Configure**. States **Unavailable** and **Stopped** are not associated with any requirement or capability. State **Working** instead

² Without loss of generality, we assume that the initial state of a management protocol has no requirements and does not provide any capability.

specifies that the capability corresponding to the **ServerContainer** requirement must be provided (by some other node) in order for **Server** to (continue to) work properly. State **Working** also specifies that **Server** provides the **WebAppRuntime** capability when in such state. Finally, all transitions (but those involving operations **Stop** and **Configure**) constrain their firability by requiring the capability that satisfies **ServerContainer** to be offered (by some other node).

Note that Def. 3 permits to define operations that have non-deterministic effects when applied in a state (e.g., a state can have two outgoing transitions corresponding to the same operation and leading to different states). This form of non-determinism is not acceptable in the management of a TOSCA application [7]. We will thus focus on *deterministic* management protocols, i.e. protocols ensuring deterministic effects when performing an operation in a state.

Definition 4. Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a **NodeType**. The management protocol $\mathcal{M}_N = \langle \bar{s}, R, C, T \rangle$ is deterministic if and only if

$$\forall \langle s_1, H_1, o_1, s'_1 \rangle, \langle s_2, H_2, o_2, s'_2 \rangle \in T: s_1 = s_2 \wedge o_1 = o_2 \Rightarrow s'_1 = s'_2$$

4.2 Modelling cloud-based application management protocols in (open) Petri nets

A (deterministic) management protocol \mathcal{M}_N of a **NodeType** N can be easily encoded by an open Petri net. Each state of \mathcal{M}_N is mapped into an internal place of the Petri net, and each capability and requirement of N is mapped into an open place of the same net. Furthermore, each transition $\langle s, H, o, s' \rangle$ of \mathcal{M}_N is mapped into a Petri net transition t with the following inputs and outputs:

- (i) The input places of t are the places denoting s , the requirements that are needed but not already available in s (i.e., $(R(s') \cup H) - R(s)$), and the capabilities that are provided in s but not in s' (i.e., $C(s) - C(s')$).
- (ii) The output places of t are the places denoting s' , the requirements that were needed but are no more assumed to hold in s' (i.e., $(R(s) \cup H) - R(s')$), and the capabilities that are provided in s' but not in s (i.e., $C(s') - C(s)$).

The initial marking of the obtained net prescribes that the only place initially containing a token is that corresponding to the initial state \bar{s} of \mathcal{M}_N .

Definition 5. Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a **NodeType**, with $\mathcal{M}_N = \langle \bar{s}, R, C, T \rangle$. The management protocol \mathcal{M}_N is encoded into an open net $\mathcal{Z}_N = \langle \mathcal{P}_N, I_N \rangle$, with $\mathcal{P}_N = \langle P_N, T_N, \bullet, \bullet \rangle$ and $I_N \subseteq P_N$, as follows.

- The set P_N of places contains a separate place for each state in S_N , for each requirement in R_N , and for each capability in C_N .
- The set $I_N \subset P_N$ of open places contains the places denoting the requirements in R_N and the capabilities in C_N .
- The set T_N contains a net transition t for each transition $\langle s, H, o, s' \rangle \in T$.
 - (i) The set $\bullet t$ of input places contains the place s , the places denoting the requirements in $(R(s') \cup H) - R(s)$, and those denoting the capabilities in $C(s) - C(s')$.

- (ii) The set $t \bullet$ of output places contains the place s' , the places denoting the requirements in $(R(s) \cup H) - R(s')$, and those denoting the capabilities in $C(s') - C(s)$.

The initial marking of \mathcal{Z}_N consists of only one token in place \bar{s} .

The above definition ensures that the Petri net encoding of a management protocol satisfies the following properties:

- There is a one-to-one correspondence between the marking of the internal places of the Petri net and the states of a management protocol. Namely, there is exactly one token in the internal place denoting the current state, and no tokens in the other internal places.
- Each operation can be performed if and only if all the necessary requirements are available in the source state, and no capability required by any connected component is disabled in the target state.

Consider for instance the management protocol \mathcal{M}_S (Fig. 4), whose corresponding Petri net is shown in Fig. 5. Each state in \mathcal{M}_S is translated into an in-

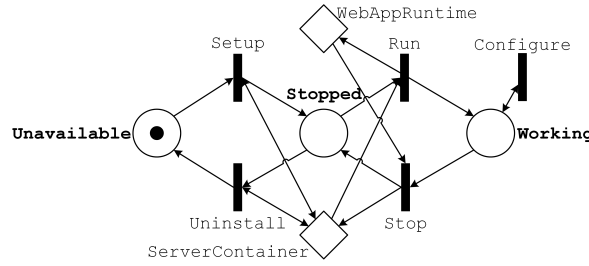


Fig. 5. Example of Petri net translation.

ternal place (represented as a circle), while the **ServerContainer** requirement and the **WebAppRuntime** capability are translated into open places (represented as diamonds). Additionally, protocol transitions are translated into net transitions. For example, the transition $\langle \text{Stopped}, \{\text{ServerContainer}\}, \text{Run}, \text{Working} \rangle$ is translated into a Petri net transition, whose inputs places are **Stopped** and **ServerContainer**, and whose outputs places are **Working** and **WebAppRuntime**.

4.3 Analysis of cloud-based application management protocols

We now show how the Petri net modelling the management protocol of a TOSCA **TopologyTemplate** (specifying a whole cloud-based application) can be obtained, in a compositional way, from the Petri nets modelling the management protocols of the **NodeTypes** in such **TopologyTemplate**.

We first need to model (by open Petri nets working as a *capability controllers*) the **RelationshipTemplates** that define in a **TopologyTemplate** the

association between the requirements of a **NodeType** and the capabilities of other **NodeType**s. To do that, we first define an utility *binding* function that returns the set of requirements with which a capability is associated.

Definition 6. Let S be a **ServiceTemplate**, and let c be a capability offered by a **NodeType** in S . We define $b(c, S) = \{r_1, \dots, r_n\}$, where r_1, \dots, r_n are the requirements connected to c in S by means of **RelationshipTemplates**.

We now exploit function b to define *capability controllers*. On the one hand, the controller must ensure that once a capability c is available, the nodes exposing the connected requirements r_1, \dots, r_n are able to simultaneously exploit it. This is obtained by adding a transition c_\uparrow able to propagate the token from place c to places r_1, \dots, r_n (i.e., the input place of c_\uparrow is c , and its output places are r_1, \dots, r_n). On the other hand, the controller has also to ensure that the capability is not removed while at least another node is actively assuming its availability (with a condition on a connected requirement). Thus, we introduce a transition c_\downarrow whose input places are r_1, \dots, r_n and whose output place is c .

Definition 7. Let S be a **ServiceTemplate**, and let c be a capability offered by a **NodeType** instantiated in S . Let r_1, \dots, r_n be the requirements exposed by the nodes in S such that $b(c, S) = \{r_1, \dots, r_n\}$. The controller of c is an open Petri net $\mathcal{Z}_c = \langle \mathcal{P}_c, \mathcal{I}_c \rangle$, with $\mathcal{P}_c = \langle P_c, T_c, \bullet, \bullet \rangle$, defined as follows.

- The set P_c of places contains a separate place for the capability c and for each requirement r_1, \dots, r_n .
- The set \mathcal{I}_c coincides with P_c .
- The set T_c contains only two Petri net transitions c_\uparrow and c_\downarrow .
 - The input and output places of c_\uparrow are the place c and the places r_1, \dots, r_n , respectively (i.e., $\bullet c_\uparrow = \{c\}$ and $c_\uparrow \bullet = \{r_1, \dots, r_n\}$).
 - The input and output places of c_\downarrow are the places r_1, \dots, r_n and the place c , respectively (i.e., $\bullet c_\downarrow = \{r_1, \dots, r_n\}$ and $c_\downarrow \bullet = \{c\}$).

The initial marking of \mathcal{Z}_c is empty (i.e., no place contains a token).

An example of controller (for a capability c connected to two requirements r_1 and r_2) is illustrated in Fig. 6.

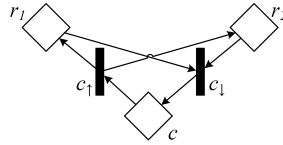


Fig. 6. Example of *capability controller*.

We can now compose the nets modelling the management protocols of the **NodeTypes** instantiated in a **ServiceTemplate**'s topology by interconnecting them with the above introduced controllers. The composition is quite simple: We just collapse the open places corresponding to the same requirements/capabilities.

Definition 8. Let S be a **ServiceTemplate**. We encode S with an open Petri net $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, where $\mathcal{P}_S = \langle P_S, T_S, \bullet, \cdot, \bullet \rangle$, as follows.

- For each node N in the topology of S , we encode its management protocol with an open Petri net \mathcal{Z}_N obtained as shown in Def. 5.
- For each capability c exposed by a **NodeTemplate** in S , we create an open Petri net \mathcal{Z}_c (acting as its controller) as shown in Def. 7.
- We then compose the above mentioned nets by taking their disjoint union and merging the places denoting the same requirement r or capability c .

The initial marking of \mathcal{Z}_S is the union of the markings of the collapsed nets.

For example, Fig. 7 shows the net obtained for the motivating scenario in Sect. 3.

The Petri net encoding of the management of a **ServiceTemplate** S , permits us defining what is a *valid plan* according to such management. Essentially, thanks to the encoding of capability controllers and to the way we compose these controllers with management protocol encodings, the obtained net ensures that no requirement can be assumed to hold if the corresponding capability is not provided, and that no capability can be removed if at least one of the corresponding requirements is assumed to hold. This permits to consider a plan valid if and only if it corresponds to a firing sequence in the net encoding of S .

Definition 9. Let S be a **ServiceTemplate** and let $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, with $\mathcal{P}_S = \langle P_S, T_S, \bullet, \cdot, \bullet \rangle$, be the Petri net encoding of S . A sequential plan³ $o_1 o_2 \dots o_m$ is valid if and only if there is a firing sequence $t_1 t_2 \dots t_n$ in \mathcal{Z}_S from the initial marking such that $o_1 \cdot o_2 \cdot \dots \cdot o_m = \lambda(t_1) \cdot \lambda(t_2) \cdot \dots \cdot \lambda(t_n)$, where \cdot indicates the concatenation operator⁴ and:

$$\lambda(t) = \begin{cases} \epsilon & \text{if } t \text{ denotes a } c_{\uparrow} \text{ or } c_{\downarrow} \text{ transition} \\ o & \text{if } t \text{ denotes a management protocol transition } \langle s, H, o, s' \rangle \end{cases}$$

It is easy to see now that plan (c) of Fig. 3 is valid since, for instance,

*AmazonEC2:Start Container*_↑ *Ubuntu:Install Ubuntu:Start SoftwareContainer*_↑
*Tomcat:Setup Tomcat:Run Tomcat:Configure WebAppRuntime*_↑ *SendSMS:Deploy*
SendSMS:Start Forex:Deploy Forex:Start

is a corresponding firing sequence for the Petri net in Fig. 7. Conversely, plans (a) and (b) in Fig. 3 are not valid as there are no corresponding firing sequences. Intuitively speaking, (a) is not valid since after firing, for instance,

*AmazonEC2:Start Container*_↑ *Ubuntu:Install Ubuntu:Start SoftwareContainer*_↑
Tomcat:Setup

³ In Def. 9 we consider sequential plans. A workflow plan is valid if and only if all its sequential traces are valid.

⁴ The empty string ϵ is the neutral element of \cdot , hence controllers' net transitions are ignored (as $\lambda(t) = \epsilon$ when t denotes a c_{\uparrow} or c_{\downarrow} transition).

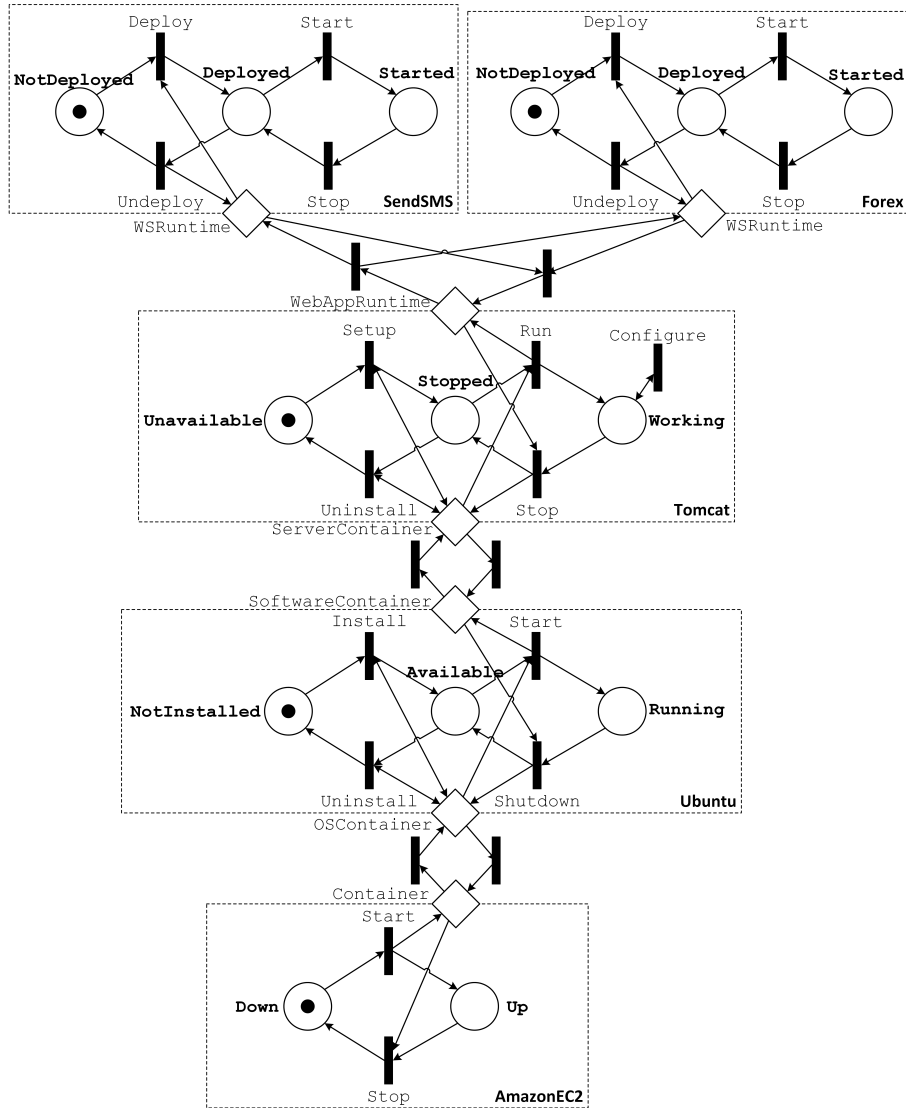


Fig. 7. Petri net encoding for the motivating scenario in Sect. 3.

transition *Tomcat:Configure* cannot be fired. It indeed requires a token in the *Working* place, but that place is empty and it is not possible to add tokens to it without firing *Tomcat:Run*. On the other hand, (b) is not valid since after firing

*AmazonEC2:Start Container*_↑ *Ubuntu:Install*

transition *Tomcat:Setup* cannot fire. It requires a token in the place denoting the *ServerContainer* requirement, but that place is empty and it is not possible to add tokens to it without firing *SoftwareContainer*_↑, which in turn cannot fire as it misses a token in the place denoting the *Ubuntu's SoftwareContainer* capability (and no token can be added to such place without firing *Ubuntu:Start*).

It is important to observe that the correspondence between firing sequences and valid plans can be exploited for many other purposes besides checking plans' validity. The effects of a plan on the states of the components of a TOSCA *ServiceTemplate*, as well as on the requirements that are satisfied and the capabilities that are available, can be directly determined from the marking that is reached performing the corresponding firing sequence. Additionally, various classical notions in the Petri net context assume a specific meaning in the context of TOSCA applications. For example the problem of finding whether there is a plan which achieves a specific goal (e.g., bringing some components of an application to specific states or making some capabilities available) can be reduced in a straightforward way to the coverability problem [18] on the associated Petri net. Moreover, it is possible to consider as initial marking any other (reachable) marking so as to analyse maintenance plans (starting from non-initial states) besides deployment plans. Obviously, the very same properties and techniques also apply in this case. Finally, the Petri net is *reversible* [18] if and only if it is always possible to (softly) reset the application. This is a very convenient property, because it guarantees that it is always possible to generate a plan for any reachable goal from any application state.

5 Related work

Automating application management is a well-known problem in computer science. With the advent of cloud computing, it has become even more prominent because of the complexity of both applications and platforms [8]. This is witnessed by the proliferation of so-called *configuration management systems*, like Chef [9] or Puppet [21]. These systems provide a domain-specific language to model the desired configuration for a machine and employ a client-server model in which a server holds the model and the client ensures this configuration is met. However, the lack of a machine readable representation of management protocols of application components inhibits the possibility of automating verification on components' configurations and dependencies.

A large body of research has been devoted to model interacting systems by means of finite state machines, Petri nets, and other formal models [4,11]. Because of space limitations, we discuss next only the work more closely related to ours, tailored to model the behaviour of cloud application management.

A first attempt to master the complexity of the cloud is given by the Aeolus component model [10]. The Aeolus model is specifically designed to describe several characteristics of cloud application components (e.g., dependencies, non-functional requirements, etc.), as well as the fact that component interfaces might vary depending on the internal component state. However, the model only allows to specify what is offered and required in a state. Our approach instead allows developers to clearly separate the requirements ensuring the consistency of a state from those constraining the applicability of a management operation. This allows developers to easily express transitions where requirements are affecting only the applicability of an operation and not the consistency of a state (e.g., the transition $\langle \text{Unavailable}, \{\text{ServerContainer}\}, \text{Setup}, \text{Stopped} \rangle$ of the management protocol \mathcal{M}_S in Fig. 4). Such a kind of transitions cannot be easily modelled in Aeolus. Furthermore, Aeolus and other emerging solutions like Juju [13] and Engage [12], differ from our approach since they are geared towards the deployment of cloud applications, thus not including also their maintenance. Additionally, Aeolus, Juju, and Engage are currently not integrated with any cloud interoperability standard, thus limiting their applicability to only some supported cloud platforms. Our approach, instead, intends to model the entire lifecycle of a cloud application component, and achieves cloud interoperability by relying on the TOSCA standard [19].

To this end, TOSCA offers a rich type system permitting to match, adapt and reuse existing solutions [7]. Since our proposal extends this type system, it can also be exploited to refine existing reuse techniques, like [5,6,22]. Currently, these techniques are matchmaking and adapting (fragments of) existing **Service-Templates** to implement a desired **NodeType** by checking whether the features of the latter are all offered by the former. To overcome syntactic differences, ontologies may be employed to check whether two different names are denoting the same concept. However, these techniques are behaviour-unaware: There is no way to determine whether the behaviour of the identified (fragment of) **Service-Template** is coherent with that of the desired **NodeType**. Since our approach permits describing the behaviour of management operations, it can be exploited to extend the aforementioned techniques to become behaviour-aware.

It is also worth highlighting that we could directly compose the finite state machines specifying management protocols, and model valid plans as the language accepted by the composite finite state machine. However, the size of the latter grows exponentially with the number of application components. This results in a high computational complexity, even if we exploit composition-oriented automata (e.g., *interface automata* [1]). On the other hand, with open Petri nets [2,14], we have a very simple composition approach, and the exponential growth only affects the amount of reachable markings (instead of the size of the net). A simpler composition approach is even more convenient since cloud applications can change over time. For instance, to add another web service to our motivating scenario, our approach just requires to add the open Petri net encoding its management protocol, and to connect the open places denoting its requirement with the corresponding c_{\uparrow} and c_{\downarrow} transitions. On the other hand, with an au-

tomata based approach, the composition would be much harder, as it requires to compute the Cartesian product of the automata's states.

6 Conclusions

In this paper we have proposed an extension of TOSCA that permits to specify the behaviour of management operations of cloud-based applications, and their relations with states, requirements, and capabilities. We have then shown how the management protocols of TOSCA components can be naturally modelled, in a compositional way, by means of open Petri nets, and that such modelling permits to automate different analyses, such as determining whether a plan is valid, which are its effects, or which plans allow to reach certain system configurations.

Please note that, while some of those Petri-net analyses have an exponential time complexity in the worst case, they still constitute a significant improvement with respect to the state of the art, where the validity of deployment plans can be verified only manually, after delving through the documentation of application components. Please also note that our approach builds on top of, but is not limited to, TOSCA. It can be easily adapted to other stateful behaviour models of systems that describe states, requirements, capabilities, and operations.

We see different possible extensions of our work. We are currently working on a prototype implementation of our approach, which includes a graphical user interface to support the definition of valid TOSCA specifications that include management protocols. The graphical user interface will compile the management protocols of a TOSCA application into a PNML file [3], hence enabling to plug-in different PNML processing environments (e.g., LoLa, ProM, or WoPeD, just to mention some) to implement the analyses described in Sect. 4.3. We also intend to improve the efficiency of the analyses by reducing the complexity of the nets that is due to the c_{\uparrow} and c_{\downarrow} transitions introduced by the controllers. Indeed the net encoding of a cloud application can be simplified by “folding” the controllers’ transitions (by modifying the transitions whose input/output places contain a place representing capability c so that they are replaced by the places representing the requirements connected to c by means of **Relationship-Templates**). Another interesting direction for future work is to investigate the applicability of more sophisticated fault diagnosis analyses (like [16,17]) to identify the reasons why a plan may not be valid (besides just showing the points in which a plan may get stuck, as we currently do). Finally, we want to extend the matchmaking and adaptation techniques we previously proposed [5,6,22] by including the behaviour information coming from management protocols (as illustrated in Sect. 5).

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the 8th European Software Engineering Conference / 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 109–120. ESEC/FSE-9, ACM (2001)

2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science* 15(01), 1–35 (2005)
3. Billington, J., Christensen, S., Van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, technology, and tools. In: *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*. pp. 483–505. ICATPN’03, Springer (2003)
4. Bochmann, G.V., Sunshine, C.A.: A survey of formal methods. In: *Computer Network Architectures and Protocols*, pp. 561–578. *Applications of Communications Theory*, Springer (1982)
5. Brogi, A., Soldani, J.: Matching cloud services with TOSCA. In: *Advances in Service-Oriented and Cloud Computing, CCIS*, vol. 393, pp. 218–232. Springer (2013)
6. Brogi, A., Soldani, J.: Reusing cloud-based services with TOSCA. In: *INFORMATIK 2014. LNI*, vol. 232, pp. 235–246. Gesellschaft für Informatik (GI) (2014)
7. Brogi, A., Soldani, J., Wang, P.: TOSCA in a Nutshell: Promises and Perspectives. In: Villari, M., Zimmermann, W., Lau, K.K. (eds.) *Service-Oriented and Cloud Computing*. LNCS, vol. 8745, pp. 171–186. Springer (2014)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25(6), 599 – 616 (2009)
9. Chef: Opscode. <https://www.opscode.com/chef>
10. Di Cosmo, R., Mauro, J., Zacchioli, S., Zavattaro, G.: Aeolus: A component model for the cloud. *Information and Computation* 239(0), 100 – 121 (2014)
11. Diaz, M.: Modeling and analysis of communication and cooperation protocols using Petri net based models. *Computer Networks* 6(6), 419 – 441 (1982)
12. Fischer, J., Majumdar, R., Esmailsabzali, S.: Engage: A deployment management system. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 263–274. PLDI ’12, ACM (2012)
13. Juju: DevOps distilled. <https://juju.ubuntu.com>
14. Kindler, E.: A compositional partial order semantics for Petri net components. In: *Proceedings of ICATPN ’97*. pp. 235–252. Springer-Verlag (1997)
15. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – Modeling Tool for TOSCA-based Cloud Applications. In: *Proceedings of the 11th International Conference on Service-Oriented Computing*. Springer (2013)
16. Lohmann, N.: Why does my service have no partners? In: Bruni, R., Wolf, K. (eds.) *Web Services and Formal Methods*, LNCS, vol. 5387, pp. 191–206. Springer (2009)
17. Lohmann, N., Fahland, D.: Where did I go wrong? In: *Business Process Management*, LNCS, vol. 8659, pp. 283–300. Springer (2014)
18. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* 77(4), 541–580 (1989)
19. OASIS: Topology and Orchestration Specification for Cloud Applications. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf> (2013)
20. OASIS: TOSCA Simple Profile in YAML. <http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf> (2014)
21. Puppet: Puppet labs. <https://puppetlabs.com>
22. Soldani, J., Binz, T., Breitenbücher, U., Leymann, F., Brogi, A.: TOSCA-MART: A method for adapting and reusing cloud applications. Tech. Rep., University of Pisa (March 2015), http://eprints.adm.unipi.it/2331/1/SBBLB15_-_TR.pdf

Unfolding CSPT-nets

Bowen Li and Maciej Koutny

School of Computing Science, Newcastle University
Newcastle upon Tyne NE1 7RU, United Kingdom
`{bowen.li,maciej.koutny}@ncl.ac.uk`

Abstract. Communication structured occurrence nets (CSONs) are the basic variant of structured occurrence nets which have been introduced to characterise the behaviours of complex evolving systems. A CSON has the capability of portraying different types of interaction between systems by using special elements to link with multiple (component) occurrence nets. Communication structured place transition nets (CSPT-nets) are the system-level counterpart of CSONs. In this paper, we investigate CSPT-nets unfoldings containing representations of all the single runs of the original nets captured by CSONs. We develop several useful notions related to CSPT-net unfoldings, and then present an algorithm for constructing the new class of unfolding.

Keywords: structured occurrence nets, place transition nets, CSPT-nets, unfolding, synchronous and asynchronous communication

1 Introduction

A complex evolving system consists of a large number of sub-systems which may proceed concurrently and interact with each other or with the external environment while its behaviour is subject to modification by other systems. The communication between sub-systems may either be asynchronous or synchronous. Structured occurrence nets (SONs) [8, 13, 14] are a Petri net based formalism that can be used to model the behaviours of complex evolving system. The concept extends that of occurrence nets [1] which are directed acyclic graphs that represent causality and concurrency information concerning a single execution of a system. In SON, multiple related occurrence nets are combined by means of various formal relationships; in particular, in order to express dependencies between interacting systems. Communication structure occurrence nets (CSONs) are the basic variant of SONs. The model has the capability of portraying different types of interaction between systems. A CSON involves occurrence nets that are connected by channel places representing synchronous or asynchronous communications. [7] introduced a system-level counterpart of CSONs called communication structured place transition nets (CSPT-nets). The nets are built out of the place/transition nets (PT-nets), which are connected by channel places allowing both synchronous and asynchronous communication.

The standard Petri nets unfoldings, introduced in [2, 12], are a technique supporting effective verification of concurrent systems modeled by Petri nets

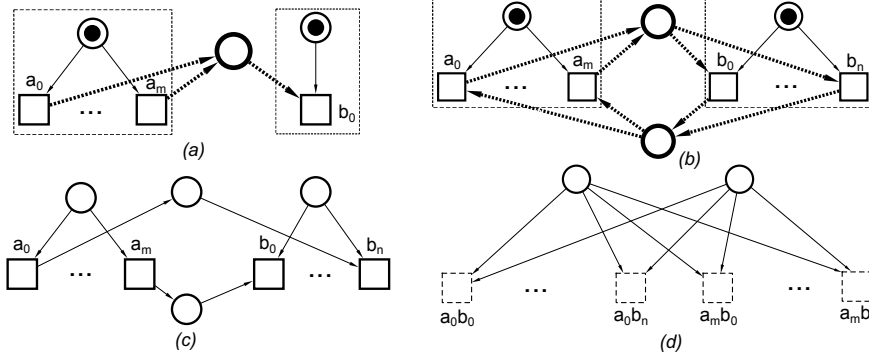


Fig. 1. Two CSPT-nets (a) and (b); together with their respective standard unfoldings semantics after applying the Petri net encodings (c) and (d).

(throughout this paper, Petri net related concepts, such as configuration, unfolding, merged process, will be referred to as *standard*). The method relies on the concept of net unfolding which can be seen as the partial order behaviour of a concurrent system. The unfolding (or branching process) of a net is usually infinite, but for bounded Petri nets one can construct a finite complete prefix of the unfolding containing enough information to analyse important behavioural properties. [9] investigated branching processes of CSPT-nets (CSPT-net unfoldings). As in the standard net theory, CSPT branching processes act as a ‘bridge’ between CSPT-nets and their processes captured by CSONs (i.e., the branching processes of a CSPT-net contains a representation of all the possible single runs of the original net). In order to reduce the complexity of branching processes of CSPT-nets, we adapt the notion of occurrence depth which was originally developed for merged processes [5].

In this paper, we introduce and discuss several key properties of branching processes of CSPT-nets. We also present an algorithm for constructing CSPT-net unfoldings, generalising the unfolding algorithm introduced in [9] which could only handle channel places with a single input and a single output transition. In particular, the new algorithm takes into account the occurrence depth of events, and fuses nodes which have same behaviours during the unfolding. In this way, the size of the resulting net can be significantly reduced when compared with the standard unfolding approach.

Consider the CSPT-nets shown in Figure 1(a) and (b). In (a), m transitions asynchronously communicate with b_0 via a single channel place. In (b), m transitions are synchronous with n transitions between two PT-nets via two channel places. Their unfolding semantics are isomorphic to the original CSPT-nets (with the sizes of $m + 1$ events in (a) and $m + n$ events in (b)). If one was only interested in marking reachability, then one might attempt to encode a CSPT-net by replacing every asynchronous channel place by a standard place and ‘glue’ transitions forming a synchronous event into a single one. One would then be able to apply the standard unfolding to this Petri net based representation. However,

the efficiency of such an approach would suffer from the introduction of exponentially many new transitions, as well as the loss of the merging on channel places which is due to the exploitation of occurrence depth. In this case, the ‘replace’ encoding for (a) yields $n + m$ events in the corresponding unfolding (c). While the ‘glue’ encoding for (b) would yield $m \times n$ events as shown in (d).

The paper is organised as follows. Section 2 provides basic notions concerning Petri nets and their unfoldings. Section 3 presents the main concepts of communication structured net theory, including CSON-nets, CSPT-nets and CSPT branching processes. In section 4, we discuss finite complete prefixes of CSPT branching processes and related properties. The CSPT unfolding algorithm is provided in Section 5. Section 6 discusses future works and concludes the paper. The technical report [10] contains proofs of formal results and an example of the algorithm run.

2 Basic Definitions

We assume that the reader is familiar with the basic notions concerning Petri nets and their unfoldings, which can be found in, e.g., [1, 2, 12]. Throughout the paper, a *multiset* over a set X is a function $\mu : X \rightarrow \mathbb{N}$, where $\mathbb{N} = \{0, 1, 2, \dots\}$. A multiset may be represented by explicitly listing its elements with repetitions. For example $\{a, a, b\}$ denotes the multiset such that $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(x) = 0$ for $x \in X \setminus \{a, b\}$.

PT-nets. A *net* is a triple $N = (P, T, F)$ such that P and T are disjoint sets of respectively *places* and *transitions* (collectively referred to as *nodes*), and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow* relation. The *inputs* and *outputs* of a node x are defined as $\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$. Moreover, $\bullet x^\bullet = \bullet x \cup x^\bullet$. It is assumed that the inputs and outputs of a transition are nonempty sets. Two nodes, x and x' , are in *conflict* if there are distinct transitions, t and t' , such that $\bullet t \cap \bullet t' \neq \emptyset$ and $(t, x) \in F^*$ and $(t', x') \in F^*$. We denote this by $x \# x'$. A node x is in *self-conflict* if $x \# x$.

A *place transition net* (PT-net) is a tuple $PT = (P, T, F, M_0)$, where (P, T, F) is a finite net, and $M_0 : P \rightarrow \mathbb{N}$ is the *initial marking* (in general, a marking is a multiset of places). A *step* U is a non-empty multiset of transitions of PT . It is *enabled* at a marking M if $M(p) \geq \sum_{t \in \bullet p} U(t)$, for every place p . In such a case, the *execution* of U leads to a new marking M' given, for every $p \in P$, by $M'(p) = M(p) + \sum_{t \in \bullet p} U(t) - \sum_{t \in p^\bullet} U(t)$. We denote this by $M[U]M'$. A *step sequence* of PT is a sequence $\lambda = U_1 \dots U_n$ ($n \geq 0$) of steps such that there exist markings M_1, \dots, M_n satisfying $M_0[U_1]M_1, \dots, M_{n-1}[U_n]M_n$. The *reachable markings* of PT are defined as the smallest (w.r.t. \subseteq) set $\text{reach}(PT)$ containing M_0 and such that if there is a marking $M \in \text{reach}(PT)$ and $M[U]M'$, for a step U and a marking M' , then $M' \in \text{reach}(PT)$. PT is *k-bounded* if, for every reachable marking M and every place $p \in P$, $M \leq k$, and *safe* if it is 1-bounded. The markings of a safe PT-net can be treated as sets of places.

Branching processes of PT-nets. A net $ON = (P, T, F)$, with places and transitions called respectively *conditions* and *events*, is a *branching occurrence*

net if the following hold: (i) F is acyclic and no transition $t \in T$ is in self-conflict; (ii) $|\bullet p| \leq 1$, for all $p \in P$; and (iii) for every node x , there are finitely many y such that $(y, x) \in F^*$. The set of all places p with no inputs (i.e., $\bullet p = \emptyset$) is the default initial state of ON , denoted by M_{ON} . In general, a *state* is any set of places. If $|p^\bullet| \leq 1$, for all $p \in P$, then ON is a *non-branching* occurrence net. Note that in a branching occurrence net, two paths outgoing from a place will never meet again by coming to the same place (the inputs of places are at most singleton sets) nor the same transition (transitions cannot be in self-conflict).

A *branching process* of a PT-net $PT = (P, T, F, M_0)$ is a pair $\Pi = (ON, h)$, where $ON = (P', T', F')$ is a branching occurrence net and $h : P' \cup T' \rightarrow P \cup T$ is a mapping, such that the following hold: (i) $h(P') \subseteq P$ and $h(T') \subseteq T$; (ii) for every $e \in T'$, the restriction of h to $\bullet e$ is a bijection between $\bullet e$ and $\bullet h(e)$, and similarly for e^\bullet and $h(e)^\bullet$; (iii) the restriction of h to M_{ON} is a bijection between M_{ON} and M_0 ; and (iv) for all $e, f \in T'$, if $\bullet e = \bullet f$ and $h(e) = h(f)$ then $e = f$. There exists a maximal branching process Π_{PT} , called the *unfolding* of PT [2].

Configurations and cuts of a branching process. Let $\Pi = (ON, h)$ be a branching process of a PT-net PT , and $ON = (P', T', F')$. A *configuration* of Π is a set of events $C \subseteq T'$ such that $\neg(e \# e')$, for all $e, e' \in C$, and $(e', e) \in F'^+ \implies e' \in C$, for every $e \in C$. In particular, the *local configuration* of an event e , denoted by $[e]$, is the set of all the events e' such that $(e', e) \in F'^*$. The notion of a configuration captures the idea of a possible history of a concurrent system, where all events must be conflict-free, and all the predecessors of a given event must have occurred. A *co-set* of Π is a set of conditions $B \subseteq P'$ such that, for all distinct $b, b' \in B$, $(b, b') \notin F'^+$. Moreover, a *cut* of Π is any maximal (w.r.t. \subseteq) co-set B . Finite configurations and cuts of Π are closely related (every marking represented in the unfolding Π_{PT} is reachable in PT , and every reachable marking of PT is represented in Π_{PT}):

- if C is a finite configuration of Π , then $Cut(C) = (M_{ON} \cup C^\bullet) \setminus \bullet C$ is a cut and $Mark(C) = h(Cut(C))$ is a reachable marking of PT ; and
- if M is a reachable marking of PT , then there is a finite configuration C of Π_{PT} such that $Mark(C) = M$.

3 Structuring PT-nets

In this section we recall the formal definitions concerning communication structured nets theory, including CSON-nets and CSPT-nets. We then introduce the notion of branching processes of CSPT-nets and several related properties.

The new models are able to portray different kinds of communication between separate systems. One can envisage that if a given PT-net attempts to represent several interacting systems, it will be beneficial to split the model into a set of component nets, and create specific devices to represent any communication between the subsystems. In the model we are interested in communication can be synchronous or asynchronous. Usually, the former implies that a sender waits for an acknowledgment of a message before proceeding, while in the latter the sender proceeds without waiting.

A communication structured net is composed of a set of component nets representing separate subsystems. When it is determined that there is a potential for an interaction between subsystems, asynchronous or synchronous communication link can be made between transitions (or events) in the different nets via a special element called a *channel place*. Two transitions (events) involved in a synchronous communication link must be executed simultaneously. On the other hand, transitions (events) involved in an asynchronous communication may be executed simultaneously, or one after the other.

Similarly as in the case of PT-nets, non-branching processes CSN-nets will represent single runs of CSPT-nets, while branching processes will capture full execution information of the corresponding CSPT-nets.

CSPT-nets. By generalising the definition of [7], we first introduce an extension of PT-nets which combines several such nets into one model using channel places.

Definition 1 (CSPT-net). A communication structured place transition net (or CSPT-net) is a tuple $CSPT = (PT_1, \dots, PT_k, Q, W, M_0)$ ($k \geq 1$) such that each $PT_i = (P_i, T_i, F_i, M_i)$ is a safe (component) PT-net; Q is a finite set of channel places; $M_0 : Q \rightarrow \mathbb{N}$ is the initial marking of the channel places; and $W \subseteq (T \times Q) \cup (Q \times T)$, where $T = \bigcup T_i$, is the flow relation for the channel places. It is assumed that the following are satisfied:

1. The PT_i 's and Q are pairwise disjoint.
2. For every channel place $q \in Q$,
 - the sets of inputs and outputs of q , $\bullet q = \{t \in T \mid (t, q) \in W\}$ and $q^\bullet = \{t \in T \mid (q, t) \in W\}$ are both nonempty and, for some $i \neq j$, $\bullet q \subseteq T_i$ and $q^\bullet \subseteq T_j$; and
 - if $\bullet q \cap T_i \neq \emptyset$ then there is no reachable marking of PT_i which enables a step comprising two distinct transitions in $\bullet q^\bullet$. \diamond

The initial marking M_{CSPT} of $CSPT$ is the multiset sum of the M_i 's ($i = 0, 1, \dots, k$), and a marking is in general a multiset of places, including the channel places.

To simplify the presentation, in the rest of this paper we will assume that the channel places in the initial states of CSPT-nets are empty.

The execution semantics of $CSPT$ is defined as for a PT-net except that a step of transitions U is *enabled* at a marking M if, for every non-channel place p , $M(p) \geq \sum_{t \in p^\bullet} U(t)$ and, for every channel place q ,

$$M(q) + \sum_{t \in \bullet q} U(t) \geq \sum_{t \in q^\bullet} U(t) . \quad (*)$$

The condition (*) for step enabledness caters for synchronous behaviour as step U can use not only the tokens that are already available in channel places at marking M , but also can use the tokens deposited there by transitions from U during the execution of U . In this way, transitions from U can ‘help’ each other

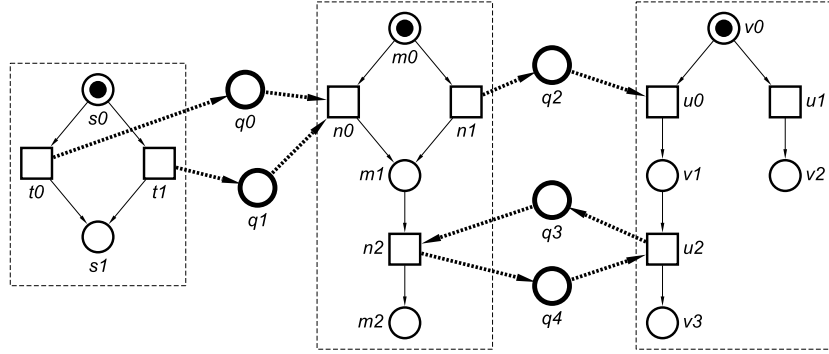


Fig. 2. A CSPT-net with three component PT-nets.

individually and synchronously pass resources (tokens) among themselves. Thus, in contrast to the step execution of a PT-net where a step consists of a number of enabled transitions, the execution of a step in a CSPT-net (i.e., $M[U]M'$) may involve synchronous communications (or interactions), where transitions execute simultaneously and behave as a transaction. Such a mode of execution is strictly more expressive than that used in PT-nets.

Figure 2 shows a CSPT-net which consists of three component PT-nets connected by a set of channel places (represented by circles with thick edges). To improve readability, the thick dashed lines indicate the flow relation W . Transitions n_2 and u_2 are connected by a pair of empty channel places, q_3 and q_4 , forming a cycle. This indicates that these two transitions can only be executed synchronously. They will be filled and emptied synchronously when both n_2 and u_2 participate in an enabled step. On the other hand, the execution of transitions n_1 and u_0 can be either asynchronous (n_1 occurs before u_0), or synchronous (both of them occur simultaneously). A possible step sequence of Figure 2 is $\lambda = \{t_0, n_1\}\{u_0\}\{n_2, u_2\}$, where n_1 and u_0 perform an asynchronous communication. Another step sequence $\lambda' = \{t_0\}\{n_1, u_0\}\{n_2, u_2\}$ shows that n_1 and u_0 can be also executed synchronously.

Definition 1(2) means that the occurrences of transitions in $\bullet q$ (as well as those in q^\bullet) are totally ordered in any execution of the corresponding component net PT_i . In other words, we assume that both the output access and the input access to the channel places is *sequential*. This will allow us to introduce the notion of depth at which an event which accessed a channel place has occurred.

Given a branching process derived for a component PT-net of a CSPT-net, consider an event e such that its corresponding transition is an input (or output) of a channel place q in the CSPT-net. Then the occurrence depth of such event w.r.t., the channel place q is the number of events such that they all causally precede e and their corresponding transitions are also inputs (or outputs) of the channel place q . Since the tokens flowing through channel places are based on the

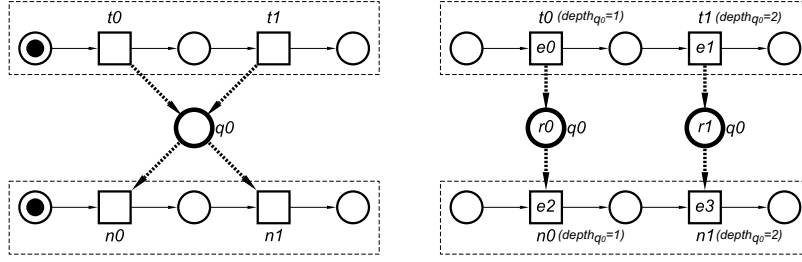


Fig. 3. (a) A CSPT-net, and (b) its branching process (event labels are shown alongside the nodes and the occurrence depths are shown in brackets).

FIFO policy. The occurrence depth intuitively represents the number of tokens which have entered (or left) the channel place q before the occurrence of e .

Definition 2 (occurrence depth). Let CSPT be as in Definition 1, and $q \in Q$ and PT_i be such that $\bullet q \cap T_i \neq \emptyset$. Moreover, let $\Pi = (ON, h)$ be a branching process of PT_i , and e be an event of $ON = (P', T', F')$ such that $h(e) \in \bullet q$. The depth of e in Π w.r.t. the channel place q is given by:

$$\text{depth}_q^\Pi(e) = |\{f \in T' \mid h(f) \in \bullet q \wedge (f, e) \in F'^*\}|.$$

Moreover, if the process Π is clear from the context, we will write $\text{depth}_q(e)$ instead of $\text{depth}_q^\Pi(e)$. \diamond

Proposition 1. Let Π and $q \in Q$ be as in Definition 2. Moreover, let e and f be two distinct events of Π satisfying $\neg(e \# f)$ and $h(e), h(f) \in \bullet q$. Then $\text{depth}_q(e) \neq \text{depth}_q(f)$.

The nets in the dashed line boxes in Figure 3(b) are two component branching processes derived from the component PT-nets of the CSPT-net in Figure 3(a). The labels are shown alongside each node, and the occurrence depth of each event connected to a (unique, in this case) channel place is shown in brackets. Let us consider event e_1 . Its corresponding transition t_1 is the input of channel place q_0 . When searching the directed path starting at the initial state and terminating at e_1 , we can find another event (viz. e_0) such that its corresponding transition is also the input of q_0 . Therefore the occurrence depth of e_1 , w.r.t. q_0 , is $\text{depth}_{q_0}(e_1) = 2$. It intuitively represents transition t_1 passing the second token to the channel.

Non-branching processes of CSPT-nets. Similarly to the way in which CSPT-nets are extensions of PT-nets, non-branching processes of CSPT-nets are extensions of non-branching occurrence nets.

Definition 3 (non-branching process of CSPT-net). Let CSPT be as in Definition 1 with M_0 being empty. A non-branching process of CSPT is a tuple $\text{CSON} = (\Pi_1, \dots, \Pi_k, Q', W', h')$ such that each $\Pi_i = (ON_i, h_i)$ is a non-branching process of PT_i with $ON_i = (P'_i, T'_i, F'_i)$; Q' is a set of channel places;

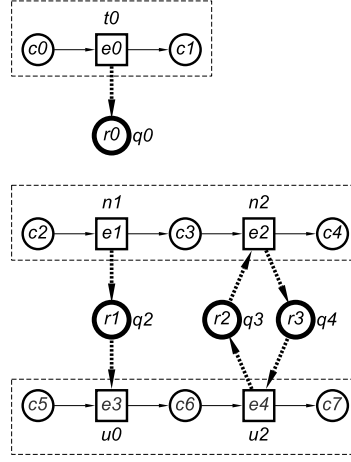


Fig. 4. A CSON-net which is a possible single run of the CSPT-net of Figure 2.

$W' \subseteq (T' \times Q') \cup (Q' \times T')$ where $T' = \bigcup T'_i$; and $h' : Q' \rightarrow Q$. It is assumed that the following hold, where $h = h' \cup \bigcup h_i$ and $F' = \bigcup F'_i$:

1. The ON_i 's and Q' are pairwise disjoint.
2. For every $r \in Q'$,
 - $|\bullet r| = 1$ and $|r\bullet| \leq 1$; and
 - if $e, f \in \bullet r\bullet$, then $\text{depth}_{h(r)}(e) = \text{depth}_{h(r)}(f)$.
3. For every $e \in T'$, the restriction of h to $\bullet e \cap Q'$ is a bijection between $\bullet e \cap Q'$ and $\bullet h(e) \cap Q$, and similarly for $e\bullet \cap Q'$ and $h(e)\bullet \cap Q$.
4. The relation $(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*$ over T' is irreflexive, where: $e \prec f$ if there is $p \in \bigcup P'_i$ with $p \in e\bullet \cap \bullet f$; and $e \sqsubset f$ if there is $r \in Q'$ with $r \in e\bullet \cap \bullet f$.
5. $h(M_{CSON}) = M_{CSPT}$, where M_{CSON} is the default initial state of CSON defined as $\bigcup M_{ON_i}$. \diamond

The above definition extends that in [7] by allowing an infinite number of nodes, and therefore provides a general meaning of a single run of a CSPT-net. To capture the behaviour systems with complex structure, we use the binary relation \sqsubset (*weak causality*) to represent a/synchronous communication between two events (see [7]). Intuitively, the original causality relation \prec represents the ‘earlier than’ relationship on the events, and \sqsubset represents the ‘not later than’ relationship. In order to ensure the resulting causal dependencies remain consistent, we require the acyclicity of not only each component non-branching process but also any path involving both \sqsubset and \prec . The condition involving the depth of two events accessing the same channel place means that the tokens flowing through channel places are based on the FIFO policy, so that the size of the subsequent full representation of the behaviours of a CSPT-net is kept low.

The CSON in Figure 4 shows a non-branching processes with the labels (alongside the nodes) coming from the CSPT-net shown in Figure 2. It corresponds, e.g., to the step sequence $\lambda = \{t_0, n_1\}\{u_0\}\{n_2, u_2\}$ in the original CSPT-net.

Branching processes of CSPT-nets. We have described two classes of structured nets, i.e., CSPT-nets and CSONs. The former is a system-level class of nets providing representations of entire systems, whereas the latter is a behaviour-level class of nets representing single runs of such systems. In this section, we will introduce a new class of branching nets which can capture the complete behaviours of CSPT-nets.

Definition 4 (branching process of CSPT-net). *Let CSPT be as in Definition 1 with M_0 being empty. A branching process of CSPT is a tuple $BCSON = (\Pi_1, \dots, \Pi_k, Q', W', h')$ such that each $\Pi_i = (ON_i, h_i)$ is a branching process of PT_i with $ON_i = (P'_i, T'_i, F'_i)$; Q' is a set of channel places; $W' \subseteq (T' \times Q') \cup (Q' \times T')$ where $T' = \bigcup T'_i$; and $h' : Q' \rightarrow Q$. It is assumed that the following hold, where $h = h' \cup \bigcup h_i$ and $F' = \bigcup F'_i$:*

1. *The ON_i 's and Q' are pairwise disjoint.*
2. *For all $r, r' \in Q'$ with $h(r) = h(r')$, as well as for all $e \in \bullet r^\bullet$ and $f \in \bullet r'^\bullet$,*

$$\text{depth}_{h(r)}(e) = \text{depth}_{h(r')}(f) \iff r = r'.$$

3. *$BCSON$ is covered in the graph-theoretic sense by a set of non-branching processes CSON of CSPT satisfying $M_{CSON} = M_{BCSON}$, where the default initial state M_{BCSON} of $BCSON$ is defined as $\bigcup M_{ON_i}$.* \diamond

Using arguments similar to those used in the case of the standard net unfoldings, one can show that there is a unique maximal branching process $BCSON_{CSPT}$, called the *unfolding* of CSPT.

A branching process of a CSPT-net consists of branching processes obtained from each component PT-net and a set of channel places. The default initial state M_{BCSON} consists of the initial states in the component branching processes. In addition, Definition 4(1) means that the component branching processes are independent, and all the interactions between them must be via the channel places. In particular, there is no direct flow of tokens between any pair of the component branching processes. Definition 4(2) implies that the occurrence depths of events inserting tokens to a channel place are the same, and are equal to the occurrence depths of events removing the tokens. Moreover, channel places at the same depth correspond to different channel places in the original CSPT-net. Finally, Definition 4(3) specifies that the label of every input and output event of a channel place in $BCSON$ matches a corresponding transition in the original CSPT-net. In general, every node and arc in the branching process belongs to at least one non-branching process of CSPT-net (CSON). This ensures that every event in the $BCSON$ is *executable* from the default initial state M_{BCSON} (i.e., it belongs to a step enabled in some reachable marking), and every condition and channel place is reachable (i.e., it belongs to the initial state or to the post-set of some executable events).

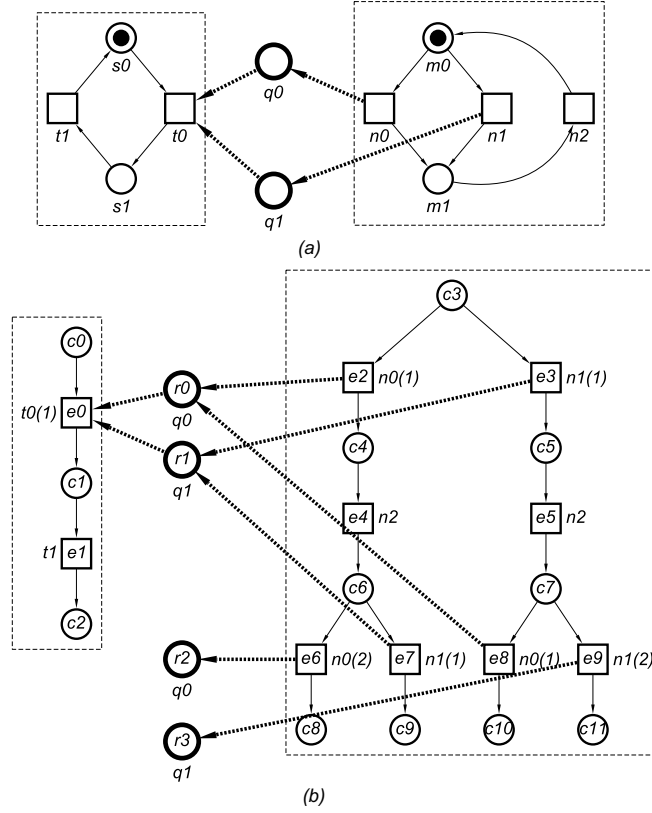


Fig. 5. (a) CSPT-net, and (b) its branching process.

Proposition 2 (safeness). *Let BCSON be as in Definition 4. Then BCSON is safe when executed from the default initial state M_{BCSON} .*

Note: This means that we treat BCSON as a CSPT-net with the initial marking obtained by inserting a single token in each condition belonging to M_{BCSON} , and safety means that no reachable marking contains more than one token in any condition, including the channel places.

The nets in Figure 3(b) and Figure 5(b) are the branching processes of the CSPT-nets showing in Figure 3(a) and Figure 5(a) respectively. We can observe that every input and output event of a channel place has the same occurrence depth which represents the token flow sequence during communication between different PT-nets. For instance, in Figure 5(b) the occurrence depths of e_0, e_2 and e_8 are $depth_{q_0}(e_0) = depth_{q_0}(e_2) = depth_{q_0}(e_8) = 1$. This means of that the transitions t_0 and n_0 were involved in a first asynchronous communication.

Remark 1. A BCSO cannot, in general, be obtained by simply unfolding every component PT-net independently and appending the necessary channel places afterwards. Proceeding in such a way can lead to a net violating Definition 4(3). This is so because an executable transition in a component PT-net does not have to be executable within the context of the CSPT-net. For example, Figure 6(b) does not show a valid branching process of the CSPT-net of Figure 2. Transition n_0 in the middle PT-net of Figure 2 can never be executed since t_0 and t_1 are in conflict, and the system is acyclic. As the result, there is no n_0 -labelled event in a corresponding branching process. Note that Figure 6(a) shows a valid BCSO since each event present there is executable. \diamond

4 Completeness of branching processes

In this section, we introduce the concept of a complete prefix of the unfolding of a CSPT-net. The prefix is a truncated part of possibly infinite unfolding which contains full reachability information about the original CSPT-net. The idea is to consider global configurations of the unfolding taking into account single runs across different component PT-nets. Then we show that the final states of all the finite global configurations correspond to the reachable markings of original CSPT-net. Using this result, it is possible to consider a finite truncation which is sufficient to represent all reachable markings.

Global configurations. A global configuration of a BCSO consists of a set of (standard) configurations, each coming from a different component branching process, joined together by channel places.

Definition 5 (global configuration). *Let BCSO be as in Definition 4. A global configuration of BCSO is a set of events $C = C_1 \cup \dots \cup C_k$ such that each C_i is a configuration of the process Π_i , and the following hold:*

1. $\bullet C \cap Q' \subseteq C^\bullet$.
2. The relation $(\sqsubset \cup \prec)^* \circ \prec \circ (\prec \cup \sqsubset)^*$ over C is irreflexive, where: $e \prec f$ if there is $p \in \bigcup P'_i$ with $p \in e^\bullet \cap f^\bullet$; and $e \sqsubset f$ if there is $r \in Q'$ with $r \in e^\bullet \cap f^\bullet$.

Moreover, if the configuration C is finite, then $\text{Fin}(C) = (M_{\text{BCSO}} \cup C^\bullet) \setminus \bullet C$ is the final state of C . The set of all global configurations of BCSO will be denoted by $\text{Conf}_{\text{BCSO}}$. \diamond

Definition 5(1) reflects the nature of a/synchronous communication between component (standard) configurations. Intuitively, if we start with an event of the global configuration which is an output event of a channel place, then there exists an input event of the same channel place that also belongs to the global configuration. Moreover, Definition 5(2) states that there are no asynchronous cycles in a global configuration.

Proposition 3 (configuration is non-branching). *Let C be a configuration as in Definition 5. Then, for all distinct $e, f \in C$, $\bullet e \cap \bullet f = e^\bullet \cap f^\bullet = \emptyset$.*

Proposition 4 (configuration is causally closed). *Let C be a configuration as in Definition 5. Then, for every $e \in C$, $p \in \bigcup P_i^!$ and $p \in e^\bullet \cap f^\bullet$ imply $f \in C$. Moreover, if $r \in Q' \cap e^\bullet$ then there is $f \in C$ such that $r \in f^\bullet$.*

Since in BCSON we use the *merging* technique in the case of channel places (i.e., different events with same occurrence depth and label will link with same instance of channel place), it is possible for a channel place to have multiple inputs or outputs. Propositions 3 and 4 imply that global configurations are guaranteed to be non-branching and causally closed w.r.t. the flow relations F' and W' . Indeed, if a channel place has more than one input (or output) events, these events are in conflict w.r.t. the flow relation F' . Hence the events belong to different configurations, and each channel place in global configuration has exactly one input and no more than one output. As a result, a global configuration retains key properties of the standard configurations, and it represents a valid execution of transitions of the original CSPT-net.

Consider the branching process in Figure 5. It has a configuration $C = \{e_0, e_1, e_2, e_4, e_7\}$ which consists of two (component) configurations $C_1 = \{e_0, e_1\}$ and $C_2 = \{e_2, e_4, e_7\}$, whereas $C' = \{e_0, e_1, e_2, e_4\}$ and $C'' = \{e_0, e_1, e_2, e_4, e_6, e_7\}$ are not valid configurations (C' has non input event for the channel place r_1 , while C'' includes two standard configurations of a single component PT-net).

Each finite configuration C has a well-defined final state determined by the outputs of the events in C . Intuitively, such a state comprises the conditions and channel places on the frontier between the events of C and events outside C . Note that a final state may contain channel places which were involved in asynchronous communications. No channel place involved in a synchronous communications can appear in $Fin(C)$, as such channel place must provide input for another event. For instance, the final state of the global configuration example above is $Fin(C) = \{c_2, c_9\}$, whereas the final state of another global configuration $C''' = \{e_2, e_4, e_6\}$ is $Fin(C''') = \{r_0, r_2, c_8\}$ which contains two asynchronous channel places.

The next result shows that a global configuration together with their outputs and the initial state form a CSON representing a non-branching process of the original CSPT-nets. And, similarly, the events of a non-branching process included in a branching one form a global configuration.

Proposition 5. *Let BCSON be as in Definition 4.*

1. *Let C be a global configuration as in Definition 5. Then $M_{BCSON} \cup C \cup C^\bullet$ are the nodes of a non-branching process of CSPT included in BCSON.*
2. *The events of any non-branching process CSON included in BCSON and satisfying $M_{CSON} = M_{BCSON}$ form a global configuration.*

Proposition 6. *Let C be a global configuration as in Definition 5. Then $h(Fin(C))$ is a reachable marking in the original CSPT-net.*

By combining Propositions 5 and 6, we obtain that finite global configurations provide a faithful representation of all the reachable marking of the original CSPT-net.

Theorem 1. *Let $BCSON_{CSPT}$ be the unfolding of CSPT. Then M is a reachable marking of CSPT if and only if $M = h(Fin(C))$, for some global configuration C of $BCSON_{CSPT}$.*

Complete prefixes of CSPT-nets. A complete prefix of the unfolding of a CSPT-net contains a full reachability information about the original CSPT-net. Such a property is referred to as *completeness*.

Finite complete prefixes of Petri nets were first introduced in McMillan's seminal work in order to avoid the state explosion problem in the verification of systems modelled with Petri nets. McMillan also provided an algorithm to generate a complete finite prefix of the unfolding which contains a full reachability information. Later, [3] refined McMillan's prefix construction algorithm to avoid creating prefixes larger than necessary.

The semantical meaning of completeness has been further addressed in [6], which extended it to more general properties. Basically, [6] associated completeness with some additional information, provided by the cut-off events which were only considered as an algorithm issue in the previous works. We can adapt the resulting notion to the current context as follows.

Definition 6 (completeness). *Let $BCSON$ be as in Definition 4, and E_{cut} be a set of events of $BCSON$. Then $BCSON$ is complete w.r.t. E_{cut} if the following hold:*

- *for every reachable marking M of CSPT, there is a finite global configuration C such that $C \cap E_{cut} = \emptyset$ and $Fin(C) = M$; and*
- *for each global configuration C of $BCSON_{CSPT}$ such that $C \cap E_{cut} = \emptyset$ and, for each event $e \notin C$ of $BCSON_{CSPT}$ such that $C \cup \{e\}$ is a global configuration of $BCSON_{CSPT}$, it is the case that e belongs in $BCSON$.*

Moreover, $BCSON$ is marking complete if it satisfies the first condition. \diamond

5 Unfolding algorithm for CSPT-net

We will now describe an algorithm for the construction of the unfolding of a CSPT-net. A key notion used by the algorithm is that of an executable event (i.e., an event which is able to fire during some execution from the default initial state) as well as that of a reachable condition or channel place (i.e., one produced by an executable event). Note that whether an event is executable in a CSPT-net is not only determined by the corresponding PT-net, but also by the behaviours of other PT-nets. This means that a component branching process in CSPT unfolding may not preserve its own unfolding structure (see Remark 1 and Figure 6(a)). In other words, there may exist events which are valid extensions in the unfolding process of a component PT-net, but become invalid when considering communication.

In particular, due to synchronous communication, it may be difficult to make sure that every extension is executable before appending it to the unfolding. Unlike the standard unfolding methods, an algorithm for CSPT-net cannot simply unfold the component branching processes adding one event at a time, and

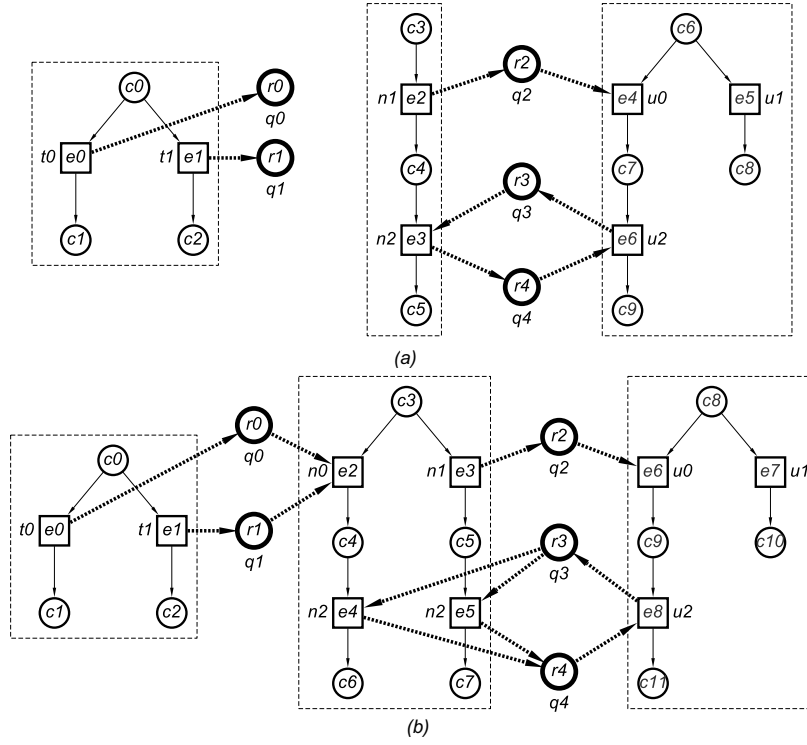


Fig. 6. (a) A valid CSPT branching process of Figure 2 (top), and (b) an invalid one (bottom).

connecting it to already existing channel places. This is because a synchronous communication in CSPT unfolding forms a cycle. It is therefore impossible to add only one of the synchronised events and guarantee its executability at the same time. Similarly, adding a synchronous event set together with all related channel places in one step may also be difficult to achieve since the use of merging may produce infinitely many events which are connected to the same channel place.

Instead, our idea is to design an algorithm which will sometimes generate non-executable events requiring tokens from channel places which have not yet been generated, in the anticipation that later on a suitable (possibly synchronous) events will provide such tokens. Roughly, the algorithm appends possible extensions together with their output conditions one by one. A new event is first marked as non-executable. The algorithm then performs an executability check for the event after constructing its a/synchronous communications. In this way, in general we obtain an ‘over-approximating unfolding’. The final stage of the algorithm can then be used to remove all the non-executable nodes.

Before providing the details of the algorithm, we introduce some auxiliary notions. In what follows, we assume that *CSPT* is as in Definition 1.

Definition 7 (local CSPT configuration). Let $e \in C$, where C is a global configuration of BCSO_N as in Definition 5. Then the local CSPT configuration of e in C , denoted by $C[e]$, is defined as $C[e] = \{f \in C \mid (f, e) \in (\prec \cup \sqsubset)^*\}$, where the relations \prec and \sqsubset are as in Definition 5. Moreover, $\text{Conf}(e) = \{C[e] \mid C \in \text{Conf}_{\text{BCSO}_N} \wedge e \in C\}$ is the set of all CSPT local configurations of e . \diamond

The CSPT local configuration of an event e in C is the set of events that are executed before (or together with) e . In general, it consists of a configuration comprising the standard local configuration of e together with a set of standard configurations coming from other branching processes. Note that an event may have different local CSPT configurations, e.g., if one of its inputs is a channel place which has multiple input events. Each such local configuration belongs to a different non-branching process. For instance, consider a global configuration $C = \{e_0, e_1, e_2, e_4, e_7\}$ in Figure 5. The CSPT local configuration of event e_0 in C is $C[e_0] = \{e_0, e_2, e_4, e_7\}$ which involves two standard local CSPT configurations, $[e_0]$ and $[e_7]$. Moreover, we can observe that the $C[e_0]$ is not the unique local configuration of e_0 , as another one is $C'[e_0] = \{e_0, e_3, e_5, e_8\}$, where $C' = \{e_0, e_1, e_3, e_5, e_8\}$.

An event may even have infinitely many local configurations. Consider again the net in Figure 5. If we continue to unfold the net, we will construct infinitely many n_0 and n_1 labelled events with occurrence depth equal to 1. All of them are input events for q_0 and q_1 labelled channel places and belong to different non-branching processes.

A/sync graphs. In order to improve the efficiency of unfolding procedure, checking for the existence of a local CSPT configuration of an event can be reduced to the problem of exploring the causal dependencies between channel places.

Below we assume that if C_i is a configuration of the unfolding of the i -th component PT-net, and $e \in C_i$ and $q \in Q$ are such that $(h(e), q) \in W$ (or $(q, h(e)) \in W$), then $r = (q, \text{depth}_q(e))$ belongs to the set of implicit channel places Q_{C_i} connected to C_i . Moreover, the label of r is q , and $(e, r) \in W_{C_i}$ (resp. $(r, e) \in W_{C_i}$) is the corresponding implicit arc.

Definition 8 (a/sync graph). Let C_i be a configuration of the unfolding of the i -th component PT-net. Then the a/sync graph of C_i is defined as $\mathcal{G}(C_i) = (Q_{C_i}, \hat{\succ}_{C_i}, \hat{\sqsubset}_{C_i})$, where $\hat{\succ}_{C_i}, \hat{\sqsubset}_{C_i}$ are two binary relations over Q_{C_i} such that, for every $r, r' \in Q_{C_i}$:

- $r \hat{\succ}_{C_i} r'$ if there are two distinct $e, f \in C_i$ such that $(r, e), (f, r') \in W_{C_i}$, and e precedes f within C ; and
- $r \hat{\sqsubset}_{C_i} r'$ if there is $e \in C_i$ with $(r, e), (e, r') \in W_{C_i}$. \diamond

$\mathcal{G}(C_i)$ captures relationships between the input and output channel places of a configuration of the unfolding of an individual component system. Its nodes are the channel places involved in C_i . Moreover, $r \hat{\succ}_{C_i} r'$ if there is a path from r to r' involving more than one event of C_i , and $r \hat{\sqsubset}_{C_i} r'$ if r is an input and r' an output of some event in C_i .

Figure 7(a) shows the unfolding of each component PT-net of Figure 2 together with their input and output channel places. By exploring the relations

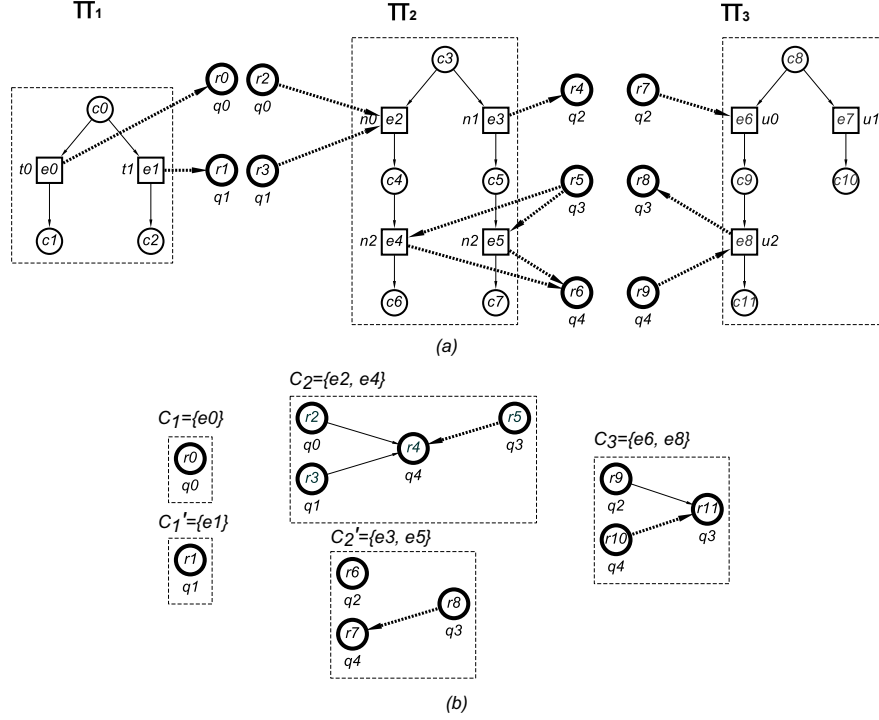


Fig. 7. (a) unfoldings of three component PT-nets of Figure 2 (together with their implicit channel places), and (b) a/sync graphs of configurations derived from these unfoldings.

between those channel places, we are able to generate a/sync graph for any configuration. For example, Figure 7(b) shows five a/sync graphs of the configurations derived from Figure 7(a), where the relations $\hat{\succ}_{C_i}$ and $\hat{\sqsubset}_{C_i}$ are represented by solid arcs and thick dashed arcs, respectively. For the left-hand side PT-net Π_1 , we have: $\mathcal{G}(C_1) = (\{r_0\}, \emptyset, \emptyset)$ and $\mathcal{G}(C_1') = (\{r_1\}, \emptyset, \emptyset)$. The a/sync graphs of the configurations in Π_2 are: $\mathcal{G}(C_2) = (\{r_2, r_3, r_4, r_5\}, \{(r_2, r_4), (r_3, r_4)\}, \{(r_5, r_4)\})$ and $\mathcal{G}(C_2') = (\{r_6, r_7, r_8\}, \emptyset, \{(r_8, r_7)\})$ and for the right-hand side PT-net Π_3 , we have $\mathcal{G}(C_3) = (\{r_9, r_{10}, r_{11}\}, \{(r_9, r_{11}), (r_{10}, r_{11})\})$.

Given a set of a/sync graphs $\mathcal{G}(C_1), \dots, \mathcal{G}(C_k)$ extracted for the k component systems, we call these graphs compatible if all inputs are produced and there is no cycle involving $\hat{\succ}$.

Definition 9 (compatibility of a/sync graphs). Let C_i ($i = 1, \dots, k$) be a configuration of the unfolding of the i -th component PT-net, and $\mathcal{G}(C_i) = (Q_{C_i}, \hat{\succ}_{C_i}, \hat{\sqsubset}_{C_i})$. Then C_1, \dots, C_k are compatible configurations if the following hold:

1. if $(r, e) \in W_{C_i}$ then that there is $j \neq i$ such that $r \in Q_{C_j}$; and

2. the relation $(\hat{\sqsubset} \cup \hat{\sqsupset})^* \circ \hat{\sqsubset} \circ (\hat{\sqsupset} \cup \hat{\sqsubset})^*$ is irreflexive, where $\hat{\sqsubset} = \bigcup \hat{\sqsubset}_{C_i}$ and $\hat{\sqsupset} = \bigcup \hat{\sqsupset}_{C_i}$. \diamond

In Figure 7, configurations C_1, C'_2, C_3 are compatible since the q_3 -labelled input channel place r_8 in $\mathcal{G}(C'_2)$ is present in $\mathcal{G}(C_3)$ (i.e., r_{11}), and the input channel places r_9, r_{10} (labelled by q_2 and q_4 respectively) in $\mathcal{G}(C_3)$ are all present in $\mathcal{G}(C'_2)$. On the other hand, we can observe that there are no compatible configurations which involve C_2 , i.e., neither configurations C_1, C_2, C_3 nor C'_1, C_2, C_3 are compatible. This is because the producers of r_2 and r_3 are in conflict in Π_1 .

Theorem 2. *Let C_1, \dots, C_k be configurations of the unfoldings of the component PT-nets, and $C = C_1 \cup \dots \cup C_k$. Then C is a global configuration if and only if C_1, \dots, C_k are compatible.*

Therefore, one can obtain the CSPT local configurations of an event e by checking whether there are compatible configurations C_1, \dots, C_k such that e belongs to one of them. Such a task can be made efficient by working with the graphs $\mathcal{G}(C_1), \dots, \mathcal{G}(C_k)$. In fact, one can just check those configurations which have dependencies on e .

Unfolding algorithm. The unfolding algorithm we are going to present significantly differs from the existing net unfolding algorithms. The key difference is that during the unfolding procedure we will be constructing nodes and connections which will not necessarily be the part of the final unfolding. This is due to the presence of synchronous communication within our model. More precisely, in the net being constructed there will be *executable* and *non-executable* events and conditions. The former will definitely be included in the resulting unfolding, whereas the latter cannot be yet included due to the absence of event(s) which are needed for communication. If, at some later stage, the missing events are generated, then the previously non-executable event and the conditions (and channel places) it produced become executable.

Although the net Unf generated by the algorithm may not strictly speaking be a branching process during its creation, we will as far as it is possible treat it as such. In particular, we will call an event e *executable* if e has at least one local configuration, i.e., $Conf(e) \neq \emptyset$. This happens if we have generated enough events to find at least one local CSPT configuration of e in Unf .

Intuitively, an executable event is the event belonging to at least one single run of a BCSO. For the example net in Figure 6(b), e_6 is an executable event since there exists a local CSPT configuration of e_6 : $C[e_6] = \{e_0, e_3, e_6\}$, where $C = \{e_0, e_3, e_6\}$. On the other hand, event e_2 is non-executable because it does not have any local configuration (we have seen the example of Figure 7 that there are no compatible configurations which involve e_2). Therefore, Figure 7(b) is not a valid CSPT branching process since according to Definition 4(3) every event in BCSO is executable. If we remove e_2 together with its successors, then all events in the new net become executable indicating the net is a valid BCSO (Figure 6 (a)).

Proposition 7. *Let e be an executable event in $BCSON$. Then each event appearing in $Conf(e)$ is executable.*

Algorithm 1 (unfolding of CSPT-net)

input: $CSPT$ — CSPT-net

output: Unf — unfolding of $BCSON$

$nonexe \leftarrow \emptyset$

$Unf \leftarrow$ the empty branching process

add instances of the places in the initial marking of $CSPT$ to Unf

add all possible extensions of Unf to pe

while $pe \neq \emptyset$ **do**

 remove e from pe

$addConnections(e)$

if $Conf(e) \neq \emptyset$ **then**

for all event f in configurations of $Conf(e)$ **do**

 remove f and all its output conditions from $nonexe$ (if present there)

 add all possible extensions of Unf to pe

delete the nodes in $nonexe$ together with adjacent arcs from Unf

The procedure for constructing the unfolding of a CSPT-net is presented as Algorithm 1.

The first part of the algorithm adds conditions representing the initial marking of the CSPT-net being unfolded. Notice that the set $nonexe$ of non-executable events and conditions is set to empty. It also adds possible extensions to the working set pe . The concept of a non-executable condition greatly improves the efficiency of the above algorithm since a *possible extension* of Unf is a pair $e = (t, B)$ with $h(e) = t$ where t is a transition of $CSPT$, and B is a set of conditions of Unf such that:

- B is a co-set in one of the subnets of Unf and $B \cap nonexe = \emptyset$;
- $h(B)$ are all the input non-channel places of t ; and
- $(t, B) \notin pe$ and Unf contains no t -labelled event with the non-channel place inputs B .

The pair (t, B) is an event used to extend $BCSON$ without considering channel places. We use the standard condition of a possible extension to choose events that can be added to a component branching process (i.e., $h(B) = \bullet t \cap P'$), while constructing the related a/synchronous communications in a separate step. In such a way, the complexity of appending groups of synchronous events is significantly reduced. Note that a possible extension e has precisely determined channel place connections since the depth values are fully determined.

Algorithm 2 provides the details of appending a possible extension e to $BCSON$ as well as constructing related channel place structure after removing e from pe .

Algorithm 2 (adding new event and a/sync connections)

```

procedure addConnections (input:  $e = (t, B)$ )
  add  $e$  to  $Unf$  and  $nonexe$ 
  create and add all the standard post-conditions of  $e$  to  $Unf$  and  $nonexe$ 
  for all channel place  $q \in {}^\bullet t^\bullet$  do
    let  $r = (q, k)$  where  $k = depth_q(e)$ 
    if there is no  $r = (q, k)$  in  $Unf$  then
      add  $q$ -labelled channel place  $r$  to  $Unf$  and  $nonexe$ 
      add a corresponding arc between  $r$  and  $e$ 

```

Each new extension and its output conditions are immediately marked as non-executable. The conditions in $nonexe$ set also indicate that they are unable to be used for deciding any further possible extension. In this way we can avoid any unnecessary extension and make sure the predecessors of every new event is executable.

The procedure then creates the a/synchronous communications of the input event if it is required. Given an event e , for every input or output channel place q of its corresponding transition $h(e)$ in the original CSPT-net, we search in Unf for the matching channel place (i.e., its label is q and its depth value equals to the occurrence depth of e). Then we create a direct connection if such a channel place exists. Otherwise, we add a new instance of the channel place together with the corresponding arc.

After adding the implicit channel places connected to e (or creating the connection for those which already existed) together with the corresponding arcs, we are able to obtain the local configuration of e by looking for compatible configurations C_1, \dots, C_k of the component nets (which may contain non-executable events) such that e belongs to one of the C_i 's. If e is executable ($Conf(e) \neq \emptyset$), we make all non-executable events in $Conf(e)$ together with their post-conditions executable (see Proposition 7). We also generate new potential extensions (each such extension must use at least one of conditions which have just been made executable). Then another waiting potential extension (if any) is processed.

The algorithm generally does not terminate when the original CSPT-net is not acyclic, and the non-executable nodes are removed at the end of the algorithm. An example run of the algorithm is presented in the appendix.

6 Conclusions and Future Work

The unfolding algorithm presented in this paper is based on standard unfolding method, which essentially works by appending possible extension one by one. A potentially very efficient approach for the construction of the unfolding could be to use the parallel unfolding technique [4]. One can, for example, unfold each component branching process in parallel, by temporarily ignoring any a/synchronous issues. The procedures of appending channel places as well as executability checking (removing unnecessary events) would proceed in parallel.

In future we intend to explore the generation of finite complete prefixes of CSPT-nets. In the case of PT-nets, this relies on the notion of *cut-off* events, which are roughly events in the unfolding that produce a marking already produced by other events with smaller histories. In general, it is impossible to generate a finite complete prefix of the unfolding of a CSPT-net even if the component PT-nets are safe. The reason is that the channel places linking the component PT-nets can be unbounded due to asynchronous communication. However, if all communications are synchronous, this is no longer a problem. Finally, the implementation of the CSPT model and its unfolding to the SON-based [11] tool are left for the future works.

References

1. Best, E., Fernández, C.: Nonsequential Processes: A Petri Net View, vol. 13 of EATCS Monographs in Theoretical Computer Science. Springer-Verlag (1988)
2. Engelfriet, J.: Branching processes of Petri nets. *Acta Informatica* 28(6), 575–591 (1991)
3. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan’s unfolding algorithm. In: *Formal Methods in System Design*. pp. 87–106. Springer-Verlag (1996)
4. Heljanko, K., Khomenko, V., Koutny, M.: Parallelisation of the Petri net unfolding algorithm. In: *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. pp. 371–385. TACAS ’02, Springer-Verlag, London, UK, UK (2002)
5. Khomenko, V., Kondratyev, A., Koutny, M., Vogler, W.: Merged processes: A new condensed representation of Petri net behaviour. *Acta Informatica* 43(5), 307–330 (2006)
6. Khomenko, V., Koutny, M., Vogler, W.: Canonical prefixes of Petri net unfoldings. *Acta Inf.* 40(2), 95–118 (2003)
7. Kleijn, J., Koutny, M.: Causality in structured occurrence nets. In: *Dependable and Historic Computing*. vol. 6875, pp. 283–297. Springer Berlin Heidelberg (2011)
8. Koutny, M., Randell, B.: Structured occurrence nets: A formalism for aiding system failure prevention and analysis techniques. *Fundamenta Informaticae* 97(1), 41–91 (Jan 2009)
9. Li, B.: Branching processes of communication structured PT-nets. In: *Proceeding. vol. 13th International Conference On Application of ConCurrency to System Design (ACSD)*, pp. 243–246 (2013)
10. Li, B., Koutny, M.: Unfolding cspt-nets. Tech. Rep. CS-TR-1463, School of Computing Science, Newcastle University (2015)
11. Li, B., Randell, B.: Soncraft user manual. Tech. Rep. CS-TR-1448, School of Computing Science, Newcastle University (Feb 2015)
12. McMillan, K.L., Probst, D.: A technique of state space search based on unfolding. *Formal Methods in System Design* 6(1), 45–65 (Jan 1995)
13. Randell, B.: Occurrence nets then and now: the path to structured occurrence nets. In: *Applications and Theory of Petri Nets*. pp. 1–16. Springer Berlin Heidelberg (Jun 2011)
14. Randell, B., Koutny, M.: Failure: their definition, modelling and analysis. In: *Theoretical Aspects of Computing–ICTAC 2007*. pp. 260–274. Springer (Sep 2007)

Discovery of Functional Architectures From Event Logs

Jan Martijn E.M. van der Werf and Erwin Kaats

Department of Information and Computing Science
Utrecht University

P.O. Box 80.089, 3508 TB Utrecht, The Netherlands

`j.m.e.m.vanderwerf@uu.nl`, `e.j.kaats@students.uu.nl`

Abstract. The functional architecture focuses on decomposing functionality into modules that offer certain features. These features require interactions in order to complete their functionality. However, functional architectures typically only focus on the static aspects of the system design. Additional modeling techniques, such as message sequence charts are often used in the early phases of software design to indicate how the software should behave.

In this paper we investigate the use of process discovery techniques to discover from these scenarios the internal behavior of individual components. Based on event logs, this paper presents an approach (1) to derive the information flows between features, (2) identify the internal behavior of features, and (3) to discover the order between features within a module. The approach results in a sound workflow model for each module. We illustrate the approach using a running example of a payment system.

1 Introduction

One of the principle tasks of a software architect is to design a software system [17], i.e., to organize the software elements the system is composed of in sets of structures, to allow reasoning about the system [4]. Many different Architectural Description Languages (ADLs) exist to document software architecture. However, due to the large competitive market in the software product industry, architecture is often neglected in software product organizations [14]. Hence, not many ADLs are used in practice. As experienced in [14], in software product organizations, architects rather use informal architectural models as an instrument of communication and discussion.

An important aspect of software architecture is the functionality it offers. To decompose and specify the functionality of software, the authors of [6] introduced the Functional Architecture Model (FAM), which offers the desired modeling technique used by many software architects in software product organizations [14]. The FAM separates the functionality into so-called features that are offered by the different modules the system is decomposed into.

Features interact with other features via information flows to offer their functionality. However, FAM only offers a static view on this interaction, i.e., the information flow only shows possible interactions, but imposes no order on or dependencies between these flows. Thus, to show how functionality is offered by the system, the architect requires additional models. One way is to define scenarios on top of the models, in which the architect can specify which features interact in which order. These scenarios then result in event logs, that can be analyzed using process mining techniques [2]. Another source for discovering the possible interactions between features is the use of system execution data [21], mapping events to the (partial) execution of features. In this way, execution data can be used to reconstruct a software architecture.

In software product organizations, time to market is often a more important priority than having a properly documented software architecture. Consequently, architecture documentation is often outdated or even missing [9]. Therefore, discovering architectural models help such organizations in maintaining their software products. In this paper, we investigate the possibility to use process mining techniques to discover the functional architecture of the system from an event log. We thereby focus on three basic questions on the functional architecture:

1. Which features interact?
2. What is the internal behavior of features?
3. What is the order in which features are executed within a module?

The first question focuses on the discovery of information flows: given an event log, is it possible to derive which features interact? Next, we investigate whether it is possible to derive the internal behavior of features based on event logs. In other words, we focus on the question how does a feature use its information flows to complete its functionality. The last question deals with the high-level view of the functional architecture. To execute the system's functionality, the features within a module are called in a certain order. Can process discovery techniques be used to discover these orders?

The remainder of this paper is structured as follows. To illustrate the approach, Sec. 2 presents a running example which we will use throughout the paper. Next, Sec. 3 presents the basic notions used in the paper. Section 4 introduces the functional architecture model in more detail, after which in Sec. 5 we will focus on solving the three questions posed in the introduction. Section 6 concludes the paper.

2 Running Example

As an running example, consider the Payment System as introduced in [10]. The system consists of three modules, *Debtor*, *Payment* and *Creditor*. The payment module serves as an intermediate between the Debtor and the Creditor. An example of such a payment module is the european SEPA standard. The payment module initiates a transaction, which the debtor needs to accept. If the debtor accepts, the payment is continued, and the creditor is contacted to start the

transaction. If for some reason the creditor rejects the transaction, the debtor is notified, and the transaction is terminated. Similarly, if the creditor accepts, the payment is passed to the debtor, and finally, the creditor receives the final payment information.

As the software evolved into the current system, no precise model exists that specifies the behavior of this system. The system only recorded the order in which the different features of the modules have been called in an event log, as shown in Tbl. 1. Each pair in the table represents the feature and the module to which that feature belongs. For readability, the features and modules are abbreviated in this event log.

The system is decomposed into three modules: the *Debtor* module (X), the *Payment* (Y) module, and the *Creditor* (Z). Based on the event log, the software architect finds the following features:

- Receive transaction request (A);
- Reject transaction (B);
- Accept transaction (C);
- Cancel transaction (D);
- Initiate payment (E);
- Send payment details (F);
- Archive transaction request (G).
- Send transaction request (H);
- Reject transaction request (I);
- Initiate creditor (J);
- Cancel transaction (K);
- Initiate payment (O);
- Handle payment (M);
- Archive transaction (N).
- Start transaction (Q);
- Handle transaction (S).

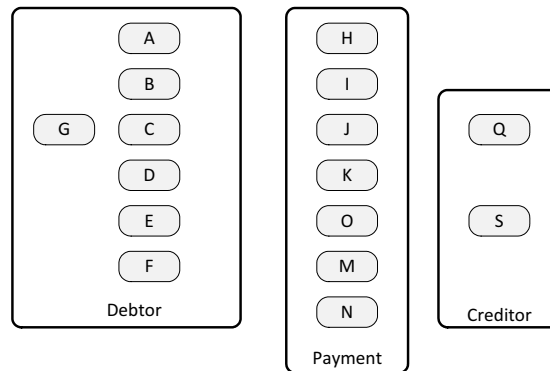


Fig. 1. Initial functional architecture model of the running example

Case	Trace
1	(H, Y), (A, X), (B, X), (G, X), (I, Y), (N, Y)
2	(H, Y), (A, X), (B, X), (I, Y), (G, X), (N, Y)
3	(H, Y), (A, X), (B, X), (I, Y), (N, Y), (G, X)
4	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (G, X), (N, Y)
5	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (D, X), (N, Y), (G, X)
6	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (K, Y), (N, Y), (D, X), (G, X)
7	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (G, X), (M, Y), (S, Z), (N, Y)
8	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (G, X), (M, Y), (N, Y), (S, Z)
9	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (S, Z), (N, Y)
10	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (G, X), (N, Y), (S, Y)
11	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (G, X), (N, Y)
12	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (S, Z), (N, Y), (G, X)
13	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (G, X), (S, Z)
14	(H, Y), (A, X), (C, X), (J, Y), (Q, Z), (S, Z), (O, Y), (E, X), (F, X), (M, Y), (N, Y), (S, Y), (G, X)

Table 1. System execution data of the payment system

Based on this information, the architect can draw the modules with their features, as shown in Fig. 1. In the remainder of this paper, we investigate a method to use event logs, such as the one shown in Tbl. 1, to complete the diagram and derive a behavioral specification of the system.

3 Preliminaries

Let S be a set. The powerset of S is denoted by $\mathcal{P}(S) = \{S' \mid S' \subseteq S\}$. We use $|S|$ for the number of elements in S . Two sets U and V are *disjoint* if $U \cap V = \emptyset$. Some set S with relation \leq is a partial order, denoted by (S, \leq) , iff \leq is reflexive, i.e. $a \leq a$ for all $a \in S$, antisymmetric, i.e. $a \leq b$ and $b \leq a$ imply $a = b$ for all $a, b \in S$, and transitive, i.e. $a \leq b$ and $b \leq c$ imply $a \leq c$ for all $a, b, c \in S$. Given a relation $R \subseteq S \times S$ for some set S , we denote its transitive closure by R^+ , and the transitive and reflexive closure by R^* .

A *bag* m over S is a function $m : S \rightarrow \mathbb{N}$, where $\mathbb{N} = \{0, 1, \dots\}$ denotes the set of natural numbers. We denote e.g. the bag m with an element a occurring once, b occurring three times and c occurring twice by $m = [a, b^3, c^2]$. The set of all bags over S is denoted by \mathbb{N}^S . Sets can be seen as a special kind of bag where all elements occur only once; we interpret sets in this way whenever we use them in operations on bags. We use $+$ and $-$ for the sum and difference of two bags, and $=, <, >, \leq, \geq$ for the comparison of two bags, which are defined in a standard way.

A *sequence* over S of length $n \in \mathbb{N}$ is a function $\sigma : \{1, \dots, n\} \rightarrow S$. If $n > 0$ and $\sigma(i) = a_i$ for $i \in \{1, \dots, n\}$, we write $\sigma = \langle a_1, \dots, a_n \rangle$. The length of a sequence is denoted by $|\sigma|$. The sequence of length 0 is called the *empty sequence*, and is denoted by ϵ . The set of all finite sequences over S is denoted by S^* . We write $a \in \sigma$ if a $1 \leq i \leq |\sigma|$ exists such that $\sigma(i) = a$. *Concatenation* of two sequences $\nu, \gamma \in S^*$, denoted by $\sigma = \nu; \gamma$, is a sequence defined by $\sigma : \{1, \dots, |\nu| + |\gamma|\} \rightarrow S$, such that $\sigma(i) = \nu(i)$ for $1 \leq i \leq |\nu|$, and $\sigma(i) = \gamma(i - |\nu|)$ for $|\nu| + 1 \leq i \leq |\nu| + |\gamma|$. A sequence σ can be projected over some set U , denoted by $\sigma|_U$, and is inductively defined by $\epsilon|_U = \epsilon$, $(\langle a \rangle; \sigma)|_U = \langle a \rangle; \sigma|_U$ if $a \in U$, and $(\langle a \rangle; \sigma)|_U = \sigma|_U$ otherwise.

Petri Nets A *Petri net* [16] is a tuple $N = \langle P, T, F \rangle$ where (1) P and T are two disjoint sets of *places* and *transitions* respectively; and (2) $F \subseteq (P \times T) \cup (T \times P)$ is a *flow relation*. The elements from the set $P \cup T$ are called the *nodes* of N . Elements of F are called *arcs*. Places are depicted as circles, transitions as squares. For each element $(n_1, n_2) \in F$, an arc is drawn from n_1 to n_2 .

Let $N = \langle P, T, F \rangle$ be a Petri net. Given a node $n \in (P \cup T)$, we define its *preset* ${}^\bullet n = \{n' \mid (n', n) \in F\}$, and its *postset* $n^\bullet = \{n' \mid (n, n') \in F\}$. We lift the notation of preset and postset to sets. Given a set $U \subseteq (P \cup T)$, ${}^\bullet U = \bigcup_{n \in U} {}^\bullet n$ and $U^\bullet = \bigcup_{n \in U} n^\bullet$. If the context is clear, we omit the N in the subscript.

A *marking* of N is a bag $m \in \mathbb{N}^P$, where $m(p)$ denotes the number of *tokens* in place $p \in P$. If $m(p) > 0$, place p is called *marked* in marking m . A Petri net N with corresponding marking m is written as (N, m) and is called a *marked Petri net*. Given a marked Petri net (N, m) , transition t is enabled, denoted by $(N, m)[t]$, if ${}^\bullet t \leq m$. If transition t is enabled in (N, m) , it can fire, resulting in a new marking m' , denoted by $(N, m)[t] \rightarrow (N, m')$, such that $m' + {}^\bullet t = m + t^\bullet$. We lift the firing of transitions to the firing of sequences in a standard way, i.e., a sequence $\sigma \in T^*$ of length n is enabled in (N, m) if markings m_0, \dots, m_n exist, such that $m = m_0$ and $(N, m_{i-1})[\sigma(i)] \rightarrow (N, m_i)$ for all $1 \leq i \leq n$. A marking m' is reachable from some marking m in N , denoted by $(N, m)[*] \rightarrow (N, m')$, if a firing sequence $\sigma \in T^*$ exists such that $(N, m)[\sigma] \rightarrow (N, m')$. A marking m' is a *home marking* of (N, m) , if for all markings m'' with $(N, m)[*] \rightarrow (N, m'')$, we have $(N, m'')[*] \rightarrow (N, m')$.

A special class of Petri nets are the workflow nets [1]. A workflow net is a tuple $\langle P, T, F, i, f \rangle$ with $\langle P, T, F \rangle$ a Petri net, (2) $i \in P$ is the only place with no incoming transitions, (3) $f \in P$ is the only place with no outgoing transitions, i.e., ${}^\bullet i = f^\bullet = \emptyset$, and (4) all transitions have at least one incoming and one outgoing arc, i.e., ${}^\bullet t \neq \emptyset \neq t^\bullet$ for all $t \in T$.

Open Petri Nets Within a network of asynchronously communicating systems, messages are passed between the elements within the network. The approach we follow is based on Open Petri nets [5]. Communication in an open Petri net (OPN) is represented by special places, called the *interface places*. An interface place is either an *input place*, receiving messages from the outside, or an *output place* that sends messages to the outside of the OPN. An input place is a place that has only outgoing arcs, and an output place has no incoming arcs.

Definition 1. An Open Petri net is defined as an 7-tuple $\langle P, I, O, T, F, i, \Omega \rangle$ where (1) $\langle P \cup I \cup O, T, F \rangle$ is a Petri net; (2) P is a set of internal places; (3) I is a set of input places, and ${}^\bullet I = \emptyset$; (4) O is a set of output places, and $O^\bullet = \emptyset$; (5) P , I and O are pairwise disjoint; (6) $i \in \mathbb{N}^P$ is the initial marking, and (7) $\Omega \subseteq \mathbb{N}^P$ is the set of final markings. We call the set $I \cup O$ the interface places of the OPN. An OPN is called closed if $I = O = \emptyset$.

An important behavioral property for OPNs is termination: an OPN should always have the possibility to terminate properly. We identify two termination properties: weak termination and soundness.

Definition 2. Let $\langle P, I, O, T, F, i, f \rangle$ be an OPN. It is weakly terminating, if for every reachable marking of the marked Petri net $(\langle P \cup I \cup O, T, F \rangle, i)$ a marking $f \in \Omega$ can be reached.

It is sound, if for every reachable marking of the marked Petri net $(\langle P, T, F \rangle, i)$ a marking $f \in \Omega$ can be reached.

Communication between OPNs is done via the interface places. Two OPNs can only communicate if the input places of the one are the output places of the other, and vice versa.

Definition 3. Two OPNs A and B are composable, denoted by $A \oplus B$, if and only if $(I_A \cap O_B) \cup (O_B \cap I_A) = (P_A \cup T_A \cup I_A \cup O_A) \cap (P_B \cup T_B \cup I_B \cup O_B)$.

If A and B are composable, they can be composed into a new OPN, denoted by $A \oplus B$, with $A \oplus B = \langle P, I, O, T, F, i, \Omega \rangle$ where $P = P_A \cup P_B \cup G$; $I = (I_A \cup I_B) \setminus G$; $O = (O_A \cup O_B) \setminus G$; $T = T_A \cup T_B$; $F = F_A \cup F_B$; $i = i_A + i_B$; and $f = \Omega_A \cup \Omega_B$ with $G = (I_A \cap O_B) \cup (O_B \cap I_A)$.

Event Logs and Behavioral Profiles Although event logs are defined as a tuple consisting of a set of case identifiers, events, and an attribute mapping [2], it is in this paper sufficient to consider an event log, denoted by \mathcal{L} , as a set of sequences over some alphabet T , i.e., $\mathcal{L} \subseteq T^*$. Given an event log \mathcal{L} , we define the *successor relation* [20] by $a <_{\mathcal{L}} b$ if a sequence $\sigma \in \mathcal{L}$ and $1 \leq i \leq |\sigma|$ exist, such that $\sigma(i) = a$ and $\sigma(i+1) = b$. Using the successor relation, we define the *behavioral profile* $(\rightarrow_c, \parallel_c, +_c)_{\mathcal{L}}$ as three relations: (1) the causality relation \rightarrow_c is defined by $a \rightarrow_c b$ iff $a <_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$, (2) the concurrency relation \parallel_c , which is defined by $a \parallel_c b$ iff both $a <_{\mathcal{L}} b$ and $b <_{\mathcal{L}} a$, and (3) the exclusive relation $+_c$ is defined by $a +_c b$ iff both $a \not<_{\mathcal{L}} b$ and $b \not<_{\mathcal{L}} a$ [20]. If the context is clear, we omit the subscript.

Given a marked Petri net (N, m) with $N = \langle P, T, F \rangle$, an event log $\mathcal{L} \subseteq T^*$ is called *complete* with respect to (N, m) iff traces $\sigma_1, \sigma_2 \in T^*$ exist such that $(N, m)[\sigma_1; \langle a, b \rangle; \sigma_2](N, \cdot)$ implies $a <_{\mathcal{L}} b$ for all $a, b \in T$.

4 Functional Architectures

To model the overview of a system, the modules it consists of, and the features these modules offer, we propose the use of the *functional architecture model* (FAM). The functional architecture of a system is “an architectural model which represents at a high level the software products major functions from a usage perspective, and specifies the interactions of functions, internally between each other and externally with other products” [6]. It offers modules containing features. Features of different modules interact via so-called *information flows*.

An example is shown in Fig. 2(a). The FAM contains 1 context module, E , 7 modules, A, B, C, D, X, Y and Z . Modules have features, depicted by the rounded rectangles. For example, module C contains two features, K and L . Between features of different modules, information flows exist, e.g., the information flow (F, q, L) between modules A and C .

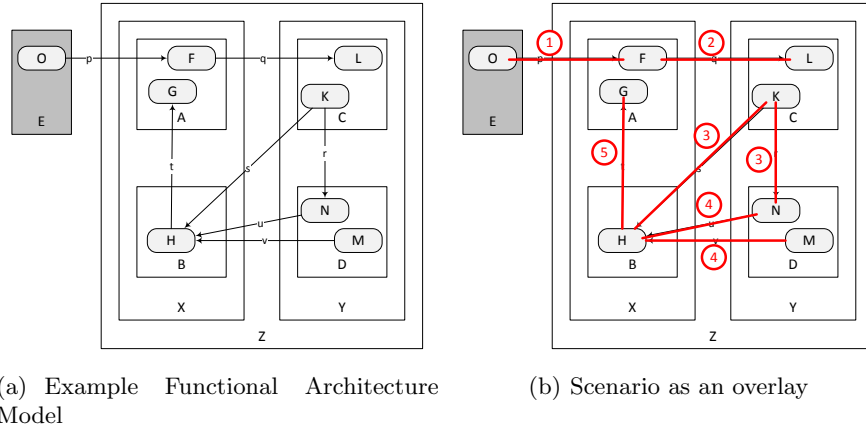


Fig. 2. Example Functional Architecture Model and corresponding scenario as overlay

Definition 4. A Functional Architecture Model (FAM) is defined as a 6-tuple $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ where

- \mathcal{M} is a finite set of modules;
- \mathcal{C} is a finite set of context modules;
- \mathcal{F} is a finite set of features;
- $h : \mathcal{M} \rightarrow \mathcal{M}$ is the hierarchy function, such that the transitive closure h^* is irreflexive;
- $m : \mathcal{F} \rightarrow \mathcal{M} \cup \mathcal{C}$ is a feature map that maps each feature to a module, possibly in the context, and this module does not have any children, i.e. $h^{-1}(m(F)) = \emptyset$ for all $F \in \mathcal{F}$;
- $\rightarrow \subseteq \mathcal{F} \times \Lambda \times \mathcal{F}$ is the information flow, with Λ the label universe, such that for $(A, l, B) \in \rightarrow$ we have $m(A) \neq m(B)$. The labels for the information flows are unique per feature, i.e., (A, l, B) and (A, l, C) imply $B = C$ for all labels $l \in \Lambda$ and $(A, l, B), (A, l, C) \in \rightarrow$.

Although the information flows define the possible interactions between modules, it remains a static overview of the system. Therefore, one can use scenarios on top of the functional architecture, e.g. by creating an overlay, highlighting the information flows that are executed and the order in which they should occur. Formally, we represent a scenario as a partial order.

Definition 5. Let $F = \langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ be a FAM. A scenario of F is a pair $(S, <)$ with $S \subseteq \rightarrow$, such that (S, \leq) with $\leq = <^*$ is a partial order.

An example is shown in Fig. 2(b). The scenario implied by the overlay can be represented by a partial order induced by $(O, p, F) < (F, q, L)$, $(F, q, L) < (K, s, H)$, $(F, q, L) < (K, r, N)$, $(K, r, N) < (N, u, H)$, $(K, s, H) < (H, t, G)$, $(N, u, H) < (H, t, G)$, $(K, r, N) < (M, v, H)$, and $(M, v, H) < (H, t, G)$.

However, such scenarios are typically not specified. Another important drawback of such scenarios is their analyzability. Although each scenario can be checked, the consistency between the different scenarios remains a difficult task. Therefore, in the remainder of this paper, we search for a method to derive the behavioral specification as a network of asynchronously communicating systems, given the system execution data produced by the actual system in the form of event logs.

5 Discovery of a Functional Architecture

In this section, we study the possibilities process mining [2] offers to generate Petri nets for each of the different modules a system consists of. Event logs describe the order in which features of a system have been executed. Such event logs are system wide. Instead of each module having its own event log, only global sequences exist, i.e., sequences concatenate the executed features over all modules. As FAM only allows features to be contained in a single module, we assume that each feature belongs to exactly one module. Also, FAM prescribes communication to be one-directional, i.e., given two communicating features A and B , we assume that either A sends a message to B , or vice versa, that B sends a message to A , but not both.

The behavioral specification of a system is three-fold: (1) communication between modules via their features, (2) the internal behavior within each feature, and (3) the order in which features are called within a module. In this section, we explore all three types of behavioral specification to come to a composed system of asynchronously communicating systems.

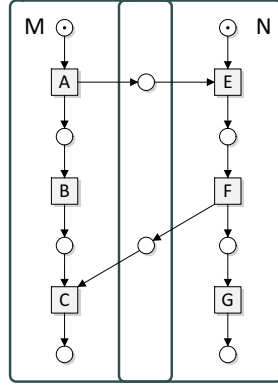
In the remainder, let \mathcal{L} be an event log over a set of features T , and let $\mathfrak{R} : T \rightarrow M$, with M the set of modules, be a function that maps each feature onto the module that contains that feature.

5.1 Communication between Features

Communication between modules within a system is asynchronous of nature: messages are sent between features in order to complete their functionality. Within an event log, we need to consider the order in which events or features occur. For example, given some trace σ , if the resource is different for two subsequent events, i.e., $\mathfrak{R}(\sigma(i)) \neq \mathfrak{R}(\sigma(i+1))$, then this might indicate that the former sends a message to the latter. This is expressed by the communication successor.

Definition 6 (Communication successor). *Let $\mathcal{L} \subseteq T^*$ be an event log. We define the communication successor relation $\ll_{\mathcal{L}} \subseteq T \times T$ by $A \ll_{\mathcal{L}} B$ iff $\mathfrak{R}(A) \neq \mathfrak{R}(B)$, $\sigma(i) = A$, and $\sigma(i+1) = B$ for some $\sigma \in \mathcal{L}$ and $1 \leq i < |\sigma|$.*

Although at first sight the communication successors seem to work, we need to remember the concurrent nature of asynchronous communication. Consider for example the communication between modules M and N as depicted in Fig. 3, and

Fig. 3. Modules M and N

Case	Trace
1	A, E, F, G, B, C
2	A, E, F, B, G, C
3	A, E, B, F, G, C
4	A, E, B, F, C, G
5	A, E, F, B, C, G
6	A, B, E, F, G, C
7	A, B, E, F, C, G

Table 2. Corresponding event log

	E	F	G
A	\rightarrow	+	+
B			
C	+	\leftarrow	

Table 3. Communication behavioral profile

the corresponding allowed sequences in Tbl. 4. We have $A \ll E$, which is indeed the communication as modeled in the composition $M \oplus N$. However, we also find $G \ll B$, indicating a possible communication between G and B . Listing all communication successors, we get $A \ll E$, $G \ll B$, $F \ll B$, $B \ll G$, $G \ll C$, $E \ll B$, $B \ll F$, $F \ll C$, $C \ll G$, and $B \ll E$. Observe that because of the asynchronous nature of the communication, features B and E are concurrently enabled in Fig. 3. Assuming the event log to be complete, this should become visible in the communication successor relation, as for the normal successor relation on event logs.

Definition 7 (Communication behavioral profile). Let $\mathcal{L} \subseteq T^*$ be an event log, and $\ll_{\mathcal{L}} \subseteq T \times T$ the corresponding communication successor relation.

The communication behavioral profile is the 3-tuple $(\rightarrow_c, ||_c, +_c)_{\mathcal{L}}^{Com}$ defined by:

- $A \rightarrow_c B$ iff $A \ll_{\mathcal{L}} B$ and $B \not\ll_{\mathcal{L}} A$;
- $A ||_c B$ iff both $A \ll_{\mathcal{L}} B$ and $B \ll_{\mathcal{L}} A$; and
- $A +_c B$ iff both $A \not\ll_{\mathcal{L}} B$ and $A \not\ll_{\mathcal{L}} B$.

Calculating the behavioral profile of the communicating transitions using the communication successor relation, results in the communication behavioral profile as shown in Tbl. 3. It shows that B and E are concurrently enabled. Following the behavioral profile, we see that the causal relation of the behavioral profile correctly identifies the feature communication.

Using the communication behavioral profile, we can construct the information flows from an event log as follows. If $A \rightarrow B$ in the communication behavioral

	Debtor							Payment							Creditor	
	A	B	C	D	E	F	G	H	I	J	K	O	M	N	Q	S
A								←	+	+	+	+	+	+	+	+
B								+	→	+	+	+	+	+	+	+
C								+	+	→	+	+	+	+	+	+
D								+	+	+	←	+	+		+	+
E								+	+	+	+	←	+	+	+	+
F								+	+	+	+	+	→	+	+	+
G								+		+	+	+			+	
H	→	+	+	+	+	+	+								+	+
I	+	←	+	+	+	+									+	+
J	+	+	←	+	+	+	+								→	+
K	+	+	+	→	+	+	+								+	←
O	+	+	+	+	→	+	+								+	←
M	+	+	+	+	+	←									+	→
N	+	+	+		+	+									+	
Q	+	+	+	+	+	+	+	+	+	←	+	+	+	+		
S	+	+	+	+	+	+		+	+	+	→	→	←			

Table 4. Communication behavioral profile for the running example

profile of the event log, then an information flow (A, x, B) exists, with x a fresh label. This results in the following translation:

Definition 8 (Generated FAM). Let \mathcal{L} be an event log, and $(\rightarrow_c, \parallel_c, +_c)_{\mathcal{L}}^{Com}$ be its communication behavioral profile. Its corresponding functional architecture model $\langle \mathcal{M}, \mathcal{C}, \mathcal{F}, h, m, \rightarrow \rangle$ is defined by:

- $\mathcal{M} = \mathfrak{R}(\mathcal{L})$;
- $\mathcal{C} = \emptyset$;
- $\mathcal{F} = T$;
- $h = \emptyset$;
- $m = \mathfrak{R}$; and
- $\rightarrow = \{(A, x, B) \mid A \rightarrow_c B, \text{ and } x \in A \text{ a fresh label}\}$.

After constructing the communication behavioral profile for the running example, shown in Tbl. 4, we can complete the functional architecture model. Based on the given system execution data, we see for example that feature H communicates with feature A , and feature S sends messages to features K and O , and receives messages from feature M . The complete functional architecture of the running example is shown in Fig. 4.

5.2 Internal Behavior of Features

As can be seen in the running example, features can send and receive multiple messages. For example, feature S sometimes sends a message to feature K and sometimes to feature O . Therefore, the next step in discovering the functional

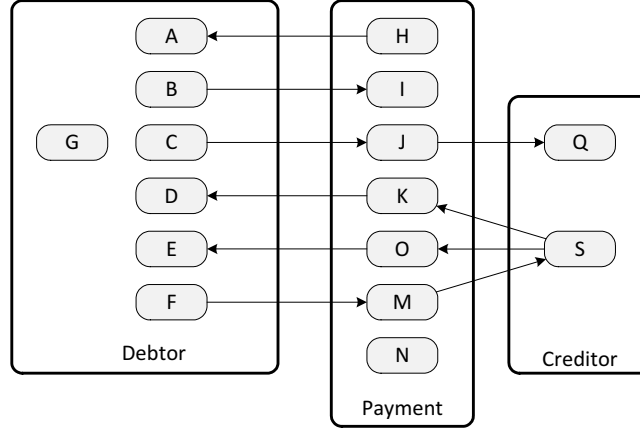


Fig. 4. Functional architecture model of the running example

architecture is to reconstruct the internal behavior of each of the features. For this, we create for each of the features an event log, containing the features that it communicates with. We call this the *feature log*.

Definition 9 (Feature log). Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \parallel_c, +_c)_{\mathcal{L}}^{Com}$ be the corresponding communication behavioral profile. The feature log \mathcal{L}_F is defined by $\mathcal{L}_F = \{\sigma|_{C(F)} \mid \sigma \in \mathcal{L}, F \in \sigma\}$ where $C(F) = \{A \mid A \rightarrow_c F \vee F \rightarrow_c A\}$.

Consider for example feature S in the running example. This feature communicates with features K , O and M , i.e., $C(S) = \{K, O, M\}$. Its feature log is the projection of the log on these features, i.e., $\mathcal{L}_S = \{\langle K \rangle, \langle O, M \rangle\}$.

On these feature logs, we apply the inductive miner [13], that always returns a sound workflow net. Next, we transform the discovered workflow net into an open Petri net, to visualize the messages sent and received by the feature. This results in a *feature net* for each of the features present in the event log.

Definition 10 (Feature Net). Let $\mathcal{L} \subseteq T^*$ be an event log, and let $F \in T$ be some feature. Let $(\rightarrow_c, \parallel_c, +_c)_{\mathcal{L}}^{Com}$ be the corresponding communication behavioral profile. The Feature net \mathcal{N}_F is the OPN $\langle P, I, O, T, F, i, \Omega \rangle$ defined by

- $P = \bar{P}$, $T = \bar{T}$, $i = [\bar{i}]$, $\Omega = \{[\bar{f}]\}$;
- $I = \{p_{A-F} \mid A \rightarrow_c F\}$;
- $O = \{p_{F-A} \mid F \rightarrow_c A\}$;
- $F = \bar{F} \cup \{(t, p_{F-A}) \mid t \in T, \lambda(t) = A, F \rightarrow_c A\} \cup \{(p_{A-F}, t) \mid t \in T, \lambda(t) = A, A \rightarrow_c F\}$.

where $\langle \bar{P}, \bar{T}, \bar{F}, \bar{i}, \bar{f} \rangle$ is the discovered workflow net.

In our running example, each of the 16 features are transformed into a feature net. Most of the features are simple, like for feature H and A , consisting of

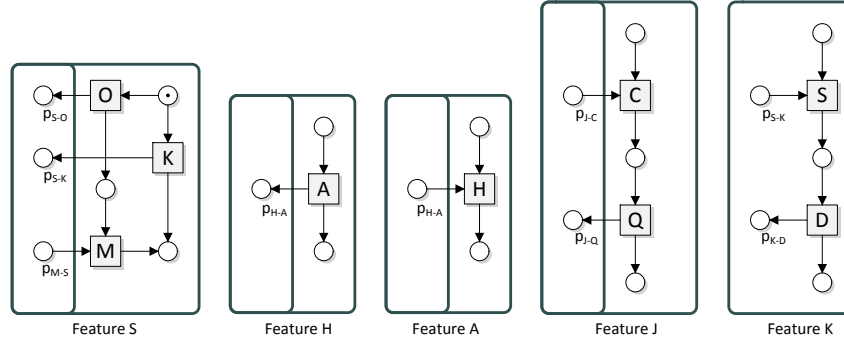


Fig. 5. Some of the feature nets for the running example

a single transition sending a message to A , and receiving a message from H , respectively. A more complex feature net is the net for feature S , which internally decides whether it sends a message to K or to O . Figure 5 depicts some of the feature nets generated using the inductive miner [13].

5.3 Feature Interaction within Modules

Now that each feature has its internal behavior defined by means of a feature net, the next step is to determine the order in which features are executed within each of the modules. As for the features, we first create event logs for each of the modules, by filtering each trace on the features it contains. This results in a *module log* for each of the modules.

Definition 11 (Module Log). Let $\mathcal{L} \subseteq T^*$ be an event log. Let $M \in \text{Rng}(\mathfrak{R})$ be a module. Let $(\rightarrow_c, \parallel_c, +_c)$ be the corresponding communication behavioral profile. The Module log \mathcal{L}_M is defined by $\mathcal{L}_M = \{\sigma \mid \{F \mid \mathfrak{R}(F) = M\} \mid \sigma \in \mathcal{L}\}$.

Within the running example, we obtain three module logs, one for each of the modules. For example, module *Debtor*, has module log $\mathcal{L}_{\text{Debtor}} = \{\langle A, B, G \rangle, \langle A, C, D, G \rangle, \langle A, C, E, F, G \rangle\}$, and for *Creditor* we have $\mathcal{L}_{\text{Creditor}} = \{\langle Q, S \rangle,$

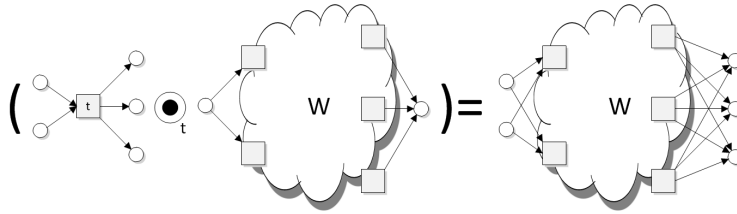


Fig. 6. Refinement of a transition by a workflow net

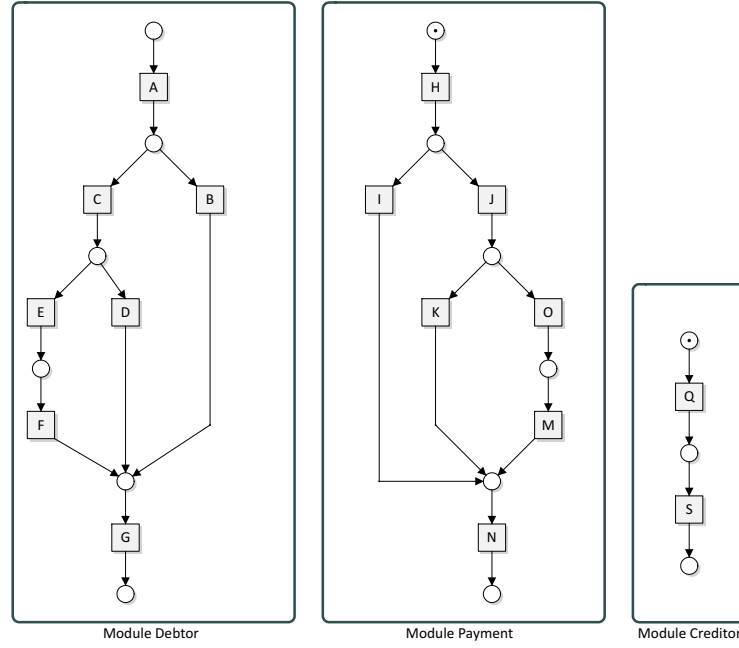


Fig. 7. Module nets for the running example

$\langle Q, S, S \rangle$. Again applying the inductive miner results in the three workflow nets as depicted in Fig. 7. Notice that, although feature S occurs twice in one of the sequences, the algorithm only adds a single feature S in the resulting workflow model.

5.4 Composition of Feature Nets and Module Nets

Last step in the process is to combine the feature nets generated for each of the features with the generated module nets. This results in an open Petri net for each of the modules, defining the interaction between the different modules.

In the module net, each feature is represented by a single transition. Next step is to refine each feature by its feature net. For this, we first define the refinement of a transition by a workflow model on open Petri nets, as shown in Fig. 6. This refinement connect each input place of the refined transition with each of the transitions in the postset of the initial place of the workflow, and similarly each output place of the refined transition with each of the transitions in the preset of the final place of the refining workflow. It is straight-forward to prove that if (1) the initial net is sound, (2) each input place of the refined transition is 1-bounded, i.e., it can contain at most one token, and (3) workflow net W is sound, then the refinement yields a sound result.

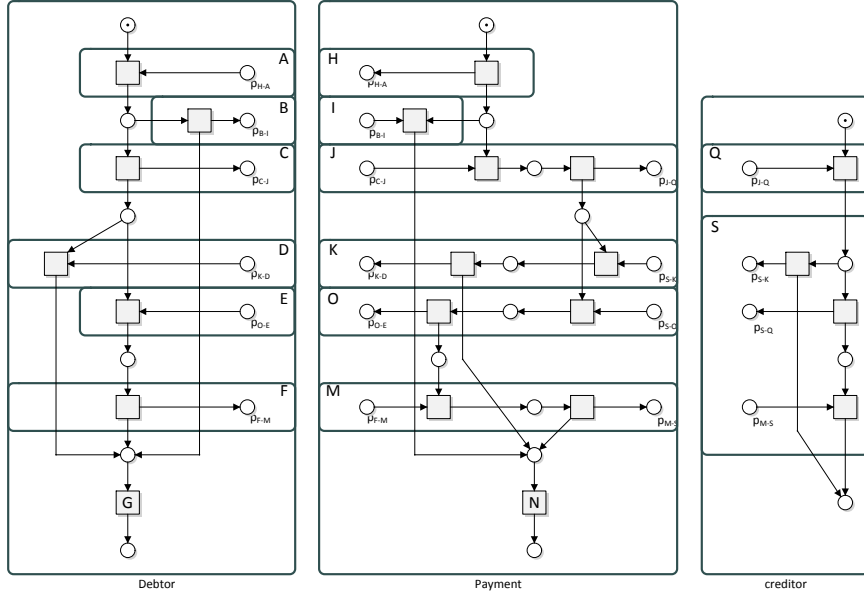


Fig. 8. Composed nets generated for the running example

The result of refining each feature by its feature net is shown in Fig. 8. As features G and N have no feature net defining communication, these transitions are not refined.

To verify whether the resulting open Petri nets are a true representation of the system, one can compose the nets into a single Petri net, and execute each of the sequences of the event log of Tbl. 1 on the resulting model, which in this example is possible. Further analyzing the resulting model shows that its only deadlocks are desirable markings: either all modules reach their final place, without any pending tokens, or the *Creditor* module remains untouched, while the *Debtor* and *Payment* module reach their final place.

6 Conclusions

Within this paper, we discussed a method to automatically generate a functional architecture model from an event log together with a mapping of each feature to the module that offers that functionality. We showed how the information flows can be derived from the communication behavioral profile. This profile not only identifies the information flow for the static structure of the functional architecture, but additionally offers sufficient information to construct the internal behavior for each of the features, and between the features within a module. Lastly, we showed how to compose feature and module nets into an open Petri net.

Discovering the interaction between different modules is not new. Techniques like service mining [3], apply process mining on event logs to discover a process model of how the services are orchestrated. In the approach presented in this paper, we focus on the discovery of the behavior of each of the modules, rather than a complete orchestration.

In [15], the authors discover the internal behavior of services based on the interaction between two services, guaranteeing deadlock freedom of the discovered service. In the setting of this paper, the exact interaction between modules is unknown, and needs to be discovered first.

The core idea of this paper is twofold: firstly to derive the information flows for a Functional Architecture Model, and secondly to derive the internal behavior for each of the modules within the architecture. Within software architecture, this is called Software Architecture Reconstruction [12]. Although some techniques take the dynamic aspects of the software operation into account, most techniques only focus on the static aspects of software architecture models, using solely the available source code [8]. For example, system execution data is used to enrich architectures with performance data [11] or to visualize traces on how the software is used [19]. In this paper, we propose a method to not only visualize software usage, but to discover module communication and to generate the internal behavior of modules within a software architecture.

Although the approach presented in this paper shows an application of the behavioral profile to discover feature interaction, additional research is required. First, the current approach requires the event log to be complete, i.e., if the log grows, the successor relation should not change. Further, for the generation of the internal feature behavior, we assume that if the sending feature is present in the event log, it enables all possible events, which is possibly a too strict assumption that deserves further investigation.

The approach in this paper is very flexible, as we derive individual models for the features and modules. For this, we apply standard process discovery algorithms returning sound workflow models. However, their composition in general does not result in a sound system of asynchronously communicating systems. Further research is required to study the conditions under which this can be guaranteed. For this, we want to identify conditions which on the one hand result in correct models, and on the other hand have a positive effect on model quality as described by [7].

Not only does this approach provide useful insights for the software architect, we expect the approach applicable to business process management as well, as for the discovery of separate business processes, the Business Process Modelling and Notation offers the swimlane notion. Therefore, we plan to implement the approach in the Process Mining toolkit ProM [18] to experiment and apply the approach on real-life examples.

Acknowledgements The authors would like to thank the anonymous reviewers for their valuable feedback, and Sjaak Brinkkemper, Fabiano Dalpiaz, Garm Lucassen, Leo Pruijt and Erik Jagroep for the fruitful discussions and valuable input on architecture and software products.

References

1. W.M.P. van der Aalst. Verification of workflow nets. In *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407 – 426. Springer, Berlin, 1997.
2. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Berlin, 2011.
3. W.M.P. van der Aalst. Challenges in service mining: Record, check, discover. In *Web Engineering*, volume 7977 of *Lecture Notes in Computer Science*, pages 1–4. Springer, Berlin, 2013.
4. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Series in Software Engineering. Addison Wesley, Reading, MA, USA, 2012.
5. D. Bera, K. M. van Hee, and J.M.E.M. van der Werf. Designing weakly terminating ros systems. In *Applications and Theory of Petri Nets, (33th International Conference, Petri Nets 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 328 – 347. Springer, Berlin, 2012.
6. S. Brinkkemper and S. Pachidi. Functional architecture modeling for the software product industry. In *ECSCA 2010*, volume 6285 of *Lecture Notes in Computer Science*, pages 198 – 213. Springer, Berlin, 2010.
7. J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. On the role of fitness, precision, generalization and simplicity in process discovery. In *On the Move to Meaningful Internet Systems: OTM 2012*, volume 7565 of *Lecture Notes in Computer Science*, pages 305–322. Springer, Berlin, 2012.
8. S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *Software Engineering, IEEE Transactions on*, 35(4):573–591, July 2009.
9. S.A. Fricker. Software product management. In *Software for People*, Management for Professionals, pages 53–81. Springer, 2012.
10. K.M. van Hee, N. Sidorova, and J.M.E.M. van der Werf. When can we trust a third party? In *Transactions on Petri Nets and Other Models of Concurrency VIII*, volume 8100 of *Lecture Notes in Computer Science*, pages 106–122. Springer, Berlin, 2013.
11. T. Israr, M. Woodside, and G. Franks. Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems and Software*, 80(4):474 – 492, 2007. Software Performance 5th International Workshop on Software and Performance.
12. R.L. Krikhaar. *Software Architecture Reconstruction*. PhD thesis, VU Amsterdam, 1999.
13. S.J.J. Leemans, D. Fahland, and W.M.P. van der Aalst. Discovering block-structured process models from event logs - a constructive approach. In *Application and Theory of Petri Nets and Concurrency*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer, Berlin, 2013.
14. G. Lucassen, J.M.E.M. van der Werf, and S. Brinkkemper. Alignment of software product management and software architecture with discussion models. In *IWSPM 2014*, pages 21–30. IEEE, Aug 2014.
15. R. Müller, C. Stahl, W.M.P. van der Aalst, and M Westergaard. Service discovery from observed behavior while guaranteeing deadlock freedom in collaborations. In *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science*, pages 358–373. Springer, Berlin, 2013.
16. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *Monographs in Theoretical Computer Science: An EATCS Series*. Springer, Berlin, 1985.

17. R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2010.
18. H.M.W. Verbeek, J.C.A.M. Buijs, B.F. van Dongen, and W.M.P. van der Aalst. XES, XESame, and ProM 6. In *Information System Evolution*, volume 72 of *Lecture Notes in Business Information Processing*, pages 60–75. Springer, Berlin, 2011.
19. R.J. Walker, G.C. Murphy, J. Steinbok, and M.P. Robillard. Efficient mapping of software system traces to architectural views. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '00, pages 12–. IBM Press, 2000.
20. M. Weidlich and J.M.E.M. van der Werf. On profiles and footprints – relational semantics for petri nets. In *Applications and Theory of Petri Nets (ICATPN 2012)*, volume 7347 of *Lecture Notes in Computer Science*, pages 148 – 167. Springer, Berlin, 2012.
21. J.M.E.M. van der Werf and H.M.W. Verbeek. Online compliance monitoring of service landscapes. In *BPM 2014 International Workshops*, volume 202 of *Lecture Notes in Business Information Processing*, pages 89–95. Springer, Berlin, 2015.

Interval-Timed Petri Nets with Auto-concurrent Semantics and their State Equation

Elisabeth Pelz ¹, Abderraouf Kabouche ¹, Louchka Popova-Zeugmann ²

¹ LACL, Université Paris-Est Créteil, France

² Department of Computer Science, Humboldt University Berlin, Germany

Abstract. In this paper we consider Interval-Timed Petri nets (ITPN), an extension of Timed Petri nets in which the discrete time delays of transitions are allowed to vary within fixed intervals including possible zero durations. These nets will be analyzed for the first time under some maximal step semantics with auto-concurrency. This matches well with the reality of time critical systems which could be modeled and analyzed with our model. We introduce in particular the notion of global firing step which regroups all what happens inbetween two time ticks. Full algebraic representations of the semantics are proposed. We introduce time-dependent state equations for a sequence of global firing steps of ITPNs which are analogous to the state equation for a firing sequence in standard Petri nets and we prove its correctness using linear algebra. Our result delivers a necessary condition for reachability which is also a sufficient condition for non-reachability of an arbitrary marking in an ITPN.

1 Introduction

Petri nets (PN) as proposed initially by Carl Adam Petri [4] are applied to design models of systems considering only causal relations in it and not temporal ones. Of course there is a huge field of applications in which time does not really matter. In real systems, however, the time is mostly indispensable and therefore it cannot be ignored. Thus a certain number of time-dependent Petri net classes had been proposed in the meanwhile, cf. ([3], [9], [5], [2], [11], [1], [6]). Moreover, it is well known that the majority of these classes are more expressive than the classic model: Almost all time-dependent Petri net classes are Turing-powerful, while the power of classic Petri nets is less than that of Turing-machines.

In this paper we are dealing with Interval-Timed Petri nets (ITPN), which are an extension of Timed Petri nets (TPN), introduced by Ramhandani in [9] and extensively studied by Sifakis [10]. TPNs are classic PNs where each transition is associated with a natural number which describes its firing duration. TPNs, as well as their extensions like ITPNs, are Turing-powerful (cf. Popova [6]).

In ITPNs the firing duration of a transition is also given by a natural number but this duration is not fixed. It may vary within an interval which is associated with the transition. The apparition of a transition is thus divided in two events, the startfire and the endfire event. Inbetween them tick events may happen,

corresponding to the passing (or elapsing) of one time unit of some global clock [1].

When transitions are enabled they must start firing. This is the reason why we consider as firing modus for ITPNs the firing in maximal steps. Two different step semantics are possible: with or without auto-concurrency. In this article, we consider ITPNs with auto-concurrency. This means that when a transition becomes enabled, irrespective of whether or not an instance of it is firing already, a new instance must immediately start firing. The firing duration of each new instance is chosen in a non-deterministic way and is a natural number, describing how many tick events may occur before the endfiring event. This number belongs to the interval associated with the transition. Contrary to previous work, zero firing durations are allowed in this article.

A configuration in a PN is described by a marking. Because of the explicit presence of time a marking alone cannot completely represent the configuration of a time-dependent Petri net however. For this reason we use the notion of “state” which includes both the marking and the corresponding temporal informations. The first aim of the paper is to introduce the maximal step semantics for the ITPNs formally: a firing step sequence in an ITPN consists of alternating so called *Globalsteps* (multisets of startfire and endfire events) and tick events. And we will prove some semantical properties.

The second aim of this paper is to provide a sufficient condition for non-reachability of states in ITPNs similar to the sufficient condition for non-reachability of markings for classic Petri nets. To illustrate this purpose, let us consider first the problem in a classic Petri net \mathcal{N} , starting with a firing sequence σ of \mathcal{N} . After the firing of such a sequence a certain marking M of \mathcal{N} is reached. We can compute this marking using the following well known equation:

$$M = M_0 + C \cdot \psi_\sigma \quad (1)$$

where C is the incidence matrix of the Petri net \mathcal{N} and ψ_σ is the Parikh vector of σ (whose i -th component gives the number of appearance of transition t_i in σ). This equation is also called the *state equation of the sequence* σ . Actually, it can be used in many more ways. We can consider each marking suitable for a net as reached after the firing of an unknown sequence. Now, we can consider the state equation of the unknown sequence, where the elements of the Parikh vector are variables. If this equality has no non-negative integer solution then there does not exist a sequence making the considered marking reachable. Therefore, this is a sufficient condition for the non-reachability of the marking. The following simple example illustrates this approach:

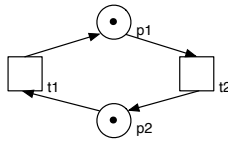


Fig. 1: PN \mathcal{N}_1 .

Let us consider the PN \mathcal{N}_1 with $M_0 = (1, 1)^T$ and show that the empty marking $M = (0, 0)^T$ is not reachable in this net. The incidence matrix of \mathcal{N}_1 is $C_{\mathcal{N}_1} = \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$. Let us assume that there is a transition sequence σ such that after its firing in \mathcal{N}_1 the empty marking is reached. When the transition $t1$ appears x_1 times in σ and $t2$ appears x_2 times then the Parikh vector of σ is $\psi_\sigma = (x_1, x_2)^T$. Subsequently, the equality (1) for this

transition sequence leads to the system of equations $\begin{cases} -1 = x_1 - x_2 \\ -1 = -x_1 + x_2 \end{cases}$. This equation system is obviously not solvable and therefore there is no such firing transition sequence σ in \mathcal{N}_1 leading to the empty marking M .

Furthermore, it is evident that the marking $M' = (2, 0)^T$ is reachable in \mathcal{N}_1 .

Let us now consider the Interval-Timed Petri net \mathcal{D}_1 arising from the PN \mathcal{N}_1 by adding time durations to each transition – the firing of each transition should take exactly one time unit, thus $[1, 1]$ is the duration interval associated to t_1 and t_2 . As both transitions are fireable from the initial state, after startfiring both transitions in one step, the empty marking M is reached. After one tick event, both transitions need to endfire in one step, and the initial state is reached again. Thus it is easy to see that in this ITPN \mathcal{D}_1 the marking M' is not reachable. This simple example shows that reachability and non-reachability in an Interval-Timed Petri net are essentially unrelated to reachability and non-reachability in its untimed skeleton. Our aim is to prove with the help of a time-dependent state equation that for instance, it is impossible to reach M' in \mathcal{D}_1 .

Of course, the time-dependent state equations we are establishing in this paper are much more complex than (1) or our previous results in [8], [7] and [2] because of the possibility of zero durations and the auto concurrent maximal step semantics. Nevertheless, our equations of a firing step sequence in an ITPN are consistent extensions of (1).

The paper is organized as follows: First formal definitions of ITPNs and their maximal step semantics are given in Section 2, and some semantical equivalence is proved. Then original algebraic representations and calculus of these semantics are proposed in Section 3. Some of them are adaptations of definitions known for the algebraic presentation of a firing step sequence for TPN [8], or ITPN without zero duration and without auto-concurrency [7], and others are entirely new here. Within this frame intermediate algebraic properties are first established in Section 4, leading then to the state equations. Full proofs of all results are included in the paper.

2 Interval-Timed Petri Nets and their semantics

This section will define the objects treated in this article.

As usual, \mathbb{N} denotes the set of all natural numbers including zero, \mathbb{N}^+ is that without zero. A matrix A is a $(m \times n)$ - matrix when A has m rows and n columns. The denotation $A = \left(a_{ij} \right)_{\substack{i=1 \dots m \\ j=1 \dots n}}$ for a matrix A means that A is a $(m \times n)$ - matrix and a_{ij} is the element of A in the (i) -th row and in the j -th column. Furthermore, $A_{.j} = (a_{.j})$ denotes the j -th column of the matrix A and $A_{i.} = (a_{i.})$ denotes the i -th row. The $(d \times d)$ - matrix O_d denotes the $(d \times d)$ zero-matrix (all its elements are zero), the $(d \times d)$ - matrix E_d is the $(d \times d)$ identity matrix.

2.1 Net definitions

A (marked) *Petri net* (PN) is a quadruple $\mathcal{N} = (P, T, v, M_0)$, where P (the set of places) and T (the set of transitions) are finite and disjoint sets and

$v : (P \times T) \cup (T \times P) \longrightarrow \mathbb{N}$ defines the arcs with their weights and $M_0 : P \longrightarrow \mathbb{N}$ fixes the initial p -marking. In general, a p -marking $M : P \longrightarrow \mathbb{N}$ is presented by a vector of dimension $|P|$. As usual, t is called enabled in a p -marking M if for all $p \in P$, $v(p, t) \leq M(p)$.

Let \mathcal{N} be a PN and $D : T \longrightarrow \mathbb{N} \times \mathbb{N}$ be a function. Then, a pair $\mathcal{Z} = (\mathcal{N}, D)$ is called an *Interval-Timed Petri net* (ITPN) where \mathcal{N} is its *skeleton* and D its *duration function* including zero duration. Thus, D defines an interval for each transition, within which its firing duration can vary.

The bounds $sfd(t)$ and $lfd(t)$ with $D(t) = (sfd(t), lfd(t))$ are called the *shortest firing duration* for t and the *longest firing duration* for t , respectively. Furthermore, each $\delta_i \in (D(t_i) \cap \mathbb{N})$ can be the actual duration of transition t_i firing. The bounds are allowed to be zero, i.e. the firing can be considered to take no time. An ITPN behaves similarly to a PN with regards to maximal step semantics. In this article auto-concurrency is not only allowed, but forced. Thus a *maximal step* will be a *multiset* of events which appears at the same moment.

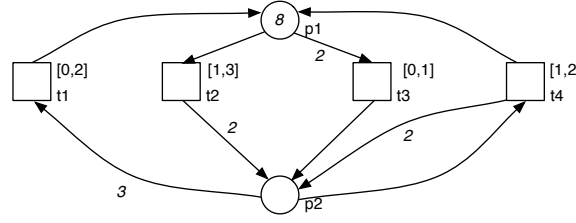
Formally, a *multiset* U of events E is a total function $U : E \longrightarrow \mathbb{N}$, where $U(e_i)$ defines the number of occurrences of the event e_i in the multiset U . We can write U in the extended set notation $U = \{e^{U(e)} \mid e \in E \text{ and } U(e) \neq 0\}$ and we denote by \uplus the operator of multisets union.

Let \bar{t} be a transitions sequence of length n , $\bar{t} = t_1 t_2 \dots t_n$. The transitions sequence \bar{t} is called an *undesired cycle* if, for all $i \leq n$, $sfd(t_i) = 0$ and for all p , $\sum_{1 \leq i \leq n} (v(t_i, p) - v(p, t_i)) \geq 0$. Thus *undesired cycles* have firing duration zero and could be infinitely repeated without time elapsing.

An ITPN is *well formed* if it has no undesired cycles. In order to avoid infinite steps only *well formed* nets are considered in this paper.

Note that a token will reach the post-set of a transition t_i only after the time corresponding to the actual duration of this transition has elapsed. The exact value of the actual duration δ_i is unknown at the beginning of the firing of t_i . The transition may stop firing after an arbitrary number $\delta_i \in D(t_i)$ of time ticks has elapsed.

As usual in time-dependent PNs, *states* in ITPNs are pairs $S = (M, h)$ of mappings, M being the p -marking and h codes the clocks of the transitions. In [7] h was defined as *clock-vector*, whereas now, in the context of auto-concurrency, h needs to be a matrix of dimension $(|T| \times d)$. Thus the *clock-matrix* h has $|T|$ rows (i.e. the number of transitions in the skeleton \mathcal{Z}) and $d = \max_{t_i \in T} (lfd(t_i)) + 1$ columns. The value $h_{i,j+1}$ represents the number of active transitions t_i with age j (i.e. fired since j time ticks), where $j \in D(t_i)$. Please, note that we need to use 'j+1' because the first column of the matrix has number 1 and not number 0. The *initial state* $S^{(0)} = (M^{(0)}, h^{(0)})$ of \mathcal{Z} is given by the initial marking $M^{(0)} = M_0$ of \mathcal{Z} and the *zero-clock-matrix* $h^{(0)}$ where $h_{i,j}^{(0)} = 0$ for all i, j . The ITPN \mathcal{Z}_o which is used as a running example is shown in Fig.2.

Fig. 2: ITPN \mathcal{Z}_0 .

2.2 Semantics of Interval-Timed Petri Nets

Now, the behavior of ITPNs will be defined. For the *transition rule* of an ITPN we distinguish three types of events, namely

- *Startfire events*: A startfire event, denoted as $[t_i]$, *must* occur immediately (even n times) if t_i becomes enabled in the skeleton (resp. if n transitions t_i become enabled at the same time). For each occurrence of $[t_i]$ the input tokens of t_i are removed from their preplaces, the clock associated with t_i will count this occurrence by incrementing the number $h_{i,1}$ and t_i will be called *active*.
- *Endfire events*: An endfire event, denoted as $t_i]$, *must* occur (even n times) if the clock associated with t_i is expiring, i.e. $h_{i,j+1} = n \neq 0$ and $j = lfd(t_i)$. The event $t_i]$ *may* occur (at most q_i times) if $\sum_{sfd(t_i) \leq j < lfd(t_i)} h_{i,j+1} = q_i \geq 1$.

For each of the endfire events $t_i]$ which occurs the corresponding $h_{i,j+1}$ is decremented and the output tokens are delivered at the postplaces of t_i . There is not only some choice, if some active transitions which need not to endfire may endfire. But once the number of these may endfire events is fixed (for instance $q \leq q_i$ times transition t_i), there is a choice to take these q events totally nondeterministically or to take deterministically those q which are the oldest among the q_i active ones.

- *Tick events*: A tick event, denoted as \checkmark , is enabled iff there is no firing event which must either start firing or stop firing. Upon occurring, a tick event increments the clocks for all active transitions. Hence the tick events are global. More precisely the incrementation is realised with a right shift of the clock-matrix and by setting the first column to zero.

The initial state is considered to be the first *after-tick state*. The whole set of such states is defined by induction in the sequel. An ITPN can change from one after-tick state into another one by the occurrence of the so-called *Globalstep*, which due to zero duration and auto concurrency extends the definition of firing triple known from [7]. A *Globalstep* consists of several parts, first a multiset of endfire events (called *Endstep*), then an *iterative union* of two multisets *Maxstep* and *EndstepZero*, (called *Iteratedstep*). A *Maxstep* is a maximal step of startfire events and an *EndstepZero* is a multiset of endfire events of transitions with zero firing duration. The iteration stops when no further *Maxstep* is possible. Note

that it always stops as only wellformed ITPNs are considered. The *Globalstep* is followed by one tick event for time elapsing.

During the execution of the ITPN *Globalsteps* and single tick events alternate in the following way. Let $S^{(1)} = (M^{(1)}, h^{(1)})$ be an arbitrary after-tick state of \mathcal{Z} .

1) An *Endstep* (for *end-firing-step*), denoted by $\mathfrak{G}^{(1)}$, represents the union of two multisets: That of all active transitions T_1 which must end their firing in this state, and a multiset T'_2 that contains several transitions which may end their firing in this state s .

Thus *Endstep* $\mathfrak{G}^{(1)} = T_1 \uplus T'_2$ where $T'_2 \subseteq T_2$,

$T_1 = \{\mathfrak{t}_i^{n_i} \mid i \in [1, |T|], h_{i,j+1}^{(1)} = n_i \neq 0, j = lfd(t_i)\}$ and

$T_2 = \{\mathfrak{t}_i^{q_i} \mid i \in [1, |T|], q_i = \sum_{sfd(t_i) \leq j < lfd(t_i)} h_{i,j+1}^{(1)}\}$.

Without loss of generality, we can choose for each i to put in T'_2 the *oldest* active transitions $t_i \in T_2$, as shown later in Theorem 3.

Its occurrence $S^{(1)} \xrightarrow{\mathfrak{G}^{(1)}} \tilde{S}^{(1)}$ leads to $\tilde{S}^{(1)} = (\tilde{M}^{(1)}, \tilde{h}^{(1)})$ such that

$$\forall p \in P \quad \tilde{M}^{(1)}(p) = M^{(1)}(p) + \sum_{t_i \in \mathfrak{G}^{(1)}} \mathfrak{G}^{(1)}(\mathfrak{t}_i) \cdot v(t_i, p) \quad (2)$$

$$\text{and } \tilde{h}_{i,j}^{(1)} := \begin{cases} 0 & \text{if } \mathfrak{G}^{(1)}(\mathfrak{t}_i) - \sum_{j' \geq j} h_{i,j'}^{(1)} \geq 0 \\ h_{i,j}^{(1)} - q & \text{if } \mathfrak{G}^{(1)}(\mathfrak{t}_i) - \sum_{j' \geq j+1} h_{i,j'}^{(1)} = q \text{ and } 0 < q < h_{i,j}^{(1)} \\ h_{i,j}^{(1)} & \text{otherwise.} \end{cases} \quad (3)$$

The state $\tilde{S}^{(1)}$ is called an *intermediate state*.

2) An *Iteratedstep* is the iterative union of two multisets, the first one being a *Maxstep*. The second one contains only *Endfiring* events of transitions with zero duration, we denote that as *EndstepZero*.

We start by setting $k := 0$ and

$$\tilde{M}^{(1,k)} = \tilde{M}^{(1,0)} := \tilde{M}^{(1)} \text{ and } \tilde{h}^{(1,k)} = \tilde{h}^{(1,0)} := \tilde{h}^{(1)}. \quad (4)$$

a) A *Maxstep* (for maximal start firing step) represents a maximal multiset of concurrently enabled transitions which must start to fire after an *Endstep* or an *EndstepZero*. The multiset of startfire events is denoted by $\mathfrak{G}_M^{(1,k+1)} =$

$\{\mathfrak{t}_i^{n_i} \mid i \in [1, |T|] \text{ and } \tilde{M}^{(1,k)} \geq \sum_{i=1}^{|T|} n_i \cdot v(t_i, p)\}$.

If there are several enabled *Maxsteps*, the choice will be arbitrary solved.

The iterative union is stopped if the calculated $k + 1$ -th *Maxstep* is empty ($\mathfrak{G}_M^{(1,k+1)} = \emptyset$, i.e. a fixpoint is reached). This implies that no further transitions can fire in this step, which always arrives because of the wellformedness of the net. The value of k is stocked in k_{max} ($k_{max} := k$).

b) An *EndstepZero*, denoted by $\mathfrak{G}_Z^{(1,k+1)}$, is a multiset of endfire events of just activated transitions, which *must* or *may* end their firing immediately.

Precisely, *EndstepZero* contains only transitions started in the same step of iteration and whose *shortest firing duration* is equal to zero; all of them whose longest firing duration is equal to zero too must end their firing; among the others an arbitrary number of transitions may end their firing. Thus *EndstepZero* is defined as

$$\mathfrak{G}_Z^{(1,k+1)} = \left\{ \mathfrak{t}_i^{n_i} \mid \begin{array}{l} i \in [1, |T|] \text{ and } sfd(t_i) = 0 \text{ and } \left[(lfd(t_i) = 0 \text{ and } \right. \\ \left. n_i = \mathfrak{G}_M^{(1,k+1)}([\mathfrak{t}_i]) \text{ or } (lfd(t_i) \neq 0 \text{ and } \right. \\ \left. n_i \leq \mathfrak{G}_M^{(1,k+1)}([\mathfrak{t}_i]) \right] \end{array} \right\}.$$

A state $\tilde{S}^{(1,k+1)}$ is calculated after the k -th iteration such that for each $p \in P$ it holds that:

$$\begin{aligned} \tilde{M}^{(1,k+1)}(p) &= \tilde{M}^{(1,k)}(p) - \sum_{t_i \in T} \mathfrak{G}_M^{(1,k+1)}([\mathfrak{t}_i]) \cdot v(p, t_i) + \sum \mathfrak{G}_Z^{(1,k+1)}(\mathfrak{t}_i) \cdot v(t_i, p) \quad \text{and} \quad (5) \\ \tilde{h}_{i,j}^{(1,k+1)} &:= \begin{cases} \tilde{h}_{i,j}^{(1,k)} + \mathfrak{G}_M^{(1,k+1)}([\mathfrak{t}_i]) - \mathfrak{G}_Z^{(1,k+1)}(\mathfrak{t}_i) & \text{if } j = 1 \\ \tilde{h}_{i,j}^{(1,k)} & \text{otherwise} \end{cases}. \quad (6) \end{aligned}$$

All newly fired and not ended events obtain age zero, i.e. are counted in column $j = 1$ of the clock-matrix.

The *Iteratedstep* is now defined by

$$\mathfrak{G}_I^{(1)} = \biguplus_{1 \leq k \leq k_{max}} (\mathfrak{G}_M^{(1,k)} \biguplus \mathfrak{G}_Z^{(1,k)}). \quad (7)$$

The occurrence of an *Iteratedstep* (\mathfrak{G}_I) $\tilde{S}^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} S'^{(1)}$ leads to $S'^{(1)} = (M'^{(1)}, h'^{(1)})$ with

$$M'^{(1)} := \tilde{M}^{(1,k_{max})} \quad \text{and} \quad h'^{(1)} := \tilde{h}^{(1,k_{max})}. \quad (8)$$

$S'^{(1)}$ is called an *intermediate state*.

3) After the *Globalstep* (\mathfrak{G}_γ , $\mathfrak{G}_I^{(l)}$), one *tick event* has to occur now in state S' , as no further firing event must happen. Its occurrence $S'^{(1)} \xrightarrow{\gamma} S^{(2)}$ leads to $S^{(2)} = (M^{(2)}, h^{(2)})$. The state $S^{(2)}$ is a new *after-tick state*, with

$$M^{(2)} := M'^{(1)} \quad \text{and} \quad h_{i,j}^{(2)} := \begin{cases} h_{i,j-1}'^{(1)} & \text{if } 1 < j \leq d \\ 0 & \text{if } j = 1 \end{cases} \quad (9)$$

4) A *firing step sequence* σ in an ITPN \mathcal{Z} is an alternating sequence of *Globalsteps* and ticks, starting with the initial time state $S^{(0)} = (M^{(0)}, h^{(0)})$

$$\begin{aligned} \sigma = S^{(0)} \xrightarrow{\mathfrak{G}_\gamma^{(0)} = \emptyset} \tilde{S}^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} S'^{(0)} \xrightarrow{\gamma} S^{(1)} \xrightarrow{\mathfrak{G}_\gamma^{(1)}} \tilde{S}^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} S'^{(1)} \xrightarrow{\gamma} S^{(2)} \xrightarrow{\mathfrak{G}_\gamma^{(2)}} \\ \tilde{S}^{(2)} \dots S^{(n-1)} \xrightarrow{\mathfrak{G}_\gamma^{(n-1)}} \tilde{S}^{(n-1)} \xrightarrow{\mathfrak{G}_I^{(n-1)}} S'^{(n-1)} \xrightarrow{\gamma} S^{(n)}. \end{aligned} \quad (10)$$

where for all $l \geq 0$, the *Endstep* $\mathfrak{G}_\gamma^{(l)}$, *Iteratedstep* $\mathfrak{G}_I^{(l)}$ and states $S^{(l)} = (M^{(l)}, h^{(l)})$, $S'^{(l)} = (M'^{(l)}, h'^{(l)})$ and $\tilde{S}^{(l)} = (\tilde{M}^{(l)}, \tilde{h}^{(l)})$ verify the above conditions. In particular each $S^{(l)}$ has the same marking, i.e. the same first column in the time marking as $S'^{(l-1)}$.

The following lemma states that the definition of $S'^{(1)}$ is well founded

Lemma 1 *Let us consider state $S'^{(l)} = (M'^{(l)}, h'^{(l)})$ as defined in (8). Then this state fulfils*

$$M'^{(l)} = \widetilde{M}^{(l)} - \sum_{i=1}^{|T|} \mathfrak{G}_I^{(l)}([\mathfrak{t}_i] \cdot v(p, t_i) + \sum_{i=1}^{|T|} \mathfrak{G}_I^{(l)}(\mathfrak{t}_i)) \cdot v(t_i, p) \quad \text{and}$$

$$h'_{i,j} = \begin{cases} \tilde{h}_{i,j}^{(l)} + [\mathfrak{G}_I^{(l)}([\mathfrak{t}_i] - \mathfrak{G}_I^{(l)}(\mathfrak{t}_i))] & \text{if } j = 1 \\ \tilde{h}_{i,j}^{(l)} & \text{otherwise.} \end{cases}$$

□

Proof. We start with

$$\begin{aligned} M'^{(l)} &\stackrel{(8)}{=} \widetilde{M}^{(l, k_{max})} \\ &\stackrel{(5)}{=} \widetilde{M}^{(l, k_{max}-1)} - \sum_{i=1}^{|T|} \mathfrak{G}_M^{(l, k_{max})}([\mathfrak{t}_i] \cdot v(p, t_i) + \sum_{i=1}^{|T|} \mathfrak{G}_Z^{(l, k_{max})}(\mathfrak{t}_i)) \cdot v(t_i, p) \end{aligned}$$

and after k_{max} iterations we obtain

$$\begin{aligned} M'^{(l)} &\stackrel{(5)}{=} \widetilde{M}^{(l, 0)} - \sum_{k=1}^{k_{max}} \sum_{i=1}^{|T|} \mathfrak{G}_M^{(l, k)}([\mathfrak{t}_i] \cdot v(p, t_i) + \sum_{k=1}^{k_{max}} \sum_{i=1}^{|T|} \mathfrak{G}_Z^{(l, k)}(\mathfrak{t}_i)) \cdot v(t_i, p) \\ &\stackrel{(4)+(7)}{=} \widetilde{M}^{(l)} - \sum_{i=1}^{|T|} \mathfrak{G}_I^{(l)}([\mathfrak{t}_i] \cdot v(p, t_i) + \sum_{i=1}^{|T|} \mathfrak{G}_I^{(l)}(\mathfrak{t}_i)) \cdot v(t_i, p). \end{aligned}$$

Further, we start with the definition of $h'^{(l)}$.

$$\begin{aligned} h'^{(l)} &\stackrel{(8)}{=} \tilde{h}^{(l, k_{max})} \\ &\stackrel{(6)}{=} \begin{cases} \tilde{h}_{i,j}^{(l, k_{max}-1)} + [\mathfrak{G}_M^{(l, k_{max})}([\mathfrak{t}_i] - \mathfrak{G}_Z^{(l, k_{max})}(\mathfrak{t}_i))] & \text{if } j = 1 \\ \tilde{h}_{i,j}^{(l, k_{max}-1)} & \text{otherwise.} \end{cases} \end{aligned}$$

and after k_{max} iterations we obtain

$$\begin{aligned} h'^{(l)} &\stackrel{(6)}{=} \begin{cases} \tilde{h}_{i,j}^{(l, 0)} + [\sum_{k=1}^{k_{max}} \mathfrak{G}_M^{(l, k)}([\mathfrak{t}_i] - \sum_{k=1}^{k_{max}} \mathfrak{G}_Z^{(l, k)}(\mathfrak{t}_i))] & \text{if } j = 1 \\ \tilde{h}_{i,j}^{(l, 0)} & \text{otherwise.} \end{cases} \\ &\stackrel{(4)+(7)}{=} \begin{cases} \tilde{h}_{i,j}^{(l)} + [\mathfrak{G}_I^{(l)}([\mathfrak{t}_i] - \mathfrak{G}_I^{(l)}(\mathfrak{t}_i))] & \text{if } j = 1 \\ \tilde{h}_{i,j}^{(l)} & \text{otherwise.} \end{cases} \end{aligned}$$

□

The set of all after-tick states and intermediate states forms the set of *reachable states* of \mathcal{Z} . The *reachability graph* start with the initial state s_0 and has all these states as nodes and the concerned *Endsteps*, *Iteratedsteps* or ticks \checkmark as arc inscriptions. Each after-tick state has as many successor nodes as the number of

subsets of the set of endfiring events which may occur in the state. Each of these nodes has as many successor nodes as *Iteratedsteps*. Thus the reachability graph grows very quickly. To avoid the construction of such an enormous reachability graph the consideration of the state equation to decide unreachability will be a good alternative.

2.3 Semantic equivalences

We could have defined firing step sequences of an ITPN as in (10) where for all $l \geq 0$, the *Endstep* $\mathfrak{G}^{(l)}$ may contain transitions to be endfired independently of their age. We would like to define the notion of similar firing step sequences which only differ in the choice of the age of transitions which may and will endfire.

Two firing step sequences σ and σ_0 are called *similar*, denoted by $\sigma_0 \sim \sigma$ if both start at the same state and in all states $S^{(l)}$ and $S_0^{(l)}$ the marking (i.e. their first column) is the same, and the *Globalsteps* are the same.

Thus, in similar firing step sequences only the clock matrices may differ, which signifies that transitions of different ages could have endfired.

The following sentence establishes that w.l.o.g., we can always use as may endfire events the oldest active transitions (as chosen in Definition 1 of Subsection 2.2. above).

Note that in both cases, transitions whose actual durations are the upper bound of their respective time interval ($\delta_i = lfd(t_i)$) must endfire. For the others active transitions (i.e. those which may endfire) we have the choice to choose which transitions do so. Choosing to endfire the oldest active transitions make the choice deterministic.

Example 2 Let be $S^{(3)} = (M^{(3)}, h^{(3)})$ the state reached from the initial state of our running example in Fig.2. by the firing steps sequence $\sigma = (\emptyset, \{\mathbf{t}_2^8\}, \checkmark), (\{\mathbf{t}_2^2\}, \{\mathbf{t}_1, [\mathbf{t}_4, \mathbf{t}_1], [\mathbf{t}_2\}, \checkmark), (\{\mathbf{t}_4\}, \mathbf{t}_2^2), \{\mathbf{t}_1^2, [\mathbf{t}_2, \mathbf{t}_1^2], [\mathbf{t}_2^2\}, \checkmark)$ with $M^{(3)} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ and $h^{(3)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$.

Note that in this state, there are eight active transitions t_2 whose time interval is $[1, 3]$.

From the clock matrix $h^{(3)}$ we can see that there are four transitions t_2 of age 3, one transition of age 2 and three transitions of age 1. Imagine that seven transitions will be endfired.

(a) If only the oldest active transitions are chosen

The intermediate state \tilde{S} with $\tilde{M}^{(3)} = \begin{pmatrix} 0 \\ 14 \end{pmatrix}$ and $\tilde{h}^{(3)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ will be reached.

(b) If transitions of any age may be chosen, then that of age two can be ignored and the following state \tilde{S} with $\tilde{M}^{(3)} = \begin{pmatrix} 0 \\ 14 \end{pmatrix}$ and $\tilde{h}^{(3)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ could be reached, too. \square

Theorem 3 Let \mathcal{Z} be an ITPN and $n \in \mathbb{N}^+$. For each firing step sequence σ of n *Globalsteps* where we choose to may endfire active transitions of any ages, we

can find a sequence σ_0 where always the oldest active transitions are endfired, and $\sigma_o \sim \sigma$. \square

Proof. Let σ be a sequence of $n \geq 1$ global steps where may endfire events are chosen arbitrarily among the active transitions independently of their ages. As defined in (10) it holds that

$$\begin{aligned} \sigma = S^{(0)} \xrightarrow{\mathfrak{G}_\rangle^{(0)}} \tilde{S}^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} S'^{(0)} \xrightarrow{\checkmark} S^{(1)} \xrightarrow{\mathfrak{G}_\rangle^{(1)}} \tilde{S}^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} S'^{(1)} \xrightarrow{\checkmark} S^{(2)} \xrightarrow{\mathfrak{G}_\rangle^{(2)}} \tilde{S}^{(2)} \dots \\ \dots S^{(n-1)} \xrightarrow{\mathfrak{G}_\rangle^{(n-1)}} \tilde{S}^{(n-1)} \xrightarrow{\mathfrak{G}_I^{(n-1)}} S'^{(n-1)} \xrightarrow{\checkmark} S^{(n)} \end{aligned}$$

and $\forall i \leq n$, $S^{(i)} = (M^{(i)}, h^{(i)})$, where $M^{(i)}$ is a marking and $h^{(i)}$ its associated clock matrix. We want to prove, by induction on n , that we can obtain another sequence σ_0 which has the same global steps as σ but different states, by endfiring the oldest active transitions first.

Base : $n = 1$. For the first global step $\sigma = S^{(0)} \xrightarrow{\mathfrak{G}_\rangle^{(0)}} \tilde{S}^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} S'^{(0)} \xrightarrow{\checkmark} S^{(1)}$ we want to construct σ_o similar to σ . The initial state is the same in both cases because we begin from the initial marking and no transition is active. Thus $S_o^{(0)} = (M^{(0)}, h^{(0)}) = S^{(0)}$. The first endfiring multi-set is empty and the age does not play any role. Thus $\tilde{S}_o^{(0)} = \tilde{S}^{(0)}$.

The iterated step contains only endfiring events of zero ages, thus we can use the same multiset of firing $S'_o{}^{(0)} = S'^{(0)}$. After the tick event $S_o^{(1)} = S^{(1)}$ holds.

We conclude that $\sigma_o = S_o^{(0)} \xrightarrow{\mathfrak{G}_\rangle^{(0)}} \tilde{S}_o^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} S'_o{}^{(0)} \xrightarrow{\checkmark} S_o^{(1)}$ is a valid firing step sequence and $\sigma_o \sim \sigma$.

The base of induction is proved.

Induction hypothesis : For all firing step sequences σ of length $i \leq n$, with arbitrarily aged endfiring events, there exists σ_o of length i such that $\sigma_o \sim \sigma$ is supposed to be true and σ_o endfires only the oldest active transitions.

Induction step: Let σ be a firing step sequence of size $(n + 1)$ with arbitrary aged endfiring events.

Thus, the prefix of σ of size n is the following firing step sequence

$$\begin{aligned} \sigma' = S^{(0)} \xrightarrow{\mathfrak{G}_\rangle^{(0)}} \tilde{S}^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} S'^{(0)} \xrightarrow{\checkmark} S^{(1)} \xrightarrow{\mathfrak{G}_\rangle^{(1)}} \tilde{S}^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} S'^{(1)} \xrightarrow{\checkmark} S^{(2)} \xrightarrow{\mathfrak{G}_\rangle^{(2)}} \tilde{S}^{(2)} \dots S^{(n-1)} \xrightarrow{\mathfrak{G}_\rangle^{(n-1)}} \tilde{S}^{(n-1)} \xrightarrow{\mathfrak{G}_I^{(n-1)}} S'^{(n-1)} \xrightarrow{\checkmark} S^{(n)} \end{aligned}$$

and its $(n + 1)$ -th global step is $S^{(n)} = (M^{(n)}, h^{(n)}) \xrightarrow{\mathfrak{G}_\rangle^{(n)}} \tilde{S}^{(n)} = (\tilde{M}^{(n)}, \tilde{h}^{(n)}) \xrightarrow{\mathfrak{G}_I^{(n)}} S'^{(n)} = (M'^{(n)}, h'^{(n)}) \xrightarrow{\checkmark} S^{(n+1)} = (M^{(n+1)}, h^{(n+1)})$. As the ages of transitions in $\mathfrak{G}_\rangle^{(n)}$ are arbitrary, we only know the following about $\tilde{h}^{(n)}, h^{(n)}$:

(a) For all i , $x_i := h_{i, lfd(t_i)+1}$ transitions t_i must endfire in this step. Thus, for each i , $t_i^{x_i}$ is in $\mathfrak{G}_\rangle^{(n)}$ and $\tilde{h}_{i, lfd(t_i)+1}^{(n)} = 0$ follows.

(b) For all i , $y_i := \sum_{1 \leq j \leq lfd(t_i)} h_{i,j}^{(n)} - \sum_{1 \leq j \leq lfd(t_i)} \tilde{h}_{i,j}^{(n)}$ is the number of may endfire transitions in $\mathfrak{G}_\rangle^{(n)}$.

(c) It follows that for all i , $z_i := x_i + y_i = \mathfrak{G}_\rangle^{(n)}(t_i)$.

Now let us prove that there exist σ_o of size $(n+1)$ with $\sigma_o \sim \sigma$, such that the oldest active transitions endfire.

By hypothesis, we have σ'_o , such $\sigma'_o \sim \sigma'$ and σ'_o ends with state $S_o^{(n)}$, such that the states $S^{(n)}$ and $S_o^{(n)}$ have the same markings but may have different clock matrices. In σ'_o only the oldest active transitions have endfired.

We need to prolongate σ'_o by the same $(n+1)$ -th global step $(\mathfrak{G}_\rangle^{(n)}, \mathfrak{G}_I^{(n)}, \checkmark)$. Thus, we have to show the existence of fitting $\tilde{h}_o^{(n)}, h_o'^{(n)}$ and $h_o^{(n+1)}$ such that $S_o^{(n)} = (M^{(n)}, h_o^{(n)}) \xrightarrow{\mathfrak{G}_\rangle^{(n)}} \tilde{S}_o^{(n)} = (\tilde{M}^{(n)}, \tilde{h}_o^{(n)}) \xrightarrow{\mathfrak{G}_I^{(n)}} S_o'^{(n)} = (M'^{(n)}, h_o'^{(n)}) \xrightarrow{\checkmark} S_o^{(n+1)} = (M^{(n+1)}, h_o^{(n+1)})$.

We have first to show that we can endfire z_i active transitions by choosing the oldest ones. Clearly, as the same global steps appeared in σ' and σ'_o , the same number of active transitions appears in the two states $S^{(n)}$ and $S_o^{(n)}$, i.e., for all i , it holds that

$$\sum_{i \geq 1} h_{i,j}^{(n)} = \sum_{i \geq 1} h_{o,i,j}^{(n)} \text{ and } z_i \leq \sum_{i \geq 1} h_{i,j}^{(n)}.$$

Because all preceding global steps are the same for the two sequences, we have precisely the same number of transitions too young to be endfired, i.e., for all i ,

$$\sum_{j \leq sf d(t_i)} h_{i,j}^{(n)} = \sum_{j \leq sf d(t_i)} h_{o,i,j}^{(n)}.$$

Thus, there are also the same number of active transitions which must or may endfire in $S^{(n)}$ and $S_o^{(n)}$.

By consequence, we can take exactly the same endfiring multiset $\mathfrak{G}_\rangle^{(n)}$ as in σ , by choosing the oldest active instance of transitions.

The state $\tilde{S}_o^{(n)} = (\tilde{M}^{(n)}, \tilde{h}_o^{(n)})$, as defined in (2) and (3), and $\tilde{S}^{(n)}$ have clearly the same markings.

Now the same iterated step $\mathfrak{G}_I^{(n)}$ can appear in both states leading to $S_o'^{(n)} = (M'^{(n)}, h_o'^{(n)})$, as defined in (5), (6) and (8), and to $S'^{(n)}$.

Finally, by the tick event we obtain $S_o^{(n+1)} = (M^{(n+1)}, h_o^{(n+1)})$, as defined in (9).

Thus, the firing step sequence σ_o is successfully completed. We can conclude that $\sigma_o \sim \sigma$. \square

3 Algebraic representations

As already quoted, the relationship between a firing step sequence σ and a reachable p -marking M in an ordinary PN with initial p -marking M_0 and a incidence matrix C can be described formally by the following linear equation, where ψ_σ is the Parikh vector of σ : $M = M_0 + C \cdot \psi_\sigma$. A Parikh vector of a word α defined over the finite set, here of transitions $T = \{t_1 \cdots t_n\}$ is a vector of dimension n and the i -th component is the number of appearance of t_i in the word α . Our goal is to obtain a similar result for ITPNs, i.e. to give an algebraic description, precisely, a linear equation, for each firing step sequence, now of

Globalsteps as defined above, in an arbitrary ITPN which takes into account the time, too. Meanwhile state equations had been introduced for TPN with fixed duration [8] and for ITPN without auto-concurrency and without zero duration [7], where the semantics had been formulated in a more algebraic way. We will present in the following the formal definitions of the notions we need later for the different proofs. Some of them are adaptations of definitions known for the algebraic presentation of a firing step sequence for TPN, or ITPN without zero duration and without auto-concurrency [8,7] and others are entirely new here.

3.1 Semantics with time markings

In this subsection we introduce a more detailed view of the p -markings in an arbitrary ITPN with respect to the time. This view makes it possible to obtain a time-dependent state equation for a firing step sequence and it delivers a sufficient condition for the non-reachability of p -markings (timeless) as well as of time markings in such a net.

First, to calculate the effect of *Endstep* \mathfrak{G}_γ we introduce a new $(|T| \times d)$ matrix, denoted by G_γ which is the matrix representation of the *Endstep* multiset, fixing which events have to endfire, by taking the oldest ones.

$$\text{Let } S^{(1)} = (M^{(1)}, h^{(1)}) \xrightarrow{\mathfrak{G}_\gamma^{(1)}} \tilde{S}^{(1)} = (\tilde{M}^{(1)}, \tilde{h}^{(1)})$$

$$G_{\gamma,i,j}^{(1)} := \begin{cases} h_{i,j}^{(1)} & \text{if } \mathfrak{G}_\gamma^{(1)}(\mathbf{t}_i) - \sum_{j' \geq j} h_{i,j'}^{(1)} \geq 0 \\ q & \text{if } \mathfrak{G}_\gamma^{(1)}(\mathbf{t}_i) - \sum_{j' > j} h_{i,j'}^{(1)} = q > 0 \text{ and } q < h_{i,j}^{(1)} \\ 0 & \text{otherwise.} \end{cases} \quad (11)$$

The element $G_{\gamma,i,j}^{(1)}$ fixes the number of t_i whose age is $(j-1)$ and which is chosen to endfire.

Lemma 4 *Let an Endstep $\mathfrak{G}_\gamma^{(1)}$ appear in state $S^{(1)}$, i.e., $S^{(1)} =$*

$(M^{(1)}, h^{(1)}) \xrightarrow{\mathfrak{G}_\gamma^{(1)}} \tilde{S}^{(1)} = (\tilde{M}^{(1)}, \tilde{h}^{(1)})$ and let $G_\gamma^{(1)}$ be its associated matrix as defined in (11). Then for all i, j it holds that $\tilde{h}_{i,j}^{(1)} = h_{i,j}^{(1)} - G_{\gamma,i,j}^{(1)}$. \square

The proof is an immediate consequence of the above definition (11).

Second, in order to describe the relation between tokens and time algebraically, we use a generalization of the p -marking, called *time marking*, cf. [8]. A time marking is a $(|P| \times (d+1))$ -matrix. The number of rows is equal to the number of places and the number of columns, $d+1$, equals the maximum of all longest durations in the considered ITPN, plus 2. They are numbered from 1 to $d+1$. Each column can be considered to be a p -marking. The first column represents the number of visible tokens in each place, i.e. the actual p -marking M . The other columns represent tokens which are on their way to the places: column number two for those arriving immediately, column number three for those arriving in one time unit (one tick later), the column number four for those arriving in two time units (after two ticks), and so on. We may observe, that only a finite number of time markings can be associated with a given p -marking M .

This number depends on the time-dimension d of the net and is exponential in $|T|$.

A *time state* s is now defined as a pair (m, h) , where m is a time marking and h is a clock-matrix. The *initial time marking* $m^{(0)}$ is defined as

$$m_{\cdot 1}^{(0)} = M^{(0)} \text{ and } m_{i,j}^{(0)} = 0 \text{ for } i = 1 \dots |P| \text{ and } j = 2 \dots d + 1. \quad (12)$$

The *initial time state* $s^{(0)}$ is the pair $(m^{(0)}, h^{(0)})$ considered now to be the *first after-tick time state*.

Example 5 Consider the ITPN \mathcal{Z}_o with $d = 4$ and $m^{(0)} = \begin{pmatrix} 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$. This initial time marking allows many possible *Globalsteps* such as, e.g.,

1. $(\mathfrak{G})^{(0)} = \emptyset, \mathfrak{G}_I^{(0)} = \{[t_2^8]\};$
2. $(\mathfrak{G})^{(0)} = \emptyset, \mathfrak{G}_I^{(0)} = \{[t_2^6, [t_3] \uplus \{t_3\}\} \uplus \{[t_4] = \{[t_2^6, [t_3, [t_4, t_3]]\}\};$
3. $(\mathfrak{G})^{(0)} = \emptyset, \mathfrak{G}_I^{(0)} = \{[t_2^2, [t_3^3] \uplus \{t_3\}^3\} \uplus \{[t_1] \uplus \{t_1\}\} \uplus \{[t_2]$
 $= \{[t_2^3, [t_3^3, [t_1, t_3]^3, t_1]\}\}.$

The choice of one *Globalsteps* among those above is arbitrary. We will consider later the third one appearing. \square

Let $s^{(1)} = (m^{(1)}, h^{(1)})$ be an after-tick time state in some ITPN \mathcal{Z} , and $(\mathfrak{G})^{(1)}, \mathfrak{G}_I^{(1)}$ a *Globalstep* which may appear from state $S^{(1)} = (M^{(1)}, h^{(1)})$ as defined in Subsection 2.1. above. We will adapt the definitions now to show how the execution of this *Globalstep* changes the time state $s^{(1)}$, by using matrix $G_{\rangle}^{(1)}$ for the calculations.

a) By firing the *Endstep* we obtain $s^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} \tilde{s}^{(1)} = (\tilde{m}^{(1)}, \tilde{h}^{(1)})$, with

$$\tilde{m}_{i,j}^{(1)} := \begin{cases} m_{i,j}^{(1)} + \sum_{k \geq 1} \left(\sum_{r \geq 1}^d G_{\rangle k,r}^{(1)} \cdot v(t_k, p_i) \right) & \text{if } j = 1 \\ m_{i,j}^{(1)} - \sum_{k \geq 1} G_{\rangle k,j'}^{(1)} \cdot v(t_k, p_i) & \text{if } j > 1 \text{ and} \\ & j' = lfd(t_k) - j + 3 \end{cases} \quad (13)$$

and $\tilde{h}_{i,j}^{(1)} := h_{i,j}^{(1)} - G_{\rangle i,j}^{(1)}$ (by Lemma 4). It is clear that $\tilde{m}_{i,2}^{(1)} = 0$.

b) By firing the *Iteratedstep* we obtain $\tilde{s}^{(1)} \xrightarrow{\mathfrak{G}_I^{(1)}} s'^{(1)} = (m'^{(1)}, h'^{(1)})$. The *Iteratedstep* change the first column of the time marking, $m'_{i,1} = M'^{(1)}$, as shown in Lemma 1. For each transition $t_k \in \mathfrak{G}_I^{(1)}$ the j -th column can be modified if $j = lfd(t_k) + 2$, but t_k does not influence the others columns. Hence, it holds that $m'_{i,j}^{(1)} :=$

$$\begin{cases} \tilde{m}_{i,j}^{(1)} - \sum_{k \geq 1} \mathfrak{G}_I^{(1)}([t_k] \cdot v(p_i, t_k) + \sum_{k \geq 1} \mathfrak{G}_I^{(1)}(t_k) \cdot v(t_k, p_i) & \text{if } j = 1 \\ \tilde{m}_{i,j}^{(1)} + \sum_{\substack{1 \leq k \leq |T| \\ j = lfd(t_k) + 2}} \left[\mathfrak{G}_I^{(1)}([t_k] - \mathfrak{G}_I^{(1)}(t_k)) \right] \cdot v(t_k, p_i) & \text{if } j > 1 \end{cases} \quad (14)$$

The clock matrix $h'^{(1)}$ does not need to be recalculated: the definitions of (6) and (8) apply.

c) Now one tick has to occur $s'^{(1)} \xrightarrow{\checkmark} s^{(2)} = (m^{(2)}, h^{(2)})$ with

$$m_{i,j}^{(2)} := \begin{cases} m_{i,j}'^{(1)} & \text{if } j = 1 \\ m_{i,j+1}'^{(1)} & \text{if } 2 \leq j \leq d \\ 0 & \text{if } j = d + 1 \end{cases} \quad (15)$$

The clock matrix $h^{(2)}$ is already defined in (9). The time state $s^{(2)}$ is a new after-tick time state. We can observe, that in the defined time markings the first column is in fact always the usual p -marking of the corresponding state.

Example 6 Let us reconsider the running example \mathcal{Z}_o and the selected *Global-step* appearing from the initial state $s^{(0)}$: $(\mathfrak{G})^{(0)} = \emptyset$, $\mathfrak{G}_I^{(0)} = \{[t_2^3, [t_3, [t_1, t_3]^3, t_1]\}$.

Then the time states reached during its firing and the subsequent tick $s^{(0)} \xrightarrow{\mathfrak{G}^{(0)}} \tilde{s}^{(0)} \xrightarrow{\mathfrak{G}_I^{(0)}} s'^{(0)} \xrightarrow{\checkmark} s^{(1)}$ have the following time markings

$$\tilde{m}^{(0)} = m^{(0)} = \begin{pmatrix} 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}, m'^{(0)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{pmatrix}, m^{(1)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \end{pmatrix}.$$

As $G^{(0)} = \mathcal{O}$, it holds that $\tilde{h}^{(0)} = h^{(0)} = \mathcal{O}$. As $\mathfrak{G}_I^{(0)}([t_3] - \mathfrak{G}_I^{(0)}(\mathfrak{t}_3)) = 3$ it follows that

$$h'_{2,1}^{(0)} = 3 \text{ and } \tilde{h}^{(0)} = h^{(0)} = \mathcal{O}, \quad h'^{(0)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad h^{(1)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ after a right shift.} \quad \square$$

Analogously to states, we call *reachable time states* all after-tick and intermediate time states reached during the execution of arbitrary firing step sequences.

3.2 Algebraic calculus of the semantics

In the following we introduce all matrices which are necessary to obtain a state equation for some ITPN, starting with the so called *time incidence matrix*.

Let \mathcal{Z} be an ITPN. The $(|P| \times (d+1) \cdot |T|)$ -matrix C is called the *time incidence matrix* of \mathcal{Z} , if $C := (C_{(1)}, C_{(2)}, \dots, C_{(|T|)})$ with $C_{(k)}$ being a $(|P| \times d)$ -matrix for each $k \in \{1, \dots, |T|\}$, such that $C_{(k)} = \left(c_{i,r}^{(k)} \right)_{\substack{i=1 \dots |P| \\ r=1 \dots d}}$ and

$$c_{i,r}^{(k)} := \begin{cases} -v(p_i, t_k) & \text{if } r = 1 \\ v(t_k, p_i) & \text{if } r - 2 = \text{lfd}(t_k) \\ 0 & \text{otherwise.} \end{cases}$$

The matrix C consists of submatrices $C_{(k)}$ representing the transitions t_k of the net. Each $c_{i,1}^{(k)}$ is the number of tokens that will be changed (decremented) at place p_i immediately when the startfire event $[t_k$ appears, and $c_{i,r}^{(k)}$ shows the number of tokens that will arrive at place p_i when the endfire event \mathfrak{t}_k appears after at most $(r - 2)$ time units.

Example 7 The time incidence matrix of \mathcal{Z}_o from Fig.2 is as follows:

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 2 & 0 \end{pmatrix}. \quad \square$$

Obviously the time incidence matrix takes into account the longest firing duration $lfd(t_i)$ for each transition t_i .

The appearance of \mathbf{t}_i^n in some $\mathfrak{G}_\gamma^{(l)}$ at a certain state $s^{(l)} = (m^{(l)}, h^{(l)})$ tells us that there are at least n active transitions. The matrix $G_\gamma^{(l)}$ associated to the end-step tells us which ones are going to endfire.

For subsequent computation we need to update the matrix C with respect to $\mathfrak{G}_\gamma^{(l)}$. This is achieved by matrix $C^{(l)}$ obtained from C where for each submatrix $C_{(i)}^{(l)}$ the first column represents the tokens consumed by the transitions to endfire and the j -th column represents the tokens arriving to the corresponding places after $j - 2$ ticks at least.

Therefore, concerning $\mathfrak{G}_\gamma^{(l)}$ in the state $s^{(l)} = (m^{(l)}, h^{(l)})$, we define the matrix

$C^{(l)} := (C_{(1)}^{(l)}, C_{(2)}^{(l)}, \dots, C_{(|T|)}^{(l)})$ as follows. Each $C_{(k)}^{(l)} = (c_{i,r}^{(l,k)})_{\substack{i=1 \dots |P| \\ r=1 \dots d}}$ is a $(|P| \times (d+1))$ -matrix with

$$c_{i,r}^{(l,k)} := \begin{cases} -v(p_i, t_k) \cdot \mathfrak{G}_\gamma(t_k) & \text{if } r = 1 \\ v(t_k, p_i) \cdot G_{\gamma,k,r'} & \text{if } r > 1 \text{ and } r' = lfd(t_k) - r + 3 \end{cases} \quad (16)$$

Example 8 In the ITPN \mathcal{Z}_o let us consider the endfiring step $s^{(l)} =$

$(m^{(l)}, h^{(l)}) \xrightarrow{\mathfrak{G}_\gamma^{(l)}} \tilde{s}^{(l)} = (\tilde{m}^{(l)}, \tilde{h}^{(l)})$ with $m^{(l)} = \begin{pmatrix} 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 10 & 8 & 0 \end{pmatrix}$, $h^{(l)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 4 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix}$ and

$\mathfrak{G}_\gamma^{(l)} = \{t_2^4, t_4^3\}$. Then its associated matrix is $G_\gamma^{(l)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$.

The time incidence matrix $C^{(l)}$ arises from the matrix C as follows:

$$C^{(l)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 0 & 6 & 0 & 0 \end{pmatrix}. \quad \square$$

Our goal now is to introduce a sparse matrix U which allows us to calculate $C^{(l)}$ from C , such that $C^{(l)} = C \cdot U^{(l)}$ for its submatrix $U^{(l)}$. Let us consider $t_k \notin \mathfrak{G}_\gamma^{(l)}$ and $t_i \in \mathfrak{G}_\gamma^{(l)}$.

We define the square matrix $U^{(l)}$ with $(d+1) \cdot |T|$ rows and $(d+1) \cdot |T|$ columns

\mathcal{O} stands for a block of zeros, $A_i^{(l)}$ is a $(d+1 \times d+1)$ matrix obtained from E_{d+1} by:

(1) Multiplying the first column of E_{d+1} by $\mathfrak{G}_\gamma^{(l)}(\mathbf{t}_i)$ which is the number of occurrences of endfiring event \mathbf{t}_i in the end-step $\mathfrak{G}_\gamma^{(l)}$.

(2) Superseding the $(lfd(t_i) - j + 3)$ -th column of E_{d+1} by the $(lfd(t_i) + 2)$ -th column multiplied by $G_{\gamma,i,j}^{(l)}$ for each $j \in [0, d]$ as follows:

$$U^{(l)} = \begin{matrix} & \begin{matrix} t_1 & & t_k & & t_i & & t_n \end{matrix} \\ \begin{matrix} t_1 \\ t_k \\ t_i \\ t_n \end{matrix} & \begin{pmatrix} A_1^{(l)} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{O} & \ddots & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} & \mathcal{O} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{O}_{d+1} & \mathcal{O} & A_k^{(l)} = \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} \\ \vdots & \mathcal{O} & \vdots & \mathcal{O} & \ddots & \vdots & \mathcal{O} \\ \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & A_i^{(l)} & \mathcal{O} & \mathcal{O}_{d+1} \\ \vdots & \mathcal{O} & \vdots & \mathcal{O} & \vdots & \ddots & \vdots \\ \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & \mathcal{O}_{d+1} & \mathcal{O} & A_n^{(l)} \end{pmatrix} \end{matrix}.$$

Example 9 In \mathcal{Z}_o from Fig. 2 we consider the same end-step $\mathfrak{G}_\rangle^{(l)} = \{t_2\}^4, t_4\}^3\}$ with

$G_\rangle^{(l)} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{pmatrix}$. We obtain the corresponding matrices $A_2^{(l)}$, $A_4^{(l)}$ and $U^{(l)}$:

$$A_2^{(l)} = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 3 \end{pmatrix}, A_4^{(l)} = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, U^{(l)} = \begin{pmatrix} O_5 & O_5 & O_5 & O_5 \\ O_5 & A_2^{(l)} & O_5 & O_5 \\ O_5 & O_5 & O_5 & O_5 \\ O_5 & O_5 & O_5 & A_4^{(l)} \end{pmatrix}. \quad \square$$

It is evident that matrix $U^{(l)}$ makes it possible to calculate $C^{(l)}$ because the values of each submatrix $C_{(k)}^{(l)}$ of $C^{(l)}$ verify with respect to the endfire events t_k

$$C_{(k)}^{(l)} = \begin{cases} C_{(k)} \cdot A_{(k)}^{(l)} & \text{if } t_k \in \mathfrak{G}_\rangle^{(l)} \\ C_{(k)} \cdot \mathcal{O}_{d+1} & \text{otherwise.} \end{cases}$$

The $(|P| \times (d+1) \cdot |T|)$ -matrix $C^{(l)} = C \cdot U^{(l)}$ is called *time incidence matrix with actual durations* for the end-step $\mathfrak{G}_\rangle^{(l)}$.

In the following calculi (just below and later) we need some matrices, all of them are sparse square $(d+1 \times d+1)$ matrices: Besides the already introduced *identity matrix* E_{d+1} and *zero-matrix* \mathcal{O}_{d+1} , we define here the matrices $L_{d+1} = (l_{ij})$, $W_{d+1} = (w_{ij})$ and the *progress matrix* $R_{d+1} = (r_{ij})$ by setting

$$l_{ij} := \begin{cases} 1 & \text{if } i \geq 2 \\ & \text{and } i = j, \\ 0 & \text{otherwise} \end{cases}, w_{ij} := \begin{cases} 1 & \text{if } i \geq 2 \\ & \text{and } j = 1, \\ 0 & \text{otherwise.} \end{cases}, r_{i,j} := \begin{cases} 1 & \text{if } (i = j = 1) \\ & \text{or } (i = j + 1) \\ 0 & \text{otherwise} \end{cases}.$$

For simplicity we write R instead of R_{d+1} if $d+1$ is clear from the context.

Example 10 For the running example \mathcal{Z}_o from Fig.1 with $d+1 = 5$ these square matrices are

$$L_5 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}, W_5 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}, R_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \quad \square$$

Now, let us observe the utility of these matrices. If we multiply an arbitrary $(l \times (d+1))$ -matrix A by L_{d+1} we obtain a $(l \times (d+1))$ -matrix $B = A \cdot L_{d+1}$ whose first column is the l -dimensional zero-vector and the rest of its columns are the same as in the matrix A . If we multiply A by W_{d+1} we obtain a $(l \times (d+1))$ -matrix $B' = A \cdot W$ whose first column is the sum of all but the first columns of A and all the other columns are zero-vectors. Finally, if we multiply A by R_{d+1} we obtain a $(l \times (d+1))$ -matrix $B'' = A \cdot W$ whose i -th column is the $(i+1)$ -th column of A , except the first one and the last one. Thus the multiplication by R insures a shift. The first column of B'' is the sum of the first and second columns of A and the last one is a zero-vector.

Now, for each *Endstep* $\mathfrak{G}_\rangle^{(l)} = \{t_{i_1}^{n_{i_1}}, \dots, t_{i_\rho}^{n_{i_\rho}}\}$ and *Iteratedstep* $\mathfrak{G}_I^{(l)} = \{[t_{i_1}^{n_{i_1}}, t_{i_1}^{q_{i_1}}], \dots, [t_{i_\kappa}^{n_{i_\kappa}}, t_{i_\kappa}^{q_{i_\kappa}}]\}$, with $q_s \leq n_s$ for all $s \in [1 \dots \rho]$. we define a matrix $B_\rangle^{(l)}$, called the *bag matrix* of $\mathfrak{G}_\rangle^{(l)}$ as well as the matrices $B_M^{(l)}$ and $B_Z^{(l)}$

called the *bag matrices* of $\mathfrak{G}_I^{(l)}$, all being $(d+1 \cdot |T| \times (d+1))$ matrices, by setting

$$B_{\rangle}^{(l)} = \begin{pmatrix} B_{\rangle(1)}^{(l)} \\ B_{\rangle(2)}^{(l)} \\ \vdots \\ B_{\rangle(|T|)}^{(l)} \end{pmatrix}, B_M^{(l)} = \begin{pmatrix} B_{M(1)}^{(l)} \\ B_{M(2)}^{(l)} \\ \vdots \\ B_{M(|T|)}^{(l)} \end{pmatrix} \text{ and } B_Z^{(l)} = \begin{pmatrix} B_{Z(1)}^{(l)} \\ B_{Z(2)}^{(l)} \\ \vdots \\ B_{Z(|T|)}^{(l)} \end{pmatrix} \text{ where}$$

$$B_{\rangle(s)}^{(l)} := \begin{cases} L_{d+1} & \text{if } s \in \{i_1, \dots, i_\rho\}, \\ 0 \cdot E_{d+1} & \text{otherwise.} \end{cases}, \quad (17)$$

$$B_{M(s)}^{(l)} := \begin{cases} \mathfrak{G}_I^{(l)}(\mathfrak{t}_s) \cdot E_{d+1} & \text{if } s \in \{i_1, \dots, i_\kappa\}, \\ 0 \cdot E_{d+1} & \text{otherwise.} \end{cases},$$

$$B_{Z(s)}^{(l)} := \begin{cases} \mathfrak{G}_I^{(l)}(\mathfrak{t}_s) \cdot L_{d+1} & \text{if } s \in \{i_1, \dots, i_\kappa\} \\ 0 \cdot E_{d+1} & \text{otherwise.} \end{cases} \quad (18)$$

Remark 1 In the bag matrices for Endsteps $B_{\rangle}^{(l)}$ and $B_Z^{(l)}$, the first column is obviously a zero vector.

Example 11 The Iteratedstep $\mathfrak{G}_I^{(l)} = \{[t_2^6, [t_3, [t_1, t_3]]\}$ of the net Z_0 from Fig. 1 yields $B_M^{(l)} = \begin{pmatrix} 1 \cdot E_5 \\ 6 \cdot E_5 \\ 1 \cdot E_5 \\ 0 \cdot E_5 \end{pmatrix}$ and $B_Z^{(l)} = \begin{pmatrix} 0 \cdot L_5 \\ 0 \cdot L_5 \\ 1 \cdot L_5 \\ 0 \cdot L_5 \end{pmatrix}$. \square

Finally, we consider two $((d+1) \cdot |T| \times (d+1))$ -matrices $K_{\rangle}^{(l)}$ and $B_I^{(l)}$ which help us to describe algebraically the effect of respectively an *Endstep* and an *Iteratedstep*.

We will prove that the following terms describe exactly this change.

$$\begin{aligned} -C^{(l)} \cdot B_{\rangle}^{(l)} + C^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d &= -\underbrace{C \cdot U^{(l)}}_{C^{(l)}} B_{\rangle}^{(l)} + \underbrace{C \cdot U^{(l)}}_{C^{(l)}} B_{\rangle}^{(l)} \cdot R^d \\ &= C \left(\underbrace{-U^{(l)} B_{\rangle}^{(l)} + U^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d}_{:=K_{\rangle}^{(l)}} \right) = C \cdot K_{\rangle}^{(l)} \end{aligned} \quad (19)$$

$$\begin{aligned} \text{and } C \cdot B_M^{(l)} - C \cdot B_Z^{(l)} + C \cdot B_Z^{(l)} \cdot R^d &= \\ C \left(\underbrace{B_M^{(l)} - B_Z^{(l)} + B_Z^{(l)} \cdot R^d}_{:=B_I^{(l)}} \right) &= C \cdot B_I^{(l)}. \end{aligned} \quad (20)$$

4 State equation

In this section we derive a state equation for an arbitrary ITPN that is analogous to the state equation (1) of time-less nets and which is consistent with the state equation for ITPNs without auto concurrency and zero durations [8].

We consider in the firing step sequence given in (10) the effects of the *Globalstep* appearing at the after-tick time state $s^{(l)}$, for some natural number $l \leq n$, as well as of its subsequent tick event :

$$s^{(l)} \xrightarrow{\mathfrak{G}_{\rangle}^{(l)}} \tilde{s}^{(l)} \xrightarrow{\mathfrak{G}_I^{(l)}} s'^{(l)} \xrightarrow{\checkmark} s^{(l+1)}. \quad (21)$$

The following two remarks are easy to prove.

Remark 2 For all $k \geq d$ it holds that $R^k =: (f_{i,j})_{i=1 \dots d, j=1 \dots d}$ with

$$f_{i,j} = \begin{cases} 1 & \text{if } i=1 \\ 0 & \text{otherwise} \end{cases}.$$

Remark 3 Let be $W^{(l)} := B_{\rangle}^{(l)} \cdot R^d$. Then the matrix $W^{(l)}$ has the following

structure: $W^{(l)} = \begin{pmatrix} w_{(1)}^{(l)} \\ w_{(2)}^{(l)} \\ \vdots \\ w_{(|T|)}^{(l)} \end{pmatrix} \quad \text{and} \quad W_{(s)}^{(l)} := \begin{cases} W_d & \text{if } t_s \in \mathfrak{G}_l \\ O_d & \text{otherwise.} \end{cases}.$

Lemma 12 Let us consider the $(|P| \times d + 1)$ - matrix $Q^{(l)} := C^{(l)} \cdot B_{\rangle}^{(l)}$. Then its elements $q_{i,j}$ have the following values:

$$q_{i,j}^{(l)} = \begin{cases} 0 & \text{if } j = 1 \\ \sum_{k=1}^{|T|} G_{\rangle k, j'} \cdot v(t_k, p_i) & \text{if } 1 < j \leq d + 1 \text{ and } j' = \text{lfd}(t_k) - j + 3 \end{cases}.$$

Proof. We compute the elements $q_{i,j}^{(l)}$.

Case 1: $j = 1$. Then $q_{i,1}^{(l)} = \left(C^{(l)} \cdot B_{\rangle}^{(l)} \right)_{i,1} = \left(\sum_{k=1}^{|T|} C_{(k)}^{(l)} \cdot B_{\rangle(k)}^{(l)} \right)_{i,1} = \sum_{r=1}^{|T|} \sum_{k=1}^{(d+1)} \underbrace{c_{i,k}^{(l,r)} \cdot b_{k,1}^{(l,r)}}_{=0} = 0.$

Case 2: $1 < j \leq d + 1$. Then

$$\begin{aligned} q_{i,j}^{(l)} &= \left(C^{(l)} \cdot B_{\rangle}^{(l)} \right)_{i,j} = \left(\sum_{k=1}^{|T|} C_{(k)}^{(l)} \cdot B_{\rangle(k)}^{(l)} \right)_{i,j} = \sum_{r=1}^{|T|} \sum_{k=1}^{d+1} \left(c_{i,k}^{(l,r)} \cdot b_{k,j}^{(l,r)} \right) \\ &= \sum_{r=1}^{|T|} \left(c_{i,j}^{(l,r)} \cdot 1 \right) \stackrel{(16)}{=} \sum_{k=1}^{|T|} G_{\rangle k, (\text{lfd}(t_k) - j + 3)} \cdot v(t_k, p_i). \end{aligned} \quad \square$$

We will first establish linear equations for the time markings around a firing step.

Theorem 13 Let \mathcal{Z} be an ITPN, and let the time states $s^{(l)} = (m^{(l)}, h^{(l)})$, $\tilde{s}^{(l)} = (\tilde{m}^{(l)}, \tilde{h}^{(l)})$, $s'^{(l)} = (m'^{(l)}, h'^{(l)})$ and $s^{(l+1)} = (m^{(l+1)}, h^{(l+1)})$ be defined as in (21). Then the time markings fulfil

$$\tilde{m}^{(l)} = m^{(l)} + C \cdot K_{\rangle}^{(l)} \quad (22)$$

$$m'^{(l)} = \tilde{m}^{(l)} + C \cdot B_I^{(l)} \quad (23)$$

$$m^{(l+1)} = m'^{(l)} \cdot R \quad (24)$$

Proof of equation (22) :

In order to derive (22) we have to show that $(\tilde{m}^{(l)})_{i,j} = (m^{(l)})_{i,j} + (C \cdot K_{\rangle}^{(l)})_{i,j}$ for each $i \in \{1, \dots, |P|\}$ and $j \in \{1, \dots, d + 1\}$.

Case 1: $j = 1$. According to the definition of time markings (13) it holds that

$$\left(\tilde{m}^{(l)}\right)_{i,1} - \left(m^{(l)}\right)_{i,1} = \left(\sum_{k=1}^{|T|} \left(\sum_{r=1}^d G_{\rangle k,r}^{(l)}\right) \cdot v(t_k, p_i)\right).$$

Thus we have to prove that $\left(\sum_{k=1}^{|T|} \left(\sum_{r=1}^d G_{\rangle k,r}^{(l)}\right) \cdot v(t_k, p_i)\right) = \left(C \cdot K_{\rangle}^{(l)}\right)_{i,1}$. It holds

$$\left(C \cdot K_{\rangle}^{(l)}\right)_{i,1} \stackrel{(19)}{=} \left(-C \cdot U^{(l)} \cdot B_{\rangle}^{(l)}\right)_{i,1} + \left(C \cdot U^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d\right)_{i,1}. \quad (25)$$

Now we first consider the term $\left(-C \cdot U^{(l)} \cdot B_{\rangle}^{(l)}\right)_{i,1}$. As the first column of the matrix $B_{\rangle}^{(l)}$ consists only of zeros, it holds that $\left(-C \cdot U^{(l)} \cdot B_{\rangle}^{(l)}\right)_{i,1} =$

$$\left(-C^{(l)} \cdot B_{\rangle}^{(l)}\right)_{i,1} = -\left(Q^{(l)}\right)_{i,1} = 0. \text{ (cf. lemma 12)} \quad (26)$$

Subsequently, we consider the second term $\left(C \cdot U^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d\right)_{i,1}$. By remark 3 we know that

$$\begin{aligned} \left(C \cdot U^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^{d-1}\right)_{i,1} &= \left(C^{(l)} \cdot W\right)_{i,1} = \sum_{k=1}^{(d+1) \cdot |T|} c_{i,k}^{(l)} \cdot w_{k,1}^{(l)} = \\ &\sum_{r=1}^{|T|} \sum_{k=1}^{d+1} c_{i,k}^{(l,r)} \cdot w_{k,1}^{(l,r)} \stackrel{(16)}{=} \sum_{k=1}^{|T|} \left(\sum_{r=1}^d G_{\rangle k,r}^{(l)}\right) \cdot v(t_k, p_i). \end{aligned} \quad (27)$$

Considering (25),(26) and (27) leads to the equation $\left(\tilde{m}^{(l)}\right)_{i,1} = \left(m^{(l)}\right)_{i,1} + \left(C \cdot K_{\rangle}^{(l)}\right)_{i,1}$, as desired.

Case 2: $j > 1$.

According to the definition of time markings in (13) it holds that

$$\tilde{m}_{i,j}^{(l)} - m_{i,j}^{(l)} = -\sum_{k=1}^{|T|} G_{\rangle k, (lf d(t_k) - j + 3)}^{(l)} \cdot v(t_k, p_i). \quad (28)$$

Thus, we have to prove that $\left(C \cdot K_{\rangle}^{(l)}\right)_{i,j} = -\sum_{k=1}^{|T|} G_{\rangle k, (lf d(t_k) - j + 3)}^{(l)} \cdot v(t_k, p_i)$.

It holds that

$$\begin{aligned} \left(C \cdot K_{\rangle}^{(l)}\right)_{i,j} &\stackrel{(19)}{=} \left(-C \cdot U^{(l)} \cdot B_{\rangle}^{(l)} + C \cdot U^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d\right)_{i,j} \\ &= \left(-C^{(l)} \cdot B_{\rangle}^{(l)} + C^{(l)} \cdot B_{\rangle}^{(l)} \cdot R^d\right)_{i,j} \\ &= \left(-Q^{(l)} + C^{(l)} \cdot W^{(l)}\right)_{i,j} \text{ (cf. Rem. 3 and Lemma 12)} \\ &= -\left(Q^{(l)}\right)_{i,j} + \left(C^{(l)} \cdot W^{(l)}\right)_{i,j} \\ &= -\left(Q^{(l)}\right)_{i,j} + 0 = q_{i,j}^{(l)} \text{ (cf. Rem. 3)} \\ &= -\sum_{k=1}^{|T|} G_{\rangle k, (lf d(t_k) - j + 3)}^{(l)} \cdot v(t_k, p_i). \text{ (cf. Lemma 12)} \end{aligned}$$

The proofs of equations (23) and (24) can be done similarly. \square

Now we can deduce the main result, i.e., the equation for the sequence (10):

Theorem 14 *Let \mathcal{Z} be an ITPN, $n \geq 1$ and σ a firing step sequence consisting of n Globalsteps, alternating with ticks, leading to the time state $s^{(n)} = (m^{(n)}, h^{(n)})$ as defined in (10). Then the time marking $m^{(n)}$ fulfils $m^{(n)} =$*

$$m^{(0)} \cdot R^n + C \cdot \Psi_\sigma \text{ where } \Psi_\sigma = \sum_{l=1}^n \left(K^{(l-1)} + B_I^{(l-1)} \right) \cdot R^{n+1-l}. \quad (29)$$

The proof can be done by induction on n . \square

We call Ψ_σ , which is a $((d+1) \cdot |T| \times |P|)$ - matrix, the *Parikh matrix* and equation (29) the *state equation* of the firing step sequence (10). Analogously to the Parikh vector, the Parikh matrix counts the number of appearances of startfire and endfire events in (10).

It is evident, that due to Theorems 13 and 14, we can analogously establish state equations for the other (intermediate) time markings, such as $m'^{(n)}$ and $\tilde{m}^{(n)}$, that appear in the firing step sequences.

The last Theorem 14 provides a sufficient condition for the non-reachability of a given time marking. Let us explain what it means to show that there does not exist a sequence, nevertheless which length, such that after firing of the sequence from the initial time state, the net is in a time state whose time marking is the given one. For this reason, similar to the case for classic Petri nets, we have to solve an system of equalities defined by the equation (29). Of course, this system of equalities is much more difficult than that for the equation (1) for classic PNs.

The number of variables in the state equation is around $n \cdot |T| \cdot ((d+1)^2/2 + 3)$, in total. Additionally, there are some more additional "local" equalities/inequalities.

Finally, we have to prove that for no n the obtained system of equalities of the state equation has an integer solution. In that case the given time marking is not reachable. In the other case - if there is an integer solution for some particular n - then no assertion can be done about the reachability of the time marking. It could be possible that the solution represents only non realizable sequences with, for instance, intermediate states which would have negative values.

5 Conclusion

In this article we have studied the class of Interval-Timed Petri nets with discrete delays in their most complex version. Firstly, zero duration is allowed (i.e. zero is possible as a lower bound of the duration interval of a transition), which has as consequence that in between two time ticks a certain number of transitions may start and end and provoke the start and perhaps ending of others, and so on. We consider only well formed nets where this number is always finite, i.e. where there is no undesired cycle of transitions of zero duration.

Then we allow auto-concurrency in the firing of transitions. This means that in maximal steps several instances of the same transition may start at the same

moment and could have independent durations. Our notion of *Globalstep*, which consists of all startfire and endfire events in between two time ticks, is original.

When in a state a subbag of concurrently active instances of the same transition should end we could choose to end the oldest ones between them or arbitrary ones. We prove that both ways are equivalent, leading to sequences composed of the same *Globalsteps*. This result allows us to choose once for all in this article to end always the oldest active transitions.

To obtain adequate formalizations, original algebraic structures have been proposed for all defined concepts.

In this complex algebraic context, our goal was to construct state equations for the considered net class. We proposed a series of results which lead to the main theorem, which establishes that each reachable time state fulfils a certain nontrivial state equation. The paper contains all proofs.

By contraposition we may conclude, that a time state is unreachable in the considered Interval-Timed Petri net when the system of equalities associated to its state equation has no solution.

References

1. H. Fleischhack and E. Pelz. Hierarchical Timed High Level Nets and their Branching Processes. In *Proceedings of ICATPN'03, LNCS 2679, Springer*, pages 397–416, 2003.
2. M. Heiner and L. Popova-Zeugmann. Worst-case Analysis of Concurrent Systems with Duration Interval Petri Nets. In *Proceedings of 5. Fachtagung Entwurf komplexer Automatisierungssysteme, TU Braunschweig*, 1997.
3. P. Merlin. *A Study of the Recoverability of Communication Protocols*. PhD thesis, Irvine, 1974.
4. C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *IFIP Congress*, 1962.
5. L. Popova-Zeugmann. On Time Petri Nets. *J. Inform. Process. Cybern. EIK* 27(1991)4, 1991.
6. L. Popova-Zeugmann. *Time and Petri Nets*. Springer, 2013.
7. L. Popova-Zeugmann and E. Pelz. Algebraical Characterisation of Interval-Timed Petri Nets with Discrete Delays. *Fundamenta Informaticae*, 120(3-4):341–357, 2012.
8. L. Popova-Zeugmann, M. Werner, and J. Richling. Using State-equation to Prove Non-reachability in Timed Petrinets. *Fundamenta Informaticae (FI)*, 61, IOS-Press, Amsterdam, 55:187–202, 2003.
9. C. Ramchandani. Analysis of Asynchronous Concurrent Systems by Timed Petri Nets. *Project MAC-TR 120, MIT*, February 1974.
10. J. Sifakis. Use of petri nets for performance evaluation. In *Proceedings of the Third International Symposium on Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North-Holland, 1977.
11. Wil M. P. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In *Application and Theory of Petri Nets 1993, 14th International Conference, Chicago, Illinois, USA, June 21-25, 1993, Proceedings*, 1993.
12. M. Werner, L. Popova-Zeugmann, M. Haustein, and E. Pelz. A Holistic State Equation for Timed Petri Nets. *Fundamenta Informaticae*, 133(2-3):305–322, 2014.

Lookahead Consistency Models for Dynamic Migration of Workflow Processes

Ahana Pradhan and Rushikesh K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Powai, Mumbai-400076, India

Abstract. Dynamic migration of workflows requires the notion of consistency for safe migration. The literature primarily covers consistency models based on the history of the workflow to be migrated. However, for several situations, the history based models are not enough to decide migratability of a state. The paper introduces lookahead models of consistency, which are based on the question of how the remaining part of the workflow is treated in the new process. Three lookahead models are described and are illustrated with the help of example cases of realistic migration scenarios. Moreover, in certain situations, even if there is a consistent lookahead migration possible, the tokens can not be directly migrated into the new net due to contradictory traces available in the new net. The paper also proposes an algorithm called *Accept-Reject Branching* to compute the contradictory segment of the new net. A detail case study of a library resource acquisition workflow is presented to highlight the contributions.

Keywords: Workflow Migration, Consistency, Lookahead.

1 Introduction

In an ever-changing business environment the business processes do not remain static. Instances of one workflow often need to follow the evolved business logic of a different, or a refined schema. When an instance migrates from the old process into the new process the issue of consistency has to be addressed. The notion of consistency ensures that a migrated instance finishes execution without encountering a runtime error or inconsistencies in application semantics.

One of the initial works discussing consistency in dynamic evolution of workflows is by Ellis et al. [1]. They describe consistency criterion as the possibility of reproducing the execution history in the new schema. Moreover, mapping the old history in the new schema obtains the runtime state of the workflow from where it can resume to follow the new business logic. In this approach the consistency is based only on the past execution of the old instance. Contemporary works by Casati et al. [2], and Sadiq et al. [3] also adopt the same notion of consistency under the terminology of *compliance*. In the context of instance-specific ad-hoc dynamic changes, Reichert et al. [4] present the notion of consistency as a criteria that preserves the validity of the instance-specific workflow schema after

modification and suits the old execution history. Later researchers have adopted this same notion of history equivalence consistency under different terminologies and with subtle differences in the interpretation of history. The notions of *valid mapping* in the work of Weske [5], *migration conditions* by Dias et al. [6], several classes of *compliances* by Rinderle et al., [7], [8], by Sun and Jiang [9], and the *consistency criteria* in our previous work [10] are examples of history-based consistency.

Notable works using Petri net models of workflows, on the other hand, adopt the notion of consistency defined on the basis of marking. In Petri net models various process states are explicitly modeled by places. In this model a marking represents the current state of the process. Consistent markings in the old and the new workflow are decided based on the equivalence of states. This approach is taken in the works by Van der Aalst [11] and later by Circirelli et al. [12].

History based consistency model derives its motivation in the need to consider as done what is already accomplished and proceed exactly thereafter by resuming the workflow as per the new logic. Therefore, a primary goal of interest in history based migration is to ascertain the preconditions before migration takes place. However, apart from the models of history and state based consistency, dynamic workflow migration in the context of other business goals such as resource optimization, incidental migration, and handling eventualities often require a lookahead consistency criterion. Motivating examples of such dynamic migration scenarios are presented in this paper, on the basis of which, we develop the notion of lookahead consistency. A Petri net based modeling notation called WF-net [13] is used for representing workflows. The consistency models are defined using the WF-net notation, which can also be adopted in generic workflow terminology in a model independent manner. Three variants of lookahead consistency called strong, accommodative and weak lookahead are introduced.

An algorithm for generating weak lookahead consistent migrations for workflow instances is presented. It is possible that newer paths may be available in the new net for a migrating instance. In order to enforce a stricter lookahead, these paths need to be blocked, which can be done by blocking the head-transitions of the contradictory segments. These blocking transitions are identified using an algorithm called *accept/reject branching*.

The paper is organized as follows. Section 2 briefly outlines the preliminaries of Petri net based workflow model. Section 3 discusses the related work and the contributions of the paper. Sections 4, 5 and 6 discuss the three lookahead consistency models with the help of realistic application scenarios. Section 7 lastly discusses the algorithms for lookahead consistency along with a case study.

2 Notations

In this section a brief background is provided for WF-net notation of Van der Aalst [13], which is used in this paper as a reference formal notation for defining consistency and for developing the case studies.

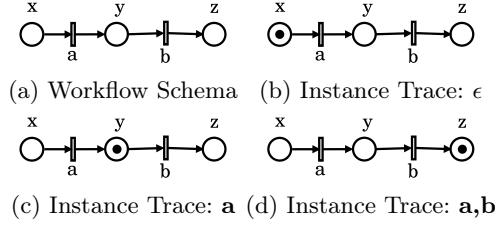


Fig. 1. WF-net Model of Workflow and Workflow Instances

Fig. 1a depicts an example WF-net schema of a sequence workflow involving two tasks **a** and **b**. Tasks are represented as *transitions* (rectangles), and conditions are represented as *places* (circles). A place models the precondition and postcondition of its immediate successor and predecessor tasks respectively. The *source place* captures the initial condition i.e. the start condition that triggers the start of the corresponding workflow. Similarly, the *sink place* models the terminal condition. A *token* (a dot in a place) makes up a *marked place*. If all pre-conditions of a transition are true, the transition is *enabled*. An enabled transition eventually *fires* by consuming one token from each of its pre-places (all pre-conditions), and produces one token in each of its post-places (post-conditions). A postcondition can be satisfied by firing of any one of its preceding transitions.

A placement of tokens in a net is called a *marking*. A marking represents the runtime state of an instance of a business process modeled by the net. A *firing* of a transition changes the marking. A *firing sequence* is a sequence of transition firings from one marking to another. The firing sequence from initial marking to the marking shown in Fig. 1b is empty. Fig. 1c shows the marking after firing of transition **a**, i.e. after completion of activity **a**, where the state of the workflow is *y*. Similarly, Fig. 1d shows the final marking of the workflow after firing of **b**.

The following notations are used to represent the dynamics of the net: A unit transition of marking $m_1 \xrightarrow{t} m_2$ means that firing of transition **t** changes the marking (state) of the net from m_1 to m_2 . In general a transition $m_1 \xrightarrow{\sigma} m_2$, where σ is a firing sequence $t_1 t_2 \dots t_n$ represents that the particular firing sequence σ changes the state of the workflow from m_1 to m_2 through some other states. Multiple firing sequences between two markings may be possible in nets with choice and concurrency. In the figure, $x \xrightarrow{a} y$ is an example unit transition, and $x \xrightarrow{ab} z$ is a transition from x to z through a longer firing sequence.

3 Related Work and Contributions of the Paper

The literature includes a variety of consistency criteria in the context of runtime migration of workflows [14], [7]. They are primarily history and state based approaches. For instance, the state based approach of *behavioral consistency* in the work of Casati et al. [2] looks into validity of the mapped state to ensure proper termination. Consequently their approach does not need to take the actual content and its variations in possible future execution traces into account.

An example of history based consistency is the approach of Ellis et al. [1], in which, the migrated instance state is required to have, starting from the initial marking, the same trace that was before migration.

The only work in this context, where the possible lookahead parameters were taken into account is the notion of *inheritance of workflows* by Van der Aalst and Basten [15]. In their approach, consistency is defined based on *branching bisimilarity* between the states of the old and the new workflows. As per the notion of bisimulation [16], equivalence is established between two states based on their future transitions and the states visited by those transitions. Branching bisimilarity considers inclusion of silent transitions in addition. Therefore, the adopted model of consistency in this work falls into the class of lookahead based model. However, adopting the notion of bisimulation defined on LTS (labeled transition system) states leads to much stronger criteria than intended in the context of process migration. The addressed domain of process migration does not consider a process to be interactive. In particular, the problem of dynamic instance migration addressed in this paper does not consider conversation or collaboration issues. Therefore, observational behavior of a process is its *trace*, and hence, equality relations based on the traces are sufficient to define consistency. Our work takes up this point to develop a range of consistency models that are based on lookahead traces.

Consulting lookahead parameters in the context of dynamic web-service protocol evolution has been identified as a necessary feature in the work of Ryu et al. [17]. They describe the notion of *forward compatibility* that is a property to be considered in the context of migrating web-service conversations. The forward compatibility captures the ability for the clients to interact with the dynamically evolved service after migration without confronting an *error*. Therefore, in order to save the ongoing conversations from failing, the lookahead parameters are vital. However, in contrast with the web-service conversation situation, as pointed out in the previous paragraph, the dynamic migration in the context of workflows are rather the changes in orchestration itself. Consequently, the need for consideration of lookahead parameters requires a solution with focus moving from interaction error to consistency in application semantics.

A benefit of these newly introduced models is that they can be used independently or in commune with the history based or state based consistency models as per the need of a particular workflow migration scenario. A lookahead based migration approach can then be applied considering the traces starting from the current state in the old workflow in the migration pair, and finding them in the new workflow to compute consistent migration.

The proposed lookahead models are demonstrated with the help of motivating cases. In the subsequent sections we define and illustrate three lookahead consistency models, which are *strong lookahead consistency*, *accommodative lookahead consistency* and *weak lookahead consistency* models respectively. The accommodative and the strong lookahead consistency models are specializations of the weak model, and the strong model is a specialization of the accommodative model.

4 Strong Lookahead Consistency

In this model, consistent states in the old and the new workflows are mapped by equivalence between possible futures in both the workflows. States of the old and the new workflows are consistent if the schedules that are yet to be completed from the old workflow state are the only schedules that are possible in the new workflow after migration. The strong lookahead model does permit changes to the net as long as the set of possible future traces is the same. This model can be applied to handle migration situations in process re-engineering cases, in which, from the point of view of the current state of the migrating instance, a change should not be perceived as far as the traces, i.e. the choices and the sequences thereby are concerned. In practice, such a situation may arise due to maintenance and compatibility issues. One such situation is illustrated later through a case study of a food packaging workflow example.

The WF-net based definition of the strong lookahead consistency model is given below. We use relational operator \blacklozenge between two markings to represent strong lookahead consistency between them. Subsequently relational operators \circ and \diamond are used to represent the accommodative and the weak lookahead models.

Definition 1 Let the old and the new workflows be modeled as WF-nets W and W' respectively, m_f and m'_f be the final markings in W and W' respectively, m be a marking in W , and m' be a marking in W' . Let $F_m = \{\sigma | m \xrightarrow{\sigma} m_f\}$, i.e. be the set of all firing sequences starting from marking m and reaching the final marking m_f in the old net. Similarly, $F'_m = \{\sigma' | m' \xrightarrow{\sigma'} m'_f\}$, i.e. be the set of all firing sequences starting from marking m' and reaching the final marking m'_f in the new net. Strong Lookahead consistency $m \blacklozenge m'$ is defined by the following trace equivalence condition: (i) $\forall \sigma \in F_m, \sigma \in F'_m$, and (ii) $\forall \sigma' \in F'_m, \sigma' \in F_m$. In other words, $m \blacklozenge m'$ is defined by the equality $F_m = F'_m$.

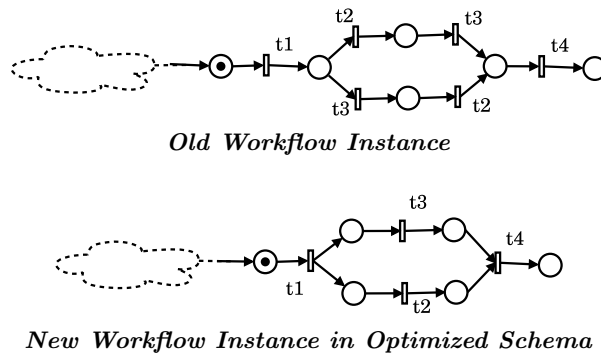


Fig. 2. Strongly Consistent Markings

It can be noted that to satisfy the above criteria, the structure of the downstream nets of the old and new workflows need not be the same. Intentionally different designs resulting into the same behavior may also fit to the definition. For example, a new net may be an alteration of its corresponding old net for achieving an optimization in the design as shown in Fig 2. In this example, the set of traces is $\{t_2t_3, t_3t_2\}$, which is same in both the nets. As the firings of transitions in Petri net semantics (reachability graphs[18]) are atomic and therefore instantaneous, a realization of parallel transitions t_2 and t_3 in the net results in two possible firing sequences t_2t_3 and t_3t_2 . Therefore, since the sets of all possible traces from the shown markings are the same, they satisfy strong lookahead consistency.

However, real-life workflows may have long duration tasks, in which, the semantics of non-overlapping atomic executions may not be possible. In such cases, the trace-based model can still be applied considering discrete events such as commencements or completions. For example, in an academic setup, if performing two courses in two consecutive semesters and performing them together in parallel in one semester needs to be considered as equivalent, it can be done so by considering the events marking the first lectures of the two courses. So, though there is physical interleaving of the task actions, if an academic process believes that the interleaving is acceptable as long as strong lookahead consistency is maintained w.r.t. the courses, a migration of a student from one system to another is possible. This assumption is useful in applying the lookahead models in non-Petri net workflow models such as ADEPT2 [19].

Fig. 3 shows packaging workflows for milk-products, chocolate and dry-fruits respectively. The packaging company imports the food items in bulk from various food processing companies and delivers them to distributors after packaging. The activities of devanning, storing items in the warehouse, quality inspection, and transport for delivery are manual activities. The fourth task in the dry-fruit packaging workflow is a manual task of inserting fruits by weight. Cutting of cheese and butter, food packaging in polythene or cardboard boxes, packet sealing and labeling are user assisted automated activities. As shown in the figure, in the case of dry-fruit, two kinds of packets are produced by the workflow using polythene packets or cardboard boxes. The workflow process is organized such that the packaging choice automatically alternates after regular intervals.

The packaging unit uses three different assembly-lines for the three food items. However, we can observe that polythene-based packaging of milk-products can use part of the assembly-line for the chocolate packaging once they reach the equivalent state (i.e. marking) p . Such an equivalent state is not available in the assembly-line of the dry-fruit packaging workflow due to the strictly alternating packaging designed feature. As the possible future execution sequences of the workflows for milk-products and chocolate packaging are exactly the same from the shown markings, a single assembly-line can take care of both of the packaging processes downstream the equivalent markings. The migration decision can be helpful when one assembly-line needs to be shut down for maintenance. Such a migration does not require any modification to the new process when strong

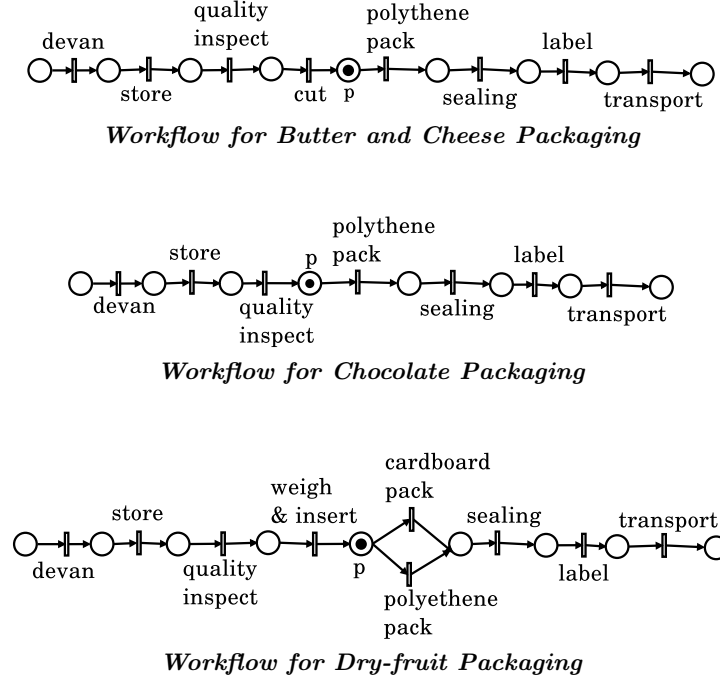


Fig. 3. Food Packaging Workflows

lookahead is established. As the alternating path of cardboard packaging in the dry-fruit assembly-line is not used for milk-products, the assembly-line is not suitable in this migration situation, in spite of the existence of the polythene packaging option inside the proposed new assembly.

The above example of dynamic workflow migration situation occurs in the context of resource maintenance, which is an assembly line in this case. It can be observed that the necessity of comparing the future of the running cases with the available assembly line is vital to finish the cases by dynamically migrating them. Clearly, history based consistency models do not serve a useful purpose, whereas a lookahead model captures the consistency requirement.

5 Accommodative Lookahead Consistency

This class of lookahead consistency notion is a weaker one as compared to the earlier class. The accommodative lookahead consistency can be defined to permit new alternatives which are not found in the old net, in addition to the existing traces. In this model, if a trace is possible in the old workflow, it is also possible in the new workflow. However, the converse is not required.

Definition 2 Following the terms m , m' , m_f , m'_f , F_m , F'_m as used in Definition 1, Accommodative Lookahead Consistency $m \diamond m'$ is defined by the following trace equivalence condition: $\forall \sigma \in F_m, \sigma \in F'_m$. In other words, $F_m \subseteq F'_m$.

From the definition we can note that the accommodative lookahead consistency between m and m' , i.e. $m \diamond m'$, satisfies the trace equivalence condition (i) of Definition 1. Therefore, strong lookahead consistency between two markings implies accommodative lookahead consistency between them as well, i.e. $m \blacklozenge m' \implies m \diamond m'$. Now we present a dynamic process migration scenario based on accommodative lookahead model.

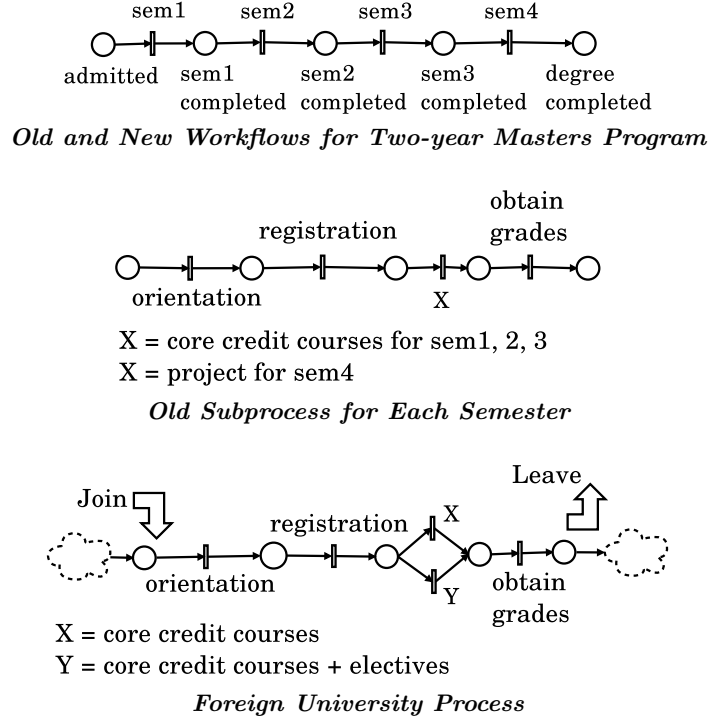


Fig. 4. Student Exchange Program

In an academic curriculum process depicted in Fig. 4, the old process in the migration situation represents a 4-semester masters program. Semesters are sequences of tasks involving orientation, registration, course work and grade reports. Course work comprises of credit courses in the first three semester, and a project in the last semester. The backlog credits are carried over into the next semester. In the process shown in the figure, the semester activities are firstly shown as single transitions of higher level, and they are expanded as a generic subprocess.

A migration situation arises when a student applies for an academic exchange program and joins a foreign university to replace a portion of her academics in the host institute. The student comes back and joins into the old process after completing the exchange credits.

The rules for the exchange program are outlined as follows:

- One student can join an exchange program to study in the foreign university in her second or/and third semester.
- The minimum cut-off CPI for availing the exchange program to join course work in a foreign university is 8.0.
- During the study in the foreign university, a student must complete the courses equivalent to the core credit courses in the home university.
- A student with CPI above 9.0 can join additional honors credit courses as electives.

The foreign subprocess that can replace sem-2 and/or sem-3 activities in the in the old 2-year program to avail the exchange program is highlighted in Fig. 4. It is only a part of a bigger process in the foreign university. A student is allowed to migrate if equivalent core credit courses are available in that semester. The list of courses offered at the foreign university website is to be consulted externally. Therefore, an application for migration are processed at states (markings) *sem-1 completed* or *sem-2 completed*. As the foreign university course work also offers the option of performing elective courses in addition to the core courses, the host institute permits the additional paths.

The above example brings us to the application of accommodative lookahead consistency at the point of migration. This migration scenario shows that a combination of past and lookahead parameters can also be used for deciding migratability. The set of new traces possible for a migrating student is a superset of the old. In this case, several historic or present parameters such as CPI, position in the academic calender are also used to determine the points of migration.

6 Weak Lookahead Consistency

The third kind of lookahead consistency model is the *weak* model. A state of the old workflow is consistent with a state in the new workflow by the weak lookahead model if at least one of the possible future traces is retained in the future of the new net. However, the future of the new net may have additional alternative traces.

Definition 3 *Following the terms $m, m', m_f, m'_f, F_m, F'_m$ as used in Definition 1, Weak Lookahead Consistency $m \diamond m'$ is defined by the following trace equivalence condition: For non-empty F_m , $\exists \sigma \in F_m$ such that $\sigma \in F'_m$. For empty F_m , $F_m = F'_m$.*

It can be observed that, quantifier $\forall \sigma \in F_m$ in Definition 2 is enough to find one such case required in Definition 3. Therefore, $m \ddot{\diamond} m' \implies m \diamond m'$. Moreover, we can obtain $m \blacklozenge m' \implies m \diamond m'$ from the already established relation $m \blacklozenge m' \implies m \ddot{\diamond} m'$.

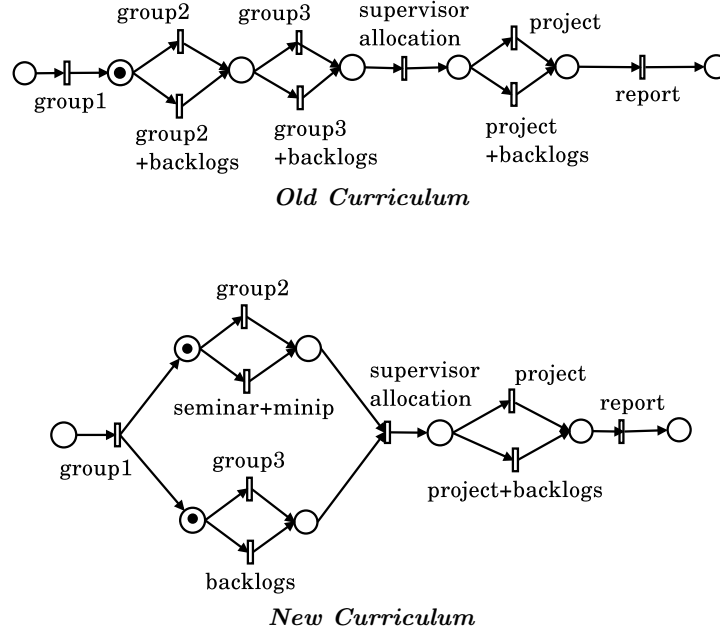


Fig. 5. Academic Curriculum Revision

Fig. 5 depicts an example case of academic curriculum revision of a 4-semester masters program. In the old curriculum, credits for three groups of courses are to be completed in the first three semesters. Alternate branches with backlog credits are available. The allocation of a project-supervisor is arranged before the final semester project work. The program ends with project reporting. The revised curriculum emphasizes on the depth in course work rather than the extent of the covered syllabus, by reducing the number of compulsory credits for the under-performing students. After completing group-1 courses in the first semester, the students have to perform two groups of courses. They can take up group-2 and group-3 courses as per the old curriculum, or they have the option of a seminar and a mini-project course. The students having backlogs can not register for group-3 courses.

A student in the old curriculum can migrate from current state into a state in the new curriculum, the design makes available at least one path in the new which is exactly a path in the old. However, new alternatives are also open to the migrating student. One set of equivalent markings by this criteria is shown in the figure. It can be noted that for the single token in the old workflow, two tokens are generated in the new workflow, which together constitute the consistent mapping. The new workflow preserves the traces *group2*, *group3*, *supervisor allocation*, *project*, *report* and *group2*, *group3*, *supervisor allocation*, *project+backlog*, *report* from the old workflow and not the others. Some traces of the old workflow are suppressed in the new workflow. In this way, the weak lookahead model can be applied to define flexible but at least minimally compatible workflows.

7 Instance Adaptations through Lookahead Consistency

In the previous sections, we have developed the lookahead consistency models and illustrated their applicability in several dynamic workflow migration situations. In this section, we first discuss the computation of weak lookahead consistent markings in the new net. Further, we discuss the approach of enforcing lookahead consistent execution on migrating instances in situations in which a weaker model is available in the new net. The *accept/reject branching* algorithm is then presented to support the lookahead consistency enforcement approach. The workflow nets considered in the following algorithms are considered to be acyclic. Moreover, all transition labels in the nets are assumed to be unique.

7.1 Algorithms for Lookahead Consistency

Algorithm 1 computes the weak lookahead consistent markings in the new net by replaying the lookahead traces possible in the old net. As only acyclic nets are considered, the set *Traces* of lookahead traces is finite. For a set of computed traces in the old net, all of them may not replay in the new net. The algorithm finds out the markings in the new net each of which can replay at least one of the traces. The outputs of the algorithm are set *S* of weak lookahead markings and set *L* containing the preserved lookahead traces in the new net.

Algorithm 1: Computation of Weak Lookahead Consistent Marking

Input: Old WF-net $N = (P, T, F)$, Marking M in N , New WF-net $N' = (P', T', F')$
Result: Set of Markings S in N' , Set of Preserved Lookahead Traces L

- 1 Let M_f and M'_f be the terminal markings in N and N' respectively
- 2 $Traces \leftarrow \{\sigma \mid M \xrightarrow{\sigma} M_f\}$
- 3 **if** $Traces = \{\}$ **then**
- 4 $S \leftarrow M'_f$
- 5 $L \leftarrow \{\}$
- 6 **return**
- 7 $Traces^r \leftarrow \{\sigma^r \mid \sigma^r \text{ is reverse of } \sigma, \sigma \in Traces\}$
- 8 $F'_{edit} \leftarrow \{(x, y) \mid (y, x) \in F'\}$
- 9 $N'_{edit} \leftarrow (P', T', F'_{edit})$
- 10 $S \leftarrow \{\}, L^r \leftarrow \{\}$
- 11 **while** $Traces^r \neq \{\}$ **do**
- 12 Let σ^r be a member of $Traces^r$
- 13 **if** $M'_f \xrightarrow{\sigma^r} M'_e$ in N'_{edit} **then**
- 14 $S \leftarrow S \cup M'_e$
- 15 $L^r \leftarrow L^r \cup \sigma^r$
- 16 $Traces^r \leftarrow Traces^r - \{\sigma^r\}$
- 17 $L \leftarrow \{\sigma \mid \sigma \text{ is reverse of } \sigma^r, \sigma^r \in L^r\}$

First it computes $Traces$, the set of lookahead traces in the old net. If set $Traces$ is empty, it indicates that the old net is already terminally marked. Therefore, the only lookahead consistent marking in the new net is the terminal marking M'_f . Also, since there is no lookahead trace in the old instance, the set of preserved lookahead traces in the new net is also empty. Therefore, the algorithm terminates here after computing these boundary outputs. Lines 3-6 handle this boundary case.

For non-empty $Traces$, it flips the member traces and stores them into set $Traces^r$. Lines 8-9 reverse the arcs directions in the new net N' and stores the reversed net as N'_{edit} . For each of the traces in $Traces^r$, the algorithm looks for its occurrence in N'_{edit} . Finally, set L contains those original lookahead traces that can be replayed in N' , and set S contains the weak lookahead consistent markings in N' .

The algorithm thus looks for all traces starting from the current marking in the old net, collecting all new markings corresponding to these traces in set S . Since, this collection is a set, a marking appears only once even though it can trigger more than one lookahead traces due to fork-join patterns. The set S can however have multiple markings in certain configurations as given in Fig. 6. For the example given in this figure, the algorithm starts with $Traces = \{t_1t_3, t_2t_3\}$ and ends with computing $S = \{\{p'_1\}, \{p'_2\}\}$, $L = \{t_1t_3, t_2t_3\}$.

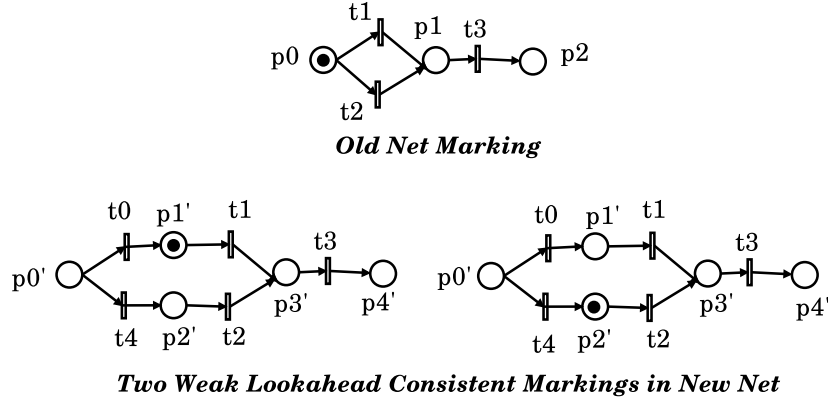


Fig. 6. An Example Case for Algorithm 1

7.2 Support vs. Enforcement of Lookahead Consistency

The motivation behind enforcing lookahead consistency is as follows. Given a marked old net and a new net schema, even though the lookahead consistent marking is supported in the new net, due to the generality of the new net, the

tokens can not be guaranteed to follow the execution path as prescribed by the old model. For example, consider two consistent markings are shown in the milk-product packaging and dry-fruit packaging workflows of Fig. 3 as per the weak or accommodative consistency model. As noted out earlier, since they do not satisfy the strong model, the assembly line of dry-fruit packaging can not take over the task of milk-product packaging. However, if we disable the traces in the new workflow which violate strong lookahead, the same can still be ensured for conforming to the desired post-migration paths. As a result, this approach achieves stricter consistency as intended for the migrating instance, which is not otherwise enforced by the new schema.

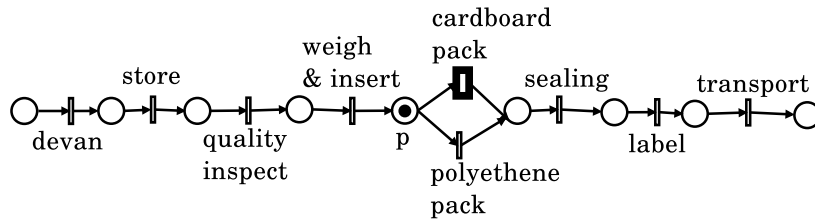


Fig. 7. Output of Accept/Reject Branching Algorithm

For the marking given in the milk-product packaging workflow in Fig. 3, Fig. 7 depicts the consistent marking and transition to be blocked to enforce strong lookahead consistency for the instance migrating into the dry-fruit packaging workflow. The only transition to be blocked in this case is shown as a box with thick border.

Next, the *accept/reject branching algorithm* given in Algorithm 2 identifies such transitions that need to be blocked to enforce the consistency preserving lookahead executions. It is assumed that a suitable implementation mechanism for disabling the transitions is available in the workflow management system.

Algorithm 2: Accept/Reject Branching

Input: WF-Net $N = (P, T, F)$, Marking M in N , Set of lookahead traces Σ

Result: Set of transitions T_{block}

- 1 let $path(e_0, e_0) = \text{TRUE}$
- 2 let $path(e_i, e_j)$ be a boolean function indicating the existence of directed path from net element (place or transition) e_i to net element e_j
- 3 $P_{exchoice} \leftarrow \{ p \mid (p, t_i), (p, t_j) \in F, i \neq j, \exists p_0 \text{ such that } M(p_0) = 1, (p_0, p) = \text{TRUE} \}$
- 4 $T_{potential} \leftarrow \{ t \mid t \in T, p \in P_{exchoice}, (p, t) \in F \}$
- 5 $T_{lookahead} \leftarrow \{ t \mid \sigma \in \Sigma, t \in T_{potential}, t \in \sigma \}$
- 6 $T_{block} \leftarrow T_{potential} - T_{lookahead}$

The inputs to the algorithm is a marking in the new net and the set of desired lookahead traces Σ which is set to L , the output of Algorithm 1. As output it produces the set of transitions T_{block} which can be blocked to disable all the lookahead traces that are not in Σ .

First the algorithm identifies the *choice gateways* which are not yet traversed by the tokens in the new net. In a sequence, disabling the head-task prevents the whole sequence. Paths following exclusive-choice gateways are sequences with head-tasks as the first tasks after the choice. The algorithm finds out the set of tasks $T_{potential}$ which holds all the head-tasks following the choice gateways in the new net. Discarding those tasks from $T_{potential}$ which are not in the lookahead traces gives the remainder portion of the net which should be left as active. In this way, the algorithm finds out the tasks to be blocked as the set T_{block} , which is $T_{potential} - T_{lookahead}$.

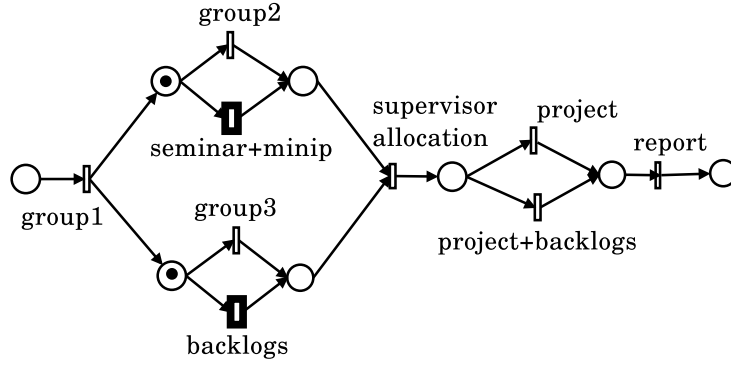


Fig. 8. Blocked transitions in the new workflow shown in Fig. 5

It can be noted that blocking of lookahead transitions for the migrated instances may not be an appropriate solution in a given application. For instance, consider the case of migrated instance in Fig. 8 of the academic workflow process shown in Fig. 5. The blocked transitions are shown as boxes with thick border. As a result of blocking these transitions, the migrating student can proceed only through *group2* and *group3* courses. However, due to the additional constraint of the university that a student having backlogs can not register for *group3* courses, such students can not be migrated since there is no path left for them in the new net. Also, migrating students can not take seminar and mini-project courses available in the changed curriculum. Therefore, with reason of retaining flexibility, blocking of transitions is not suitable for this situation. On the contrary, the approach of blocking transitions is suitable for those cases where migration is inevitable, and yet sticking to the old execution paths is necessary for application semantics. A case study of a library resource acquisition workflow given in Section 7.3 describes one such suitable migration scenario for accept/reject branching.

The following conditions summarize the inferences regarding the class of lookahead consistency that can be drawn from the outputs of the two algorithms.

1. $L \neq \{\}$ confirms weak lookahead consistency. (Algo. 1)
2. In addition, $|S| = 1$ and $L = \text{Traces}$ confirms accommodative lookahead consistency. If S contains more than one markings, no single marking can fire all lookahead traces since there are no duplicate transitions. Hence, the condition $|S| = 1$. (Algo. 1)
3. In addition, $T_{block} = \{\}$ confirms strong lookahead consistency. (Algos. 1, 2)
4. $S = \{\}$ implies absence of lookahead consistency.
5. If $T_{block} \neq \{\}$, blocking of the transitions in set T_{block} is a mechanism to enforce the desired lookahead traces. This converts accommodative lookahead to strong lookahead. In the case of weak lookahead, the lookahead traces found can indeed be enforced by blocking these transitions.

7.3 Case Study of a Library Resource Acquisition Workflow

A library resource acquisition workflow case is considered for this case study. Processes in this system are orders of the kind *firm orders*. (There are other types of library acquisitions such as serial subscriptions, standing orders and blanket orders which are not considered in the case study.) In the old system, the institute has separate processes of resource acquisition for the academic departments and for the central library as shown in Fig. 9a and Fig. 9b respectively. It is proposed to merge these processes into a single one. We first describe the old processes after which the new process is outlined. After this, a consistent lookahead based migration solution is worked out.

The Old Departmental Process The departmental process can be followed only for acquisition of hard-copies. First the bibliographer has to prepare the list of books to be purchased. The overall expense is then estimated and an application is sent next to the department office for approval of the budget. Arrival of the funding approval initiates the negotiation procedure with the vendors. In the case of rejection of the funding application from the department office, the workflow can proceed in one of the two ways. If the applicant can arrange money from her/his project funds, the workflow can proceed to join the flow of usual acquisition process by initiating the price negotiation. Otherwise, in the case of unavailability of funding, the acquisition case is dropped. Next, the payment is carried out for the agreed price as a confirmation of the purchase order to the vendor. Delivery of the books along with the invoice completes the resource acquisition. The workflow finishes after recording the acquisition details in the department resource database and with cataloging of the acquired resources.

The Old Central Library Process The central library workflow follows similar logic for hard-copy resource purchase, though the funding agencies are different. All acquisition requests are sent to the academic office for funding approval.

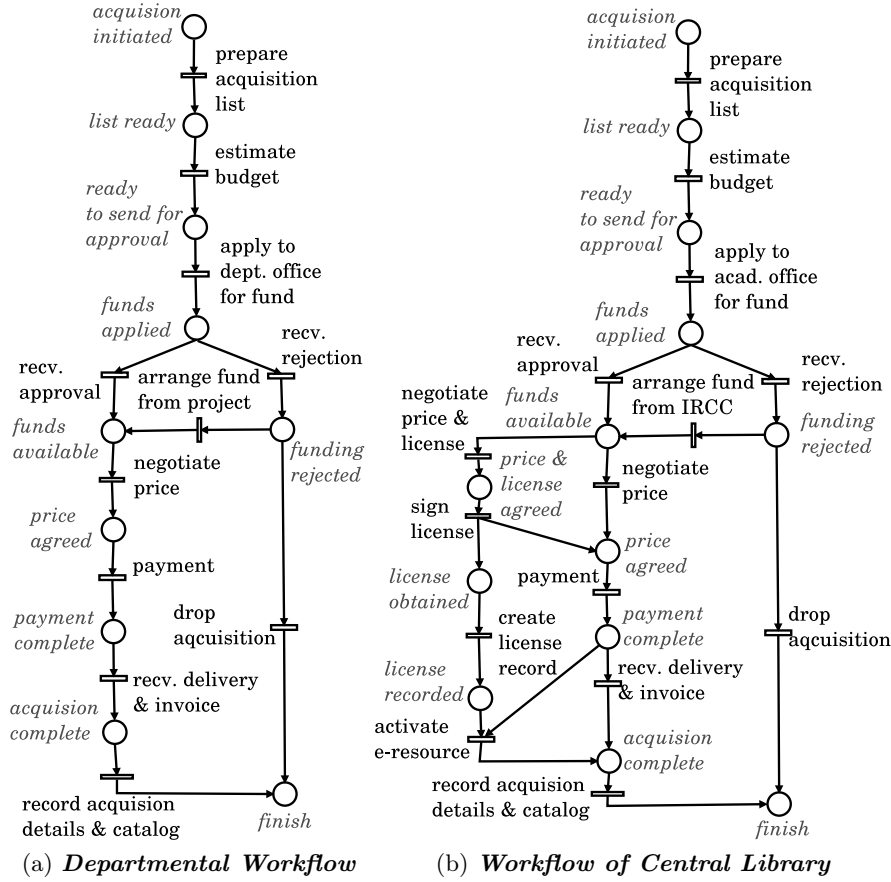


Fig. 9. Library Workflows for Resource Acquisition

In the case of rejection, either the Industrial Research and Consultancy Center (IRCC) supports the funding issues, or the case has to be dropped. In addition to the above, the central library workflow supports purchases of e-resource. In the case of e-resource acquisition, additional activities for license negotiation and agreement are incorporated in the workflow. Once the license is signed, its copy is received by the library which is recorded in the central library database. After payment, the e-resource is activated. According to the license agreement, this step can involve storing a local copy of the resource or enabling password protected access of the document residing on its remote host. Cataloging of the resource is performed on the central library database, which wraps up the process.

Process Re-engineering A process re-engineering team decides to merge all the departmental workflows with the central library workflow due to the following reasons.

- A consolidated storage of the library resources across all the academic departments can achieve better resource sharing among several departments, to the benefit of multi-disciplinary studies and projects.
- E-resource acquisition can be performed using the departmental or individual project funds introducing a level of flexibility.
- Individual departments can leave the responsibilities of library-staff recruitment and related activities to the central library authority reducing the redundant efforts in the old system.

As a result, the department resource databases has to be merged with the central library database. Secondly, the running instances need to be migrated in a consistent way by applying appropriate models as per the needs of individual old cases. It is observed that the task *record acquisition details & catalog* in the old process updates department resource database, whereas, in the new merged process it updates the central library database. This requires migration of all old incomplete instances into the new schema.

Lookahead Based Consistent Dynamic Instance Migration Fig. 10a shows such an on-going workflow instance that is in state *funds available*. The migration scenario requires weak lookahead consistency criterion for the safe migration of the running instances in order to complete the hard-copy acquisition process from the department. The merged workflow schema and the migrated marking is shown in Fig. 10b.

In addition to the migration, the situation requires that the same path be enforced for the migrating instance without giving the additional flexibility of the alternative of e-resource purchase. Therefore, the execution of these running cases must be prevented from traversing the path for e-resource purchase, which is achieved by blocking the transitions which are output of the accept/reject branching algorithm. The traces of both the algorithms are given below.

- $Traces = \{ negotiate\ price, payment, recv.\ delivery \ \& \ invoice, record\ acquisition\ details \ \& \ catalog \}$.
- $L = \{ negotiate\ price, payment, recv.\ delivery \ \& \ invoice, record\ acquisition\ details \ \& \ catalog \}$.
- $S = \{ funds\ available \}$.
- $P_{exchoice} = \{ funds\ available, payment\ complete \}$.
- $T_{potential} = \{ negotiate\ price, negotiate\ price\ and\ license, recv.\ delivery \ \& \ invoice, activate\ e-resource \}$.
- $T_{lookahead} = L$.
- As a result, $T_{block} = \{ negotiate\ price\ and\ license, activate\ e-resource \}$.
- It can be noted that, $L \neq \{\}$ ensures weak lookahead consistency. Further, $|S| = 1$ and $L = Traces$ ensures accommodative lookahead consistency. However, $T_{block} \neq \{\}$ implies that strong consistency is not supported by the modified workflow schema.

The transitions in set T_{block} are shown as boxes with thick border in Fig. 10b. These transitions can be blocked in the new workflow execution environment

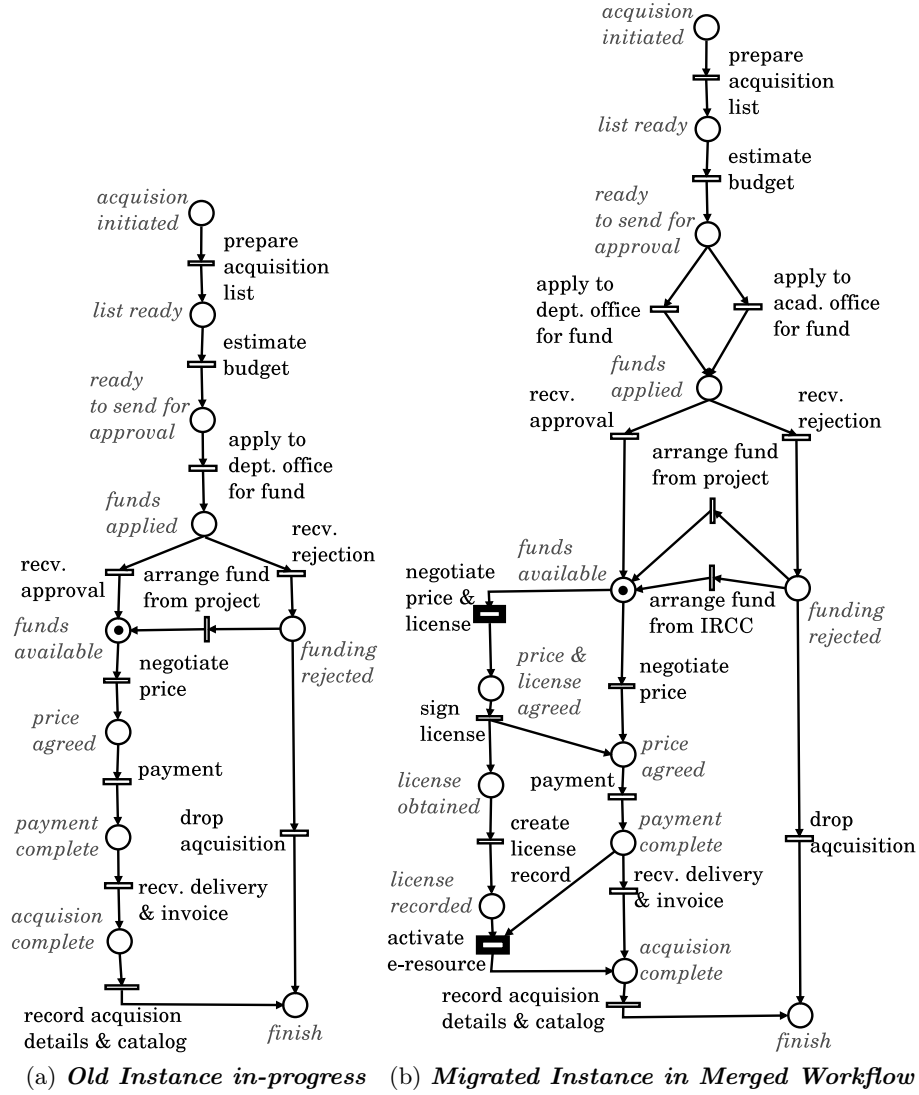


Fig. 10. Instance Migration into the Re-engineered Workflow

only for this migrating instance to enforce its required lookahead consistency as discussed above.

8 Discussion and Conclusion

In the literature, execution history based consistency notions are widely adopted wherever dynamic evolution and adaptation of workflows are considered. The proposed new model of lookahead consistency uses the remainder of the workflow instead of the completed history in defining consistency between states in

a migration scenario. Three broad classes of the lookahead consistency were brought out with illustrative examples. The weak lookahead model looks for preservation of at least one lookahead trace of the old instance across the new net. For accommodative lookahead model, preservation of all lookahead traces of the old instance is necessary. The strong model requires the lookahead traces of the old instance and its migrated marking in the new net to be same. Strong lookahead implies accommodative lookahead, which in turn implies weak lookahead consistency.

Besides their usefulness and the inter-relationships among the models, two related algorithms were also presented for deciding lookahead consistency, computing token transfer, and for lookahead trace enforcement. The new set of lookahead models provide future-centric approaches of varying flexibility for token transfer in dynamic migration scenarios with applications to process re-engineering and maintenance. A case study of library resource acquisition workflow demonstrated the relevance of the approach.

As the past execution traces are not taken into account in the context of lookahead based consistency models, these can not be used stand-alone in situations requiring history equivalence. However, the lookahead models can also be applied in combination with history equivalence as per the needs of business goals. The algorithms for acyclic nets were implemented in GNU Octave [20]. We are working on an integration of this implementation with the facility of blocking the unintended transitions in a workflow engine [21] environment, and an extension of the ideas to work with cyclic nets.

References

1. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: Proceedings of conference on Organizational computing systems, ACM (1995) 10–21
2. Casati, F., Ceri, S., Pernici, B., Pozzi, G.: Workflow evolution. *Data & Knowledge Engineering* **24**(3) (1998) 211–238
3. Sadiq, S.W.: Workflows in dynamic environments - can they be managed? In: In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications. (1999) 27–28
4. Reichert, M., Dadam, P.: Adeptflex—supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems* **10**(2) (1998) 93–129
5. Weske, M.: Formal foundation and conceptual design of dynamic adaptations in a workflow management system. In: System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on. (Jan 2001) 10 pp.–
6. Dias, P., Vieira, P., Rito-Silva, A.: Dynamic evolution in workflow management systems. In: Database and Expert Systems Applications, 2003. Proceedings. 14th International Workshop on, IEEE (2003) 254–260
7. Rinderle-Ma, S., Reichert, M., Weber, B.: Relaxed compliance notions in adaptive process management systems. In: Conceptual Modeling-ER 2008. Springer (2008) 232–247

8. Reichert, M., Rinderle-Ma, S., Dadam, P.: Flexibility in process-aware information systems. In: Transactions on Petri Nets and Other Models of Concurrency II. Springer (2009) 115–135
9. Sun, P., Jiang, C.: Analysis of workflow dynamic changes based on petri net. Information and Software Technology **51**(2) (2009) 284 – 292
10. Pradhan, A., Joshi, R.K.: Token transportation in petri net models of workflow patterns. In: Proceedings of the 7th India Software Engineering Conference. ISEC '14, ACM (2014) 17:1–17:6
11. van der Aalst, W.M.: Exterminating the dynamic change bug: A concrete approach to support workflow change. Information Systems Frontiers **3**(3) (2001) 297–317
12. Cicirelli, F., Furfaro, A., Nigro, L.: A service-based architecture for dynamically reconfigurable workflows. Journal of Systems and Software **83**(7) (2010) 1148 – 1164
13. van der Aalst, W.M.: The application of petri nets to workflow management. Journal of circuits, systems, and computers **8**(01) (1998) 21–66
14. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems—a survey. Data & Knowledge Engineering **50**(1) (2004) 9–34
15. van der Aalst, W.M., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. Theoretical Computer Science **270**(1) (2002) 125–203
16. Milner, R.: Communicating and mobile systems: the pi calculus. Cambridge university press (1999)
17. Ryu, S.H., Casati, F., Skogsrud, H., Benatallah, B., Saint-Paul, R.: Supporting the dynamic evolution of web service protocols in service-oriented architectures. ACM Transactions on the Web (TWEB) **2**(2) (2008) 13
18. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
19. Reichert, M., Weber, B.: Enabling flexibility in process-aware information systems: challenges, methods, technologies. Springer Science & Business Media (2012)
20. Eaton, J.W., Bateman, D., Hauberg, S.: GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations. CreateSpace Independent Publishing Platform (2009) ISBN 1441413006.
21. Pradhan, A., Joshi, R.K.: Architecture of a light-weight non-threaded event oriented workflow engine. In: The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014. (2014) 342–345

Catalog-based Token Transportation in Acyclic Block-Structured WF-nets

Ahana Pradhan and Rushikesh K. Joshi

Department of Computer Science and Engineering
Indian Institute of Technology Bombay, Powai, Mumbai-400076, India

Abstract. The problem of workflow instance migration occurs during dynamic evolutionary changes in processes. The paper presents a *catalog-based* algorithm called the *Yo-Yo Algorithm* for consistent instance migration in Petri net workflow models. It uses a technique of folding and unfolding of nets. The algorithm is formulated in terms of *Colored Derivation Trees*, a novel representation of the runtime states of workflow nets. The approach solves the problem for certain types of changes on acyclic block-structured workflow nets built in terms of primitive patterns moving much of the computation to schema level on account of the use of two critical ideas, a catalog and the folding order. The approach is illustrated with the help of examples and comments on its correctness.

Keywords: Block structured Workflows, Dynamic Evolution, Structural Compatibility, Token Transportation, Workflow Specification

1 Introduction

Organizational goals are realized by executing business processes that involve people, resources, schedules and technology. In order to cope up with changing environments, changing requirements or new internal challenges, business processes need to be changed. Traditional workflow management systems (WFMS) are well-suited for rigid processes. However, the volatile nature of business processes requires intricate facilities for changing the workflows at runtime in WFMS in a valid and consistent manner. In absence of this support the information system susceptible to changes needs to be tackled by slower porting processes, it not being immediately usable due to the not easily bridgeable gap between *the pre-planned* and *the evolved actuality*.

An evolutionary change includes process change at schema level and also instance migration for all running cases. This paper describes the *Yo-Yo* algorithm for Petri net models of acyclic block-structured workflows to carry out consistent runtime instance migration in this context. At the schema level a Yo-Yo compatibility property is specified to define the scope of the proposed instance migration algorithm. A specialty of the algorithm is that it gives the consistent token transportation based on pre-computed catalog solutions. Secondly, from the two workflow net schemas, we are able to separate immediately migratable and immediately not migratable markings. This work uses the Petri

net based workflow model called WF-net, which was introduced by Van der Aalst [1]. We follow a block-structured formulation of WF-nets, in terms of blocks, which are primitive workflow patterns namely the Sequence, the Parallel Block and the Exclusive-choice Block. The algorithm is presented for carrying out runtime token transportation under a set of change operations, which are the inter-convertibilities among the primitive pattern blocks. The intuition of this algorithm is presented in our earlier work [2]. The paper provides the full formulation of the algorithm and its proof of correctness. The formulation is developed in terms of a new runtime workflow state representation called the Colored Derivation Tree.

The paper is organized as follows. After discussing the related work, we first present our pattern based block structured workflow specification approach in Section 3. Following this, the novel representation called *Derivation Tree* and its colored form are developed in Section 4. In Section 5, the intuition behind the algorithm is first outlined. The ingredients of the algorithm are then discussed developing the notions of the *Yo-Yo compatibility* property between two nets, instance level correctness of migration in the form of a valid and consistent catalog of token transportation and folding, unfolding operations on WF-net. Lastly, the algorithm, its working and its applicability are explained with the help of a practical example scenario. A proof of correctness of Yo-Yo algorithm is given in Section 6. The algorithm works on colored derivation tree representing the old net and produces colors in the derivation tree of the new net, i.e. colors corresponding to the marking in the new net. The catalog is used for color transfer at each iterative step in the algorithm.

2 Related Work & Contributions of the Paper

In this section, we present a brief account of existing research on dynamic evolutionary changes in Petri net models of workflows to highlight the achievements so far and respective limitations. The literature in this field can be categorized into two kinds. Firstly, the *change region* based approaches are designed to work with arbitrary structural changes. The second category is of *state based* approaches.

2.1 Change Region Based Approaches

An earlier among the change region based approaches is the approach of Ellis et al. [3], representing dynamic change as replacement of a part of the old net and preserving the same history of execution by an altered making after the replacement. In their approach called *token transfert*, the old part referred to as the *old change region* is replaced by a new change region. However, some cases are unsafe to migrate when the old change region is marked, and hence, the transfers are delayed until the tokens in the old net reach a safe state. This delayed execution of changeover requires migration to be withheld till a consistent point of execution is reached when tokens come out of the change region. This early work in the field does not suggest a method for identification of unsafe

change regions or an algorithm for consistent migration. Ellis and Keddera [4] demonstrate realization of *token transfert* with the help of transitions called *flow jumpers* connecting places in the old and the new nets. However, no algorithm to compute the flow jumpers has been suggested in this work.

Aalst [5] presents an algorithm for computing change regions in old and new nets, the notion of which was introduced earlier [3]. Outside the change regions, the marking in the old net can be carried forward into the new net one on one without violating validity. However, the adopted consistency criterion uses a notion of validity based on marking equality in terms of place labels, ignoring the notion of consistency in terms of history equivalence. Therefore, for several types of changes, the computed change region does not ensure consistent migration.

Sun and Jiang [6] present a variation of the algorithm given by Aalst [5] for generating the change region. Their work handles dynamic changes for *upward compatibility*, where the behavior of the old net in terms of execution traces is preserved in the new net. For consistent instance migration, a weaker version of history equivalence criteria is specified. Unlike usual notion of state in Petri net formalism, in addition to marking, this work represents the runtime state by considering execution trace. The work also formulates a property for migratability at the level of instance, including those inside change regions. However, this approach does not provide an algorithm for consistent instance migration.

The work of Cicirelli et al. [7] describes an implementation and a case study of dynamic evolution based on the theory founded in the works of Ellis et al. [3] and Van der Aalst [5]. Their work uses the change region generation algorithm given by Van der Aalst to compute the unsafe regions for instance migration. The migration strategy is termed as *decentralized migration*, since the executions in different parallel branches are independently inspected and set for migration. The tokens in the old instance are then tagged according to their presence inside or outside the change region. In a particular state set for migration, some tokens may be inside the change region, whereas some are outside. Tokens outside change region are migrated immediately. Tokens inside change region continue till they come out of it and enter in a safe state suitable for migration. That point of execution creates a valid marking in the new schema.

2.2 State Based Approaches

The difference between this category and the earlier one is that the state based approaches do not pre-compute the change regions. Instead, they directly provide state based mappings in the new net. If consistent mapping does not exist in a particular state, this approach can not make use of any possibility of *delayed migration* as in change region based approaches.

It has been noted [6] that the change region based approach is a pessimistic approach since inside a change region, there may be migratable markings. In the state based approach, this drawback is removed with the additional cost of instance based solutions. Another shortfall of the existing change region computation algorithms is that they overlook the history equivalence criteria. For example, a change region computation focuses on finding a mapping for state

(p_1, p_2, p_3) to state (p'_1, p'_2, p'_3) ignoring one-to-many or many-to-one mappings of other kinds such as mappings to states (p'_1, q) , (p'_1, p'_2) or (p'_1, p'_2, p'_3, p'_4) in this case which have the same history of transition firings. The class of state based approaches solve this problem by keeping the observable behavior of the nets in focus, thereby exploring richer markings which need not be identical.

The approach of Agostini and Michelis [8] implement the feature of dynamic change in their MILANO workflow system. This work allows a set of change operations, which are *parallelization*, *sequentialization* and *swapping*. The mappings of runtime states between the old and the new workflows are precomputed over the entire state space modeled as *reachability graph*. Instead of identifying regions, state to state mappings are generated for valid migration points.

Van der Aalst and Basten [9] have looked into the problem of dynamic change in light of *inheritance* relations between the nets in a migration pair. If the new workflow *specializes* the *observable behavior* of the old workflow by *hiding* or *blocking* some of the additional tasks, then the new workflow is considered as a subclass of the old one. They show that if two nets are related by inheritance, it is always possible to have a correct instance migration from the old to the new workflow. The work also shows that addition or deletion of cycle, sequence, parallel or choice branches preserve inheritance relation. The mapping between the runtime states of the two processes is given by transfer rules guaranteeing *soundness*. However, the problem of consistency in terms of history is not formally addressed, though the authors point out a supporting example.

2.3 Contributions of Our Work

The paper presents an algorithm for token transportation to ensure the consistency criterion of history equivalence by applying catalog solutions without replaying the history, unlike most of the existing approaches. For a practical scenario of evolution, where thousands of instances need to be migrated, replaying history for each of them or solving the *state equation* [10] along with the solution for legal firing sequence problem [11] may be computation intensive. The Yo-Yo algorithm improves the runtime by pre-computing *migrations among primitive patterns*, and by generating what is called *Yo-Yo compatible derivation trees* at the schema level. Moreover, for the chosen types of nets and change patterns, the Yo-Yo algorithm successfully carries out consistent migration even for those cases many of which are not suitable for migration as per the change region based approaches due to their pessimistic prediction of non-migratability. Yo-Yo approach does not compute change region, but in turn, it looks for catalog based transportation which succeeds if the case is migratable by the consistency criteria of history equivalence.

3 The Pattern Based Approach of Workflow Modeling

The Yo-Yo token transportation approach offloads some of the complexities incurred by traditional change region based or state based approaches by means of

its block structured workflow specification. Confining the scope to block structured workflows is in line with the philosophy of block structured executable process models in BPEL [12] and pattern based models such as [13].

The permissible change operations on workflow schemas have been referred to as *change patterns* in the literature [14]. The change patterns in the Yo-Yo approach are *inter-convertibilities among the primitive patterns* shown in Fig. 1. The following six kinds of pattern changes are considered: SEQ to AND, AND to SEQ, SEQ to XOR, XOR to SEQ, AND to XOR, and XOR to AND.

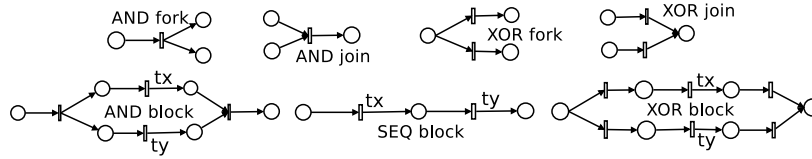


Fig. 1. Primitive Gateways and Patterns

3.1 Workflow Primitives

Patterns are commonly occurring configurations in architecture. Control flow behavior of patterns in workflow processes were described by Aalst et al. [15]. In our work, we formulate and use a grammar for workflow nets in terms of the primitive workflow patterns. Fig. 1 shows the Petri-net models of primitive fork-join gateways and pattern blocks. It can be noted that the transitions in the gateways are kept unlabeled, since these are used only to model the control-flows and not the workflow tasks. Consequently they are omitted from the specifications in the string based language of WF-nets which is introduced below. The string based language captures the control flow dependencies through delimiters.

3.2 CWS: A Compact Block Structured Workflow Specification

Block-structured workflows are composed by nesting the primitive patterns. This approach simplifies complex processes in terms of blocks. The block structures are directly folded in or out in the Yo-Yo algorithm. For the purpose of our work we assume that there is no repetition of transition-labels in a net.

Now, a compact string-based specification language called CWS is introduced for specifying block structured acyclic WF-nets. Unlike graphical and tuple based existing description methods for Petri net based workflows, in CWS, the places are dropped and only the labeled transitions are included. The reason for excluding the places is that the consistency criterion based on task execution traces does not require any role from the places. However, places shown in the pictorial models can be regenerated by parsing CWS specifications. The execution control transitions used in fork-join patterns are implicitly encoded into the delimiters, and only the application transitions are included in the specification.

$Start \rightarrow SEQ;$ $SEQ \rightarrow SEQ \mathbf{t} SEQ \mathbf{t} SEQ \mid SEQ \text{ AND } SEQ \mid SEQ \text{ XOR } SEQ \mid \epsilon;$ $AND \rightarrow (SEQ \mathbf{t} SEQ) (SEQ \mathbf{t} SEQ);$ $XOR \rightarrow [SEQ \mathbf{t} SEQ] [SEQ \mathbf{t} SEQ];$	
Workflow Net	CWS Specification
SEQ block in Fig. 1	$t_x t_y$
AND block in Fig. 1	$(t_x)(t_y), (t_y)(t_x)$
XOR block in Fig. 1	$[t_x][t_y], [t_y][t_x]$
The net in Fig. 8(a)	$t_1(t_2(t_3)(t_4))((t_5)(t_6)t_7)t_8$
The net in Fig. 8(b)	$t_1 t_2 t_3 t_4 (t_5)(t_6) t_7 t_8$

Fig. 2. CWS Grammar, Example Nets, and their Specifications

The CWS grammar is shown in Fig. 2. A terminal symbol \mathbf{t} represents a transition corresponding to a task in the workflow. Round and square bracket pairs are used to mark *AND* and *XOR* fork-join patterns respectively. The top level pattern is always a Sequence that can generate either an empty string or a nesting of blocks. Example nets specified in CWS are given in Fig. 2. It can be seen that parallel or choice branches can be specified in any order, which creates multiple equivalent specifications.

4 Derivation Tree of a Workflow Net

Parsing of workflow models into hierarchical blocks has been implemented earlier in the approach of Refined Process Structure Tree (RPST) [16]. It provides unique parsing of a WF-graph in terms of canonical single-entry-single-exit regions which can be of arbitrary length. However, this approach results in an infinite-sized catalog, which counters the advantage of our approach.

For Yo-Yo algorithm, the nets are required to be parsed in a hierarchy of fixed-size ingredient blocks. This parsing obtains the Derivation Tree representation of a WF-net. The derivation tree is obtained after cleaning up the delimiters from the CWS parse tree. For the primitive nets shown in Fig. 1, their respective parse trees and derivation trees are shown in Fig. 3. Table 1 shows the correspondence between symbols in the derivation tree and WF-net.

The terminals in the parse tree are application transitions, delimiters and empty sequences; and the non-terminals represent block-structured configurations in the net. The derivation tree excludes delimiters and empty sequences resulting in *leaf non-terminals* (e.g. n_i and n_x''' in Fig. 3f).

The child nodes of *AND* and *XOR* non-terminals are organized into two *triplets* representing the two fork-join branches. Each triplet contains two places and a transition. The arcs in a triplet are ordered left to right, showing a transition sandwiched between pre- and post-places respectively.

A derivation tree can be viewed as a hierarchical composition of the *derivation tree patterns*, which are the derivation trees of the primitive patterns (Figs. 3b,d,f). An Example of derivation tree patterns in a bigger non-primitive net appears in Fig. 5, where the derivation tree patterns are marked as dotted ellipses.

In a derivation tree pattern, the arcs coming out of a *SEQ* node are ordered from left to right. A *SEQ* node representing only a sequence of two transitions has five arcs, two for the transitions and three for the places. A *SEQ* node representing the grammatical reduction of an *AND* or an *XOR* branching has three arcs to connect to the branching and the places before and after it.

Table 1. Mapping of Derivation Tree Symbols

Derivation Tree Element/CWS Element	Symbol Used	Correspondence with WF-net
Leaf non-terminal/empty <i>SEQ</i>	Unfilled circle	Unfolded Place
Non-leaf non-terminal/ <i>SEQ</i>	Unfilled circle	Abstraction (Folded place)
Non-leaf non-terminal/ <i>AND</i>	Circle marked \wedge	Two parallel branches
Non-leaf non-terminal/ <i>XOR</i>	Circle marked \times	Two exclusive-choice branches
Terminal/ <i>t</i>	Symbol t_{label}	Labeled transition t_{label}

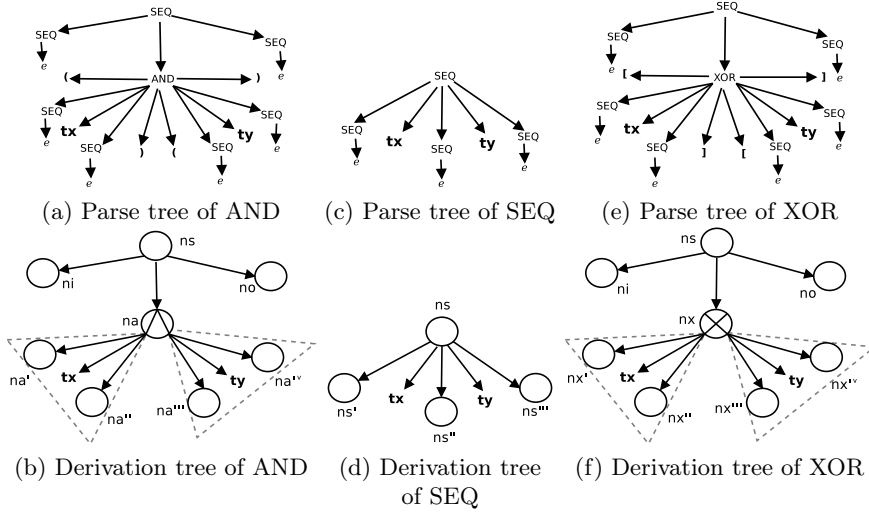


Fig. 3. CWS Parse Trees and Derivation Trees of Primitive Patterns

4.1 Yield of a Non-terminal

Yield of a non-terminal is a *sequence* obtained by depth-first traversal on the terminals in the entire subtree rooted at the non-terminal of a derivation tree, where elements of the sequence are terminals or sets of terminals which can be further nested. During the traversal, swapping the traversal order of triplets under an *AND* or *XOR* node does not alter the yield. Hence, yield of an *AND* or *XOR* node is formulated as a set of yields of the two triplets. Operator $yield(n)$ generates this traversal for a non-terminal n . For example, $yield(n_s) = \{t_x, t_y\}$ in Fig. 3b, $yield(n_s) = t_x t_y$ and $yield(n'_s) = \epsilon$ in Fig. 3d, yield of the root in Fig.

5a is a sequence with one element $\{t_1 t_2 t_3, t_4 t_5 t_6\}$, and the root of the derivation tree for the net given in Fig. 8a has yield $t_1 \{t_2 \{t_3, t_4\}, \{t_5, t_6\} t_7\} t_8$.

4.2 Local Terminal Coverage of a Pattern

To recall, a pattern is any of the tree structures shown in Figs. 3(b,d,f). The notion of *local terminal coverage* (LTC) defined on patterns establishes pattern to pattern correspondence called *peers* in two nets. LTC of a pattern p , i.e. $LTC(p)$, is a cross-product $s \times o$ of set s of terminals in the pattern and boolean value o indicating whether the set is ordered (i.e. *SEQ* block). The individual elements can be accessed through a dot operator as $p.s$ and $p.o$.

Peer Patterns: Two LTCs can be compared by comparing their terminal sets and the ordering. The comparison operator (equality) called *peer* is defined as follows. Let $=_s$ be the set equality operator and $=_o$ be the ordered set equality operator. We can define the comparison operator $peer(p, q)$ for two patterns p, q in terms of their respective LTCs, as an operator returning a boolean value: $peer(p, q) = ((p.o \wedge q.o) \wedge (p.s =_o q.s)) \vee (\neg(p.o \wedge q.o) \wedge (p.s =_s q.s))$. If both sets are ordered (first part of the disjunction), then the comparison operator checks for element ordering. This condition defines peer relation between sequences. For example, Sequence $t_x t_y$ and $t_y t_x$ are not peers, their LTCs being $(\{t_x, t_y\}, 1)$ and $(\{t_y, t_x\}, 1)$. If one of the sets is unordered (second part of the disjunction), the comparison operator checks for set equality not considering the ordering of elements. This condition defines peer relation between two patterns when one of them is not a sequence. For example, Sequence $t_x t_y$ and AND $(t_y)(t_x)$ are peers, their LTCs being $(\{t_x, t_y\}, 1)$ and $(\{t_y, t_x\}, 0)$.

The *peer* operator is thus used to identify pattern to pattern correspondence between two nets, which contributes to the formulation of hand-in-hand folding and unfolding of the nets. For formulation of Yo-Yo compatible derivation trees of a given pair of nets, identification of peer pattern pairs is the very first requirement. The Yo-Yo algorithm carries out token transportation by transferring colors between the peer patterns. Examples of peer patterns can be seen in Fig. 4, where any two derivation trees satisfy the peer relation.

4.3 Colored Derivation Trees

Coloring of a derivation tree represents a *marking* of the corresponding net. Non-terminals can be colored following Definitions 1 and 2. Fig. 4 shows examples of colorings of derivation trees and the corresponding net markings.

Definition 1 Black Non-terminal: (i) A leaf non-terminal corresponds to a marked place in the net, and (ii) a non-leaf non-terminal abstracts a marked subnet in which no labeled-transition has been fired yet.

Definition 2 Red Non-terminal: It is a non-leaf non-terminal that abstracts a marked subnet where at least one labeled-transition is fired.

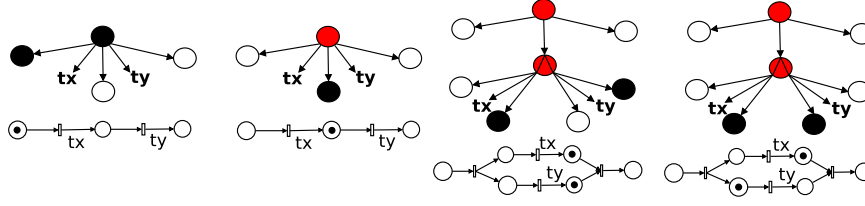


Fig. 4. Examples of Derivation Tree Coloring

5 The Yo-Yo Algorithm for Token Transportation

This section first develops an intuition to the Yo-Yo algorithm, and then discusses the ingredients of the algorithm, which are the consistency and validity notions, token transportation catalog, pattern hierarchy and folding order. The algorithm is discussed at the end of this section.

As discussed previously, a net is considered as a composition of primitive patterns, and the derivation tree of the net is a hierarchy of derivation trees of the component patterns. Due to the hierarchical structuring of patterns, the patterns in the upper level have places that are abstractions of patterns in the lower level. In other words, a pattern is *folded into a place* of a pattern which is at a higher level in the hierarchy. The Yo-Yo algorithm transports tokens from old net to new net by transporting tokens between peer patterns of two derivation trees starting from the top level. The tokens move into their places in the new net as they trickle down when the *folded places unfold*. Resemblance between the stretching and squeezing of the string of the Yo-Yo toy, and the nets being folded and unfolded caused the nomenclature of the algorithm. A *token transportation catalog* is constructed for the purpose of peer to peer token transportation. Transportation in a larger net is thus carried out by applying the cataloged solutions repetitively through the process of folding and unfolding of the patterns organized in the hierarchy. Given two pattern hierarchies the *Folding Order* is formulated, using which both of the nets are folded hand-in-hand.

5.1 Yo-Yo Compatibility at Schema Level

Yo-Yo compatibility at schema level is a structural property, which is necessary for instance migration by the Yo-Yo algorithm. It ensures that the old and the new nets can be folded and unfolded hand-in-hand. During a workflow life-cycle, at the time of building the new net from the old net, if the changes are confined to only the allowed pattern alterations, the schema compatibility can be achieved.

A pattern in a derivation tree can be from any of *AND*, *XOR* and *SEQ* blocks. A pattern occurring at any level in the derivation tree can be replaced by another pattern without changing the tasks involved in the pattern. Consequently, the tree of the old net is modified by replacing a derivation tree pattern by another. Two such replacements can be observed in the tree pair shown in Fig. 7. Syntactically, when a Sequence is changed into an *AND* and *XOR*, triplets are formed by including additional nodes according to the grammar. The reverse

happens in the case of a change from *AND* or *XOR* to Sequence. These syntactic alterations among the primitive patterns can be observed in Figs. 3b,d,f.

The operator $compatible(n_1, n_2)$ defines the Yo-Yo schema compatibility between two nets whose derivation trees are rooted at nodes n_1 and n_2 respectively: $compatible(n_1, n_2) = (yield(n_1) =_y yield(n_2))$, where the yield equality operator $=_y$ compares two yields considering that swaps of triplets in a fork-join pattern are permissible. An example pair of compatible yields is $t_1\{t_2\{t_3, t_4\}, \{t_5, t_6\}t_7\}t_8$ and $t_1t_2t_3t_4\{t_5, t_6\}t_7t_8$ for the trees shown in Fig 8a,b.

5.2 Correctness of Token Transportation

In order to ensure the correctness of the applied dynamic change, validity and consistency of the resultant marking in the new net must be ensured. For Yo-Yo migratability, the following models of consistency and validity are adopted.

Axiom 1 Consistency: *The tasks which are already completed in the old net are also completed in the new net, and vice-versa.*

Axiom 2 Validity: *Resulting marking in the new net is reachable from its initial marking.*

5.3 Pattern Hierarchy and Folding Order

The process of abstracting a single primitive pattern into a place in a net is called *folding*. The reverse, i.e. expansion of a folded place into a pattern is referred to as *unfolding*. Folding operation simplifies the structure of a net consisting of multiple patterns converging into a single pattern at the top level. Folding operation can be applied multiple times, each one simplifying the net further until the whole net is folded into a single pattern at the top. The original net can be obtained by the reverse process of unfolding abstract places into patterns.

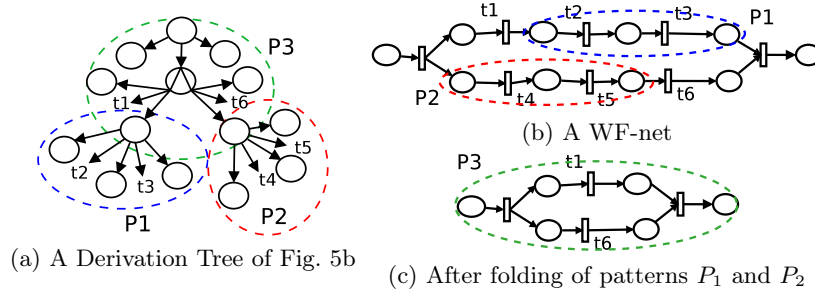


Fig. 5. Derivation Tree and Folding Operation

Pattern hierarchy of a derivation tree is a partial order capturing the nesting hierarchy of derivation tree patterns. In the corresponding net, it gives the

hierarchy of folding of primitive patterns into places. For example, the derivation tree shown in Fig. 5a for the net in Fig. 5b shows the patterns in dotted ellipses for which the pattern hierarchy is given by bottom-up partial order $(\{P_3 \leftarrow P_2, P_3 \leftarrow P_1\})$. Each folding expression $P \leftarrow C$ in the tree gives a pair of parent-child patterns, where child C is folded into a place in parent P . Expressions in round brackets represent sequences, and sets represent no ordering constraint among its folding expression elements.

Given two pattern hierarchies, a *folding order* is formulated for applying the Yo-Yo algorithm which is the order to *fold the peer patterns hand-in-hand*. All parent-child relations among the patterns for each of the two trees are covered (directly or transitively) in the folding order. The top-down folding order expression for the two bottom-up pattern hierarchy expressions $(\{P_3 \leftarrow P_2, P_3 \leftarrow P_1\})$ and $(P'_1 \leftarrow P'_2, P'_3 \leftarrow P'_1)$ is a sequence $(\langle P_3-P'_3 \rangle, \langle P_1-P'_1 \rangle, \langle P_2-P'_2 \rangle)$ consisting of folding expressions $\langle P_i-P'_i \rangle$, $i \in \{1, 2, 3\}$, where P_i, P'_i are peer patterns. An expression in angular brackets is pair of peer patterns.

5.4 Enumeration of the Token Transportation Catalog

The catalog handles token transportation between two *different* patterns. Transfer between the same patterns are handled by the algorithm through a simpler generic step. Consistent migrations between valid markings of different peer patterns create the cases of the token transportation catalog given in Fig. 6. The counts of valid markings for the three patterns *SEQ*, *AND* and *XOR* blocks are 3, 6 and 6 respectively. Each marking further generates variants based on (1) whether the influential places are folded and (2) if a marked place is folded, whether a token in it represents none, partial or full completion of the subnet abstracted in it. Some of the resultant markings that are not migratable due to the consistency criterion are omitted from the catalog. The catalog shown in Fig. 6 contains 37 entries all in all, and the transportation mappings among them. The entries are enlisted as colored derivation trees. A bidirectional arrow between migratable colorings of different patterns means that if one is the old pattern coloring the other can be the new coloring. For example, consider the mapping between case 28 and 26. Case 28 is a sequence, where the token is after t_x and before t_y . Case 26 is an AND pattern which has consistent mapping from case 28. It can be seen that there are two tokens in the two parallel branches of case 26, one after t_x and another before t_y . Thus, both the cases have completed task t_x and hence they are defined to be consistent with each other.

In some cases, a *SEQ* coloring can be mapped to more than one *AND* or *XOR* colorings. These ties are broken based on the conditions on non-empty yields noted in Table 2. A tag symbol is associated with each condition for use in Fig. 6. Also, if a node in the catalog is identified as a *leaf* or *non-leaf* or as a *just completed* folded place (i.e. holding a token just before the exit), the constraint has to be matched. In the table, node o is in the old *SEQ* pattern tree, and n_1, n_2 are in the new fork-join pattern tree. When o is the leftmost child, n_1 is the leftmost child, and n_2 is the node to the left of t_x . When o is the middle child, n_1 is the node to the right of t_x , and n_2 is the node to the left of

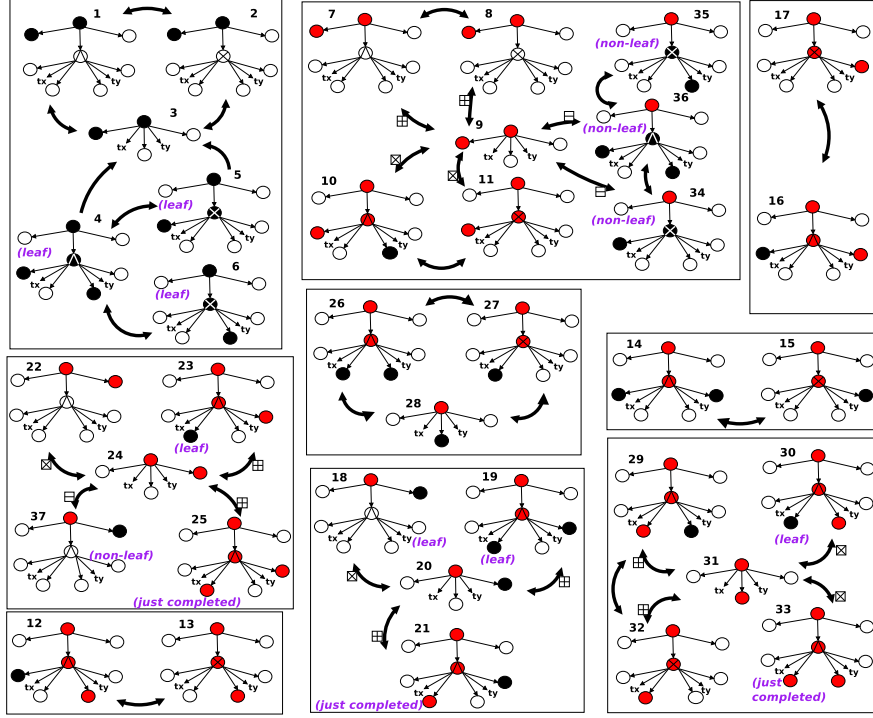


Fig. 6. Consistent and Valid Token Transportation Catalog

t_y . When o is the rightmost child, n_1 is the node to the right of t_y , and n_2 is the rightmost child. Node n is in the subtree rooted at o s.t. $yield(n) =_y yield(n_2)$. It can be noted that, since swapping of two tasks in a sequence is not included in the present catalog, the sequence t_x, t_y never becomes sequence t_y, t_x .

Table 2. Tags and Conditions for Catalog Cases

Tag	Condition to be evaluated
⊞	$yield(o) =_y yield(n_1)$ or $yield(o) =_y yield(n_1).yield(n_2) \wedge \text{uncolored } n$
⊠	$yield(o) =_y yield(n_2)$, or, $yield(o) =_y yield(n_1).yield(n_2) \wedge \text{Red } n$
⊡	$yield(o) =_y yield(n_1).yield(n_2) \wedge \text{Black } n$

5.5 Token Transportation Algorithm

The Yo-Yo algorithm formulates a consistent marking in the new net given the marked old net and a Yo-Yo compatible derivation tree pair of the two nets. The folding order for the derivation tree pair and the token transportation catalog are required for the computation. The algorithm is given in Algorithm 1.

At first, the marking of the old net is translated into a coloring in the derivation tree of the net. Then, the algorithm colors the new derivation tree pattern

by pattern in top-down fashion given by the *folding order*. When all the color ripples reach the leaves, the algorithm successfully terminates.

Algorithm 1: Yo-Yo Algorithm

Input: Old Marked Net N , Unmarked New Net N' , Uncolored Old Derivation Tree D , Uncolored New Derivation Tree D' , Folding Order F , Token Transportation Catalog
Result: Marking in N'

```

1 colorTree( $D, N$ )
2 Let  $\langle p-q \rangle$  be the first folding expression 'fetched' from  $F$ , where  $p$  and  $q$  are
  peer patterns
3 if modularTransport( $p, q$ )  $\neq$  true then return false
4 for every folding expression  $\langle p-q \rangle$  'fetched' from the remainder of  $F$ , not
  violating the partial order specified in  $F$ , where  $q$  has colored root do
5   if  $p$  is colored then
6     if modularTransport( $p, q$ )  $\neq$  true then return false
7   else localPropagation( $q$ )
8 Mark the places in  $N'$  corresponding to Black leaves in  $D'$ 
9 return true

```

Procedure colorTree(Uncolored Derivation Tree D , Marked Net N)

Result: Coloring in D

```

1 for each leaf non-terminal  $n$  in  $D$  corresponding to a marked place in  $N$  do
2   color  $n$  Black
3  $S \leftarrow$  set of colored nodes in  $D$  having uncolored parent
4 while  $S$  is not  $\phi$  do
5    $n \leftarrow$  any element from  $S$ 
6    $p \leftarrow$  colorParent( $n$ )
7    $S \leftarrow S \setminus \{n\}$ 
8   if  $p$  is not NULL then  $S \leftarrow S \cup \{p\}$ 

```

Fig. 7 shows the color propagation traces in the old and then in the new trees. Old Tree: Steps 1-4 depict the bottom-up coloring of the old tree performed by procedure **colorTree**. It starts by coloring the leaf nodes black corresponding to the marked places in the net of Fig. 8a. Then for each colored node, its parent is colored either red or black by procedure **colorTree** until the root is colored.

After transferring color between the top peers, the algorithm goes through the peer patterns $\langle p_i, q_i \rangle$ from the folding order confronting the following cases: (i) root of q_i is uncolored, (ii) p_i is colored, root of q_i is colored, and (iii) p_i is uncolored, root of q_i is colored. In case (i) there is no color transfer. In case (ii), procedure **modularTransport** colors q_i . When p_i and q_i are the same patterns, after replicating the color of p_i to q_i , a red color transferred to a leaf is turned

black to preserve the validity of coloring. If p_i and q_i are different, cataloged transportations are applied. In case (iii), proc. `localPropagation` colors q_i .

Procedure `localPropagation(Uncolored Pattern q)`

Result: Coloring in q

```

1  $r \leftarrow$  root node of  $q$ 
2 if  $r$  is Red then
3    $n_r \leftarrow$  rightmost child of  $r$ 
4   if  $n_r$  is leaf then color  $n_r$  Black else color  $n_r$  Red
5 else color the leftmost child of  $r$  Black

```

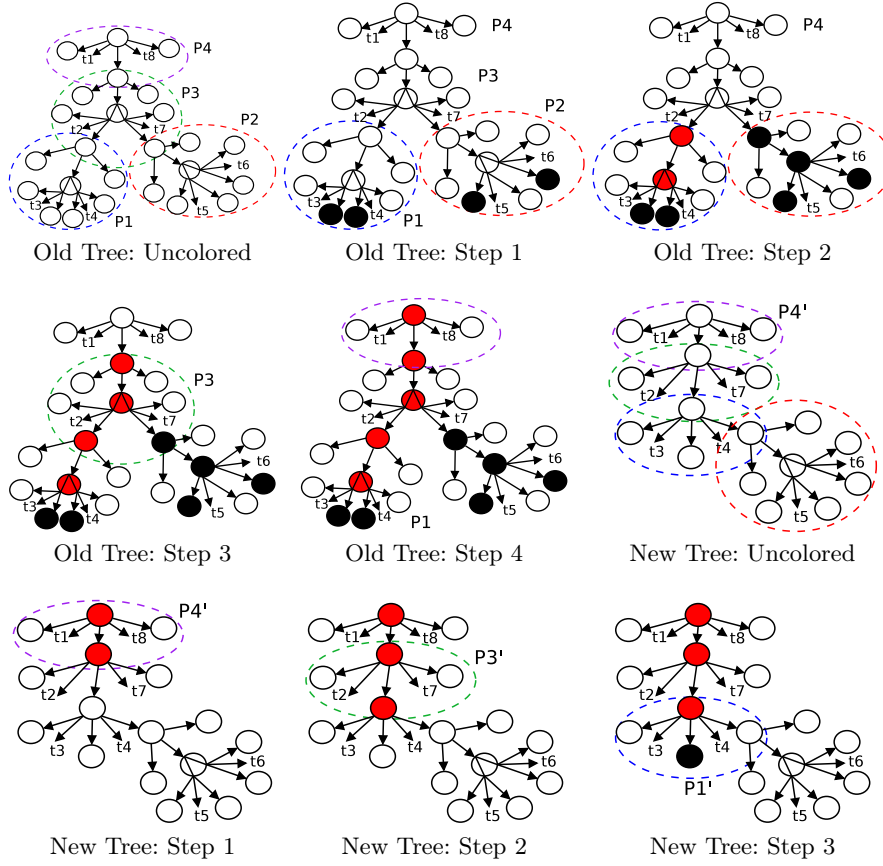


Fig. 7. Derivation Trees Traces in Yo-Yo Transportation

Function modularTransport(Colored Pattern p , Uncolored Pattern q)**Data:** Token Transportation Catalog**Result:** Coloring in q

```

1 if  $p$  and  $q$  are same patterns then
2   | color  $q$  as  $p$  // same color transfer
3   | change the Red leaves of  $q$  to Black // no leaf is left Red
4   | return true
5 search catalog for colored  $p$  and its mapping to  $q$ 
6 if no mapping found then
7   | print instance not consistently migratable, return false
8 color  $q$  as per the search result, return true

```

Function colorParent(non-terminal n)

```

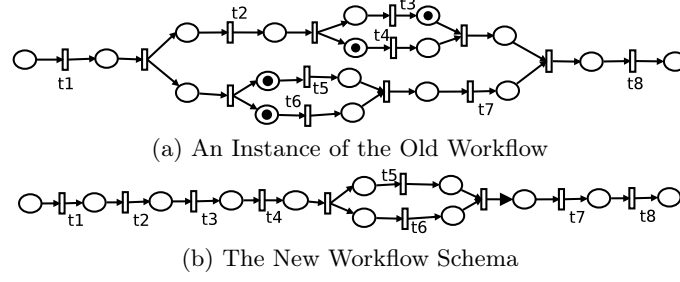
1 if  $n$  is root node in  $D$  then return NULL
2  $p \leftarrow$  parent node of  $n$ 
3 if  $n$  is Red then color  $p$  Red, return  $p$ 
4 if  $n$  is of type SEQ then
5   if  $p$  is of type SEQ then
6     if  $n$  is leftmost child of  $p$  then color  $p$  Black else color  $p$  Red
7   else if  $p$  is of type AND then
8     if  $n$  is left child in any triplet from  $p$  and left child in the
9     other triplet is Black then color  $p$  Black else color  $p$  Red
10  else
11    if  $n$  is left child in any triplet from  $p$  then color  $p$  Black
12    else color  $p$  Red
13 else
14   if non-terminal left to  $n$  is leaf then color  $p$  Black else color  $p$  Red
15 return  $p$ 

```

5.6 An Example Application Scenario

A realistic scenario of dynamic evolution in the reimbursement process in an academic institute is now illustrated where the Yo-Yo algorithm is used for token transportation. The old process schema is modeled by the net depicted in Fig. 8(a). The actual tasks corresponding to each labeled-transition are given in Table 3. As per this design, a student has to first fill the reimbursement form and submit it to initiate a reimbursement request. Next, two concurrent subprocesses begin, one of which is submission of the bills and then approval by guide and head of the department. In parallel, the verification of the funding history for the applicant and funding availability is performed by the awards' committee, after a favorable result is confirmed by the committee approval. The reimbursement amount granted by these three approvals are lastly credited to the student's scholarship account thereby completing the workflow.

This design is evolved into the new schema, depicted in Fig. 8(b) due to the following reasons: every time an application is approved by the head of the

**Fig. 8.** Reimbursement Workflow**Table 3.** Tasks in the reimbursement process in an academic institute

Task Label	Actual Task	Task Label	Actual Task
t_1	fill form & submit	t_5	Funding history verification
t_2	submit documents	t_6	Funding availability verification
t_3	Guide's approval	t_7	Awards' committee's approval
t_4	HOD's approval	t_8	credit transaction

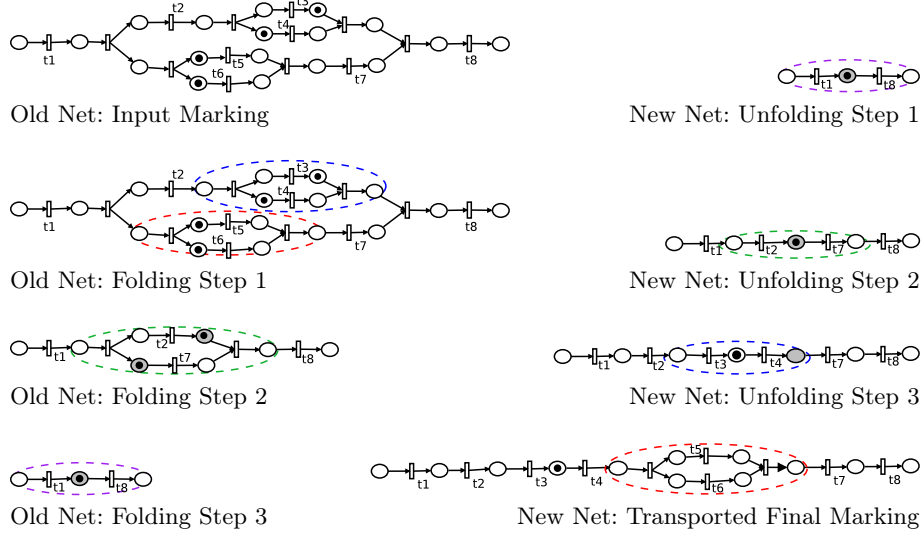
department only after its approval by the student's guide. In the new design, this dependency is reflected explicitly to prevent an applicant from making approval request to the HOD prior to his/her guide. Also, for some cases, though the funding background is verified by the awards' committee, reimbursement is not granted due to rejection either from respective guide or the head. Therefore, to alleviate the unusable funding verification by the awards' committee, the designed concurrency is now made sequential by moving the funding related activities in the later part of the process.

Dynamic migration of the reimbursement applications already in progress relieves the applicants from having to start fresh. Also, the process is too simple to maintain different versions. Therefore, consistent dynamic instance migration in response to the evolutionary changes are desired. The Yo-Yo algorithm carries out the consistent token transportation as shown in Fig. 8.

The visualization of the transportation in the given net pair is depicted in Fig. 9. The bottom-up coloring of the old derivation tree is equivalent to successive folding operations of the marked old net. Again, pattern by pattern top-down coloring of the new derivation tree is equivalent to unfolding a folded pattern in the new net and marking it each time. Movement of the color ripple into a leaf node is equivalent to reaching of a token into an actual place. In this case, the algorithm terminates when all the transported tokens are placed. Fig. 9 shows the nets being squeezed and released as they undergo token transportation.

6 Correctness of the Algorithm

This section provides a sketch of the proof of correctness and comments on the runtime complexity of the Yo-Yo algorithm. A top-down proof is given based on

**Fig. 9.** Token Transportation Through Yo-Yo Steps

a precondition, which is first outlined below.

Completeness of the Catalog A derivation tree of a arbitrary-sized net is composed of derivation trees of the primitive patterns. The folding operation enables us to abstract a bigger net into a single primitive pattern configuration. Given this folding, derivation trees of primitive patterns which are located at a lower level of a bigger derivation tree are abstracted as folded leaf non-terminals in the derivation tree of a pattern located at a higher level. Therefore, a derivation tree of a primitive pattern can have one or more of the following two types of leaf non-terminals based on where the pattern is located in the entire tree: (1) leaves which represent folded lower level patterns. These are tagged as *non-leaf* in the catalog, and (2) leaves which are actual places in the entire net. These are unfolded leaves which are tagged as *leaf* in the catalog.

Coloring of derivation trees is an encoding scheme under which all places fall into either of the three color based classes shown in the Table 4.

Table 4. Coloring Scheme for Catalog Patterns

Type of node	Marking Status	Execution Status	Color
Folded (non-leaf)	Unmarked	Not applicable	Uncolored
Unfolded (leaf)	Unmarked	Not applicable	Uncolored
Folded (non-leaf)	Marked	null-executed (just started)	Black
Unfolded (leaf)	Marked	Not applicable	Black
Folded (non-leaf)	marked	full-executed (just completed)	Red
Folded (non-leaf)	marked	partially-executed	Red

Every place in a pattern can belong to one of the three classes provided that the resultant marking is a valid marking. Every marked place in a valid pattern marking can be colored black or red. This leads to 6 possible colorings of SEQ pattern given that there are 3 valid markings of the primitive SEQ pattern. There are 6 valid markings of the primitive AND pattern which leads to 20 colorings. Similarly, for 6 valid marking of primitive XOR, there are 12 colorings. Out of these 38 cases, three cases of AND, and two cases of XOR are not migratable to any other consistent and valid making in any other pattern as per the correctness criteria. This gives a total of 33 unique migratable colorings of derivation trees of all primitive patterns. However, 4 more cases need to be considered as follows.

It can be seen that the six rows of the table have been colored using three colors. Using six different colors results in a much bigger catalog. It was found that clubbing the cases reduces the size of the catalog considerably, leaving out 4 extra cases that need to be handled separately. The clubbing is done based on whether the nodes are marked, and if marked, whether at least one task in the folded section is done. In the catalog these 4 extra cases are due to pairs 18 and 37, 4 and 36, 5 and 34, 6 and 35. One case in each pair is covered in the above 33 cases. In this way we obtain 37 valid and exhaustive entries in the catalog.

Mappings among this group are given as per the consistency criteria. For some cases among these 37 cases, there are multiple mappings possible. These are resolved by yield-based tie-breaker rules as explained previously.

The Correctness Argument First, the algorithm colors the old derivation tree as per the old net marking, preserving the semantics of derivation tree coloring as given in Section 4.3. Next, it transports the color from the old top pattern to the new top pattern. If this step is not possible without violating consistency, the algorithm terminates. After the transfer, the colors are propagated further down through the descendant patterns in the new tree following the folding order. The color transfer iteratively continues until either no pattern can be colored thus, or till the algorithm terminates on finding the case not migratable. If a case is migratable, Lemma 1 proves that given the yield compatibility at the roots of a peer patterns guaranties that consistent color transfer between them leads to consistent color transfer in the immediate child patterns. To prove that this can be done repetitively for the entire tree Lemma 2 is used. Lemma 3 proves validity of each color transfer. As a result, the algorithm is guaranteed to terminate and produces correct token transportation.

Lemma 1 *For a given pattern P' in the new tree having yield compatible root with peer pattern P in the old tree, consistent color transfer to P' guaranties to find consistent coloring of the immediate child patterns of P' .*

Proof: Coloring P' either by `modularTransport` or `localPropagation` leads to coloring of the roots of the child patterns visible to P' . Given the yield compatibility between the roots of P and P' , this color passing either by catalog cases or `localTransport` can result in the following variants coloring of a child root against the root of its peer in the old tree. Let Q' be a direct child pattern of

P' . Let Q' have peer Q in the old tree. (i) Roots of Q and Q' have the same color (ii) Both have no color, and (iii) One of them is colored and the other is uncolored. (Note that, if P' and P are the same patterns, color transfer to P' follows either case (i) or case (ii)).

In case (i), either one of the catalog mappings or the same color replication mapping between Q and Q' is guaranteed to be applicable. Since both of them are preserve consistency by construction, the lemma applies for this case. In case (ii), a pattern remains uncolored if either token has moved past it or has not reached in it yet. In case (iii), the colored root can be either black or red, which gives us four possibilities. If Q is uncolored and the root of Q' is black, the token has not reached in Q , whereas in Q' black color means that is in the source place indicating that no labeled-transition is fired yet. If the root of Q' is red, the token is past Q , whereas in Q' it is in the sink place just after firing the last transition in it. The other two possibilities in this case are reverse of the first two possibilities. Given the yield compatibility between the parent roots, i.e. the roots of P and P' , it ensures the same relative positioning of Q and Q' with their respective parent patterns, i.e. they are both either left, right or middle children. Therefore, when P and P' are consistently colored, a token before Q and after Q' is a contradiction, which proves case (ii). Similar argument follows for case (iii) also. In this way consistent transportation for the direct children can be achieved. Case (i) is handled by `modularTransport`, case (ii) and the last two possibilities of case (iii) do not require any coloring action, and the first two possibilities of case (iii) are handled by `localPropagation`.

Lemma 2 *Let two Yo-Yo compatible derivation trees have patterns P, Q in the old tree and their respective peer patterns P', Q' in the new tree. Let Q be child of P and Q' is child of P' . Let the roots of P and P' satisfy yield compatibility. If P' and Q' satisfy Lemma 1, then so do Q' and all of its immediate children.*

Proof: The lemma is about a structural property achieved due to Yo-Yo compatibility and folding order that ensures yield compatibility between the roots of peer patterns extracted from the folding order for coloring at each step of iteration. We use notation S_X to represent the yield sequence of the root of derivation sub-tree X . Let the transition terminals of peer patterns P and P' be denoted by t_x (left) and t_y (right). P and P' can be either of Sequence or fork-join patterns. We analysis the case where P is a Sequence and P' is a fork-join. The other three cases can be proved similarly. When P is a Sequence and P' is a fork-join, P' has at most six immediate child patterns rooted at its six leaf non-terminals. These are shown in Fig. 10a as $Q'_{11}, Q'_{12}, Q'_{21}, Q'_{22}, Q'_{31}$, and Q'_{32} . The term Q' in the lemma refers to each one of them. Similarly, P being a Sequence has three child patterns Q_1, Q_2 and Q_3 as shown in Fig. 10b. Since the roots of P and P' are yield compatible, $S_{Q_1}t_xS_{Q_2}t_yS_{Q_3} =_y \{S_{Q'_{11}}S_{Q'_{12}}t_xS_{Q'_{21}}, S_{Q'_{22}}t_yS_{Q'_{31}}S_{Q'_{32}}\}$. From this, at subtree level we can observe that $S_{Q_i} =_y S_{Q'_{i1}}S_{Q'_{i2}}, i \in \{1, 2, 3\}$. As each pattern has two transitions, to accommodate four transitions, each of Q_1, Q_2 and Q_3 is a hierarchy of two patterns as shown in Fig. 10c or 10d.

Now there are two cases: Q' can be either Q'_{i1} or Q'_{i2} . For the first case, P has subtree as Fig. 10c, and for the second P has subtree as Fig. 10d. In the folding

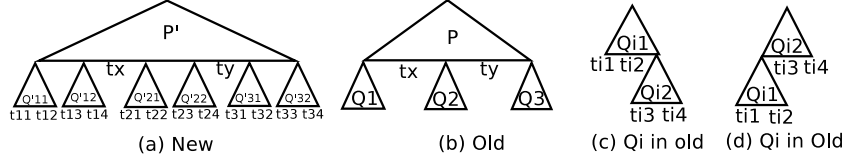


Fig. 10. Visualization of the subtrees of P' and P in Lemma 2

order corresponding to the first case, element $\langle Q_{i1}-Q'_{i1} \rangle$ precedes element $\langle Q_{i2}-Q'_{i2} \rangle$, and for the second case they are swapped. As the coloring follows the folding order, after every step the visible parts of the nets are composed of the patterns from the already traversed. In this regard, for the extracted element $\langle Q_{ij}-Q'_{ij} \rangle$, the roots of Q_{ij} and Q'_{ij} are yield compatible and hence, consistent color transfer to Q'_{ij} guaranties consistent color transfer to the next level, thereby proving Lemma 1 for this case. The justification for other combinations of P and P' mentioned at the beginning of this argument is skipped due to lack of space.

Lemma 3 *Peer to peer color transfer preserves validity pattern of coloring.*

Proof: For catalog transfer cases validity is preserved by construction. For same color replication, validity of the newly produced color is preserved by the correctness of the old pattern coloring and the validity preserving step of the algorithm (line 3 of `modularTransport`). For local transportation, validity is preserved by following the definitions. Therefore, the lemma is proved.

Time Complexity and Brief Comparison As it can be observed from the algorithm, the asymptotic time complexity of the runtime token transportation in terms of number of patterns n is linear. This computation does not include parsing of the workflow specifications, finding out the compatible derivation tree pairs and the folding order. Therefore, the Yo-Yo approach improves the runtime migration cost by pushing much of the complexity into one-time schema level computations and design time catalog construction depending on the grammar. As compared to history-replay approach, Yo-Yo transportation does not compute or reproduce history. Transportation via pre-computed mappings among *marked patterns* achieves the desired migration.

7 Conclusions and Future Work

The paper developed a novel catalog based dynamic token transportation technique called Yo-Yo algorithm with the help of contributory concepts such as CWS specification grammar for block structured WF-nets, derivation trees and their colorings, peer patterns, Yo-Yo compatibility, catalog based transportation, and folding and unfolding of hierarchically organized patterns. The algorithm uses the ready-made consistent and valid migration solutions from the token transportation catalog repetitively to achieve correct transportation for non-primitive bigger nets. Also, immediately non-migratable markings are automatically identified by the algorithm due to the non-existence of the corresponding entries in

the catalog. We supplemented the discussion on the algorithm with an example realistic situation and also provided the sketch of the proof of correctness. We plan to integrate an implementation of this algorithm in the workflow engine described in [17]. We aim to generalize the approach to handle replacement, removal, addition and swapping of tasks.

References

1. van der Aalst, W.M.: The application of petri nets to workflow management. *Journal of circuits, systems, and computers* **8**(01) (1998)
2. Pradhan, A., Joshi, R.K.: Token transportation in petri net models of workflow patterns. In: 7th India Software Engineering Conference, India, 2014. (2014)
3. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: *Proceedings of conference on Organizational computing systems*, ACM (1995)
4. Ellis, C.A., Keddara, K.: A workflow change is a workflow. In: *Business Process Management, Models, Techniques, and Empirical Studies*, Springer-Verlag (2000)
5. van der Aalst, W.M.: Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers* **3**(3) (2001)
6. Sun, P., Jiang, C.: Analysis of workflow dynamic changes based on petri net. *Information and Software Technology* **51**(2) (2009)
7. Cicirelli, F., Furfaro, A., Nigro, L.: A service-based architecture for dynamically reconfigurable workflows. *Journal of Systems and Software* **83**(7) (2010)
8. Agostini, A., Michelis, G.D.: Improving flexibility of workflow management systems. In: *Business Process Management: Models, Techniques, and Empirical Studies*. LNCS 1806, Springer (2000)
9. van der Aalst, W.M., Basten, T.: Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science* **270**(1) (2002)
10. Murata, T.: Petri nets: Properties, analysis and applications. *Proceedings of the IEEE* **77**(4) (1989) 541–580
11. Morita, K., Watanabe, T.: The legal firing sequence problem of petri nets with state machine structure. In: *Circuits and Systems, 1996. ISCAS'96., Connecting the World., 1996 IEEE International Symposium on*. Volume 3., IEEE (1996) 64–67
12. Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., Bloch, B., Curbera, F., Ford, M., Goland, Y., et al.: Web services business process execution language version 2.0. *OASIS standard* **11** (2007)
13. Gschwind, T., Koehler, J., Wong, J.: Applying patterns during business process modeling. In: *Business process management*. Springer (2008)
14. Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - enhancing flexibility in process-aware information systems. *Data Knowl. Eng.* **66**(3) (2008)
15. Van Der Aalst, W.M.P., Ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. *Distrib. Parallel Databases* **14**(1) (2003)
16. Vanhatalo, J., Völzer, H., Koehler, J.: The refined process structure tree. *Data & Knowledge Engineering* **68**(9) (2009) 793–818
17. Pradhan, A., Joshi, R.K.: Architecture of a light-weight non-threaded event oriented workflow engine. In: *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, India, 2014.* (2014) 342–345

Part IV

Poster Abstracts

De-Materializing Local Public Administration Processes

Giancarlo Ballauco¹, Paolo Ceravolo², Ernesto Damiani³, Fulvio Frati², and
Francesco Zavatarelli²

¹ I-Conn, Trento, Italy

`giancarlo.ballauco@nitidaimmagine.it`

² Computer Science Department, Università degli Studi di Milano, Italy

`{paolo.ceravolo, fulvio.frati, francesco.zavatarelli}@unimi.it`

³ Etisalat British Telecom Innovation Center/Khalifa University, Abu Dhabi, UAE

`ernesto.damiani@kustar.ac.ae`

Abstract. We describe a framework for the de-materialization of local public administration processes that provides remote assistance by human operators when needed. Our framework is in an advanced state of development and will be tested in several municipalities of the province of Trento (Italy).

Like many European countries, Italy is merging little municipalities into districts in order to streamline local services, re-organize staff, and reduce indirect costs [4]. As local services get relocated to municipalities chosen as district leaders, citizens face longer trips and increased inconvenience in accessing services. This situation has triggered research [1, 2] on technologies able to de-materialize Local Public Administration (LPA) processes, providing remote access to them via smartphones or special-purpose access points located in schools, shopping malls, and shops. Experience has shown that citizens used to face-to-face interactions find hard to access LPA processes via Web sites. Whenever processes involve choices that may generate a penalty (for instance, paying a local tax) users require the assistance of a human [3, 4].

Our solution relies on a platform that executes all LPA process steps remotely, calling in a human operator when necessary. The level of assistance is context-dependent, i.e. takes into account the task at hand, the logistics of the point of access, the age, hearing and eyesight capabilities of the user as well as the current state of the Internet connection and access devices. If the user looks uncertain or confused, the remote assistance gets activated automatically without waiting for the specific request.

In our system, LPA processes are described using BPMN. The standard BPMN palette has been extended with specific elements associate to proprietary scripts capable to detect and activate the right assistance level and send direct commands to I/O devices (printer, scanner, ...). We defined a set of heuristic rules [5] that evaluate the context (for instance `ConnectionQualityLevel`

> 3) and decide the action to deal with it, e.g. `StartAudioCall`. Our rules are written as annotations to the BPMN diagram. Fig. 1 shows an extended BPMN diagram, where icons in the upper left corner of each activity determine the type of rule implemented in the activity.

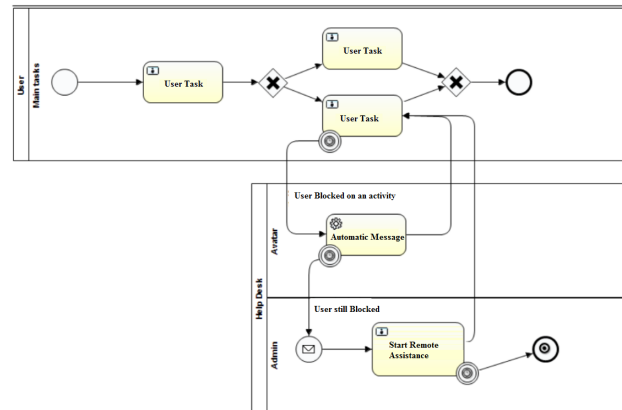


Fig. 1. Example of enriched BPMN diagram.

Conclusions and Future Work

Our solution for de-materializing LPA processes preserves human assistance to users. Our BPMN extension expresses the interaction types that can be activated for each activity.

References

1. S. Armenia, D. Canini, and N. Casalino. A System Dynamics Approach to the Paper Dematerialization Process in the Italian Public Administration. *Interdisciplinary Aspects of Information Systems Studies*, Springer, 2008.
2. N. Mirabella, L. Rigamonti, and S. Scalbi. Life cycle assessment of Information and Communication Technology application: a case study of dematerialization in the Italian Public Administration. *Journal of Cleaner Production*, vol. 44, pp. 115-122, 2013.
3. J. Becker, B. Niehaves, P. Bergener, and M. Raeckers. Digital Divide in eGovernment: The eInclusion Gap Model. *Electronic Government Lecture Notes in Computer Science*, vol. 5184, pp. 231-242, 2008.
4. M.J. Goldsmith, E.C. Page, eds. *Changing Government Relations in Europe. From Localism to Intergovernmentalism*. Oxon, Routledge, 2010.
5. Y. Sakurai, K. Takada, M. Anisetti, V. Bellandi, P. Ceravolo, E. Damiani, and S. Tsuruta. Toward sensor-based context aware systems. *Sensors* 12:1, pp. 632-649, 2012.

Renew – The Reference Net Workshop

Lawrence Cabac, Michael Haustermann and David Mosteller

University of Hamburg, Department of Informatics
<http://www.informatik.uni-hamburg.de/TGI/>

RENEW is a continuously developed extensible Petri net tool, which enables modeling and simulating of various Petri net formalisms. One unique characteristic of the tool is the full support for Java reference nets [2], which combine the concepts of nets-within-nets and synchronous channels with a reference semantics using a pattern/instance mechanism analogously to object oriented programming languages. Furthermore Java can be used as inscription language whereby the formalism is well-suited for the implementation of concurrent software systems. Since RENEW is written in Java it is available for multiple platforms (including Windows, Linux and Mac). The current version 2.4.2 is available for download¹ free of charge including the source code [3].

RENEW provides an easy to use graphical editor for Petri net models and other types of models and a simulation engine, which is seamlessly integrated into this editor. It has a plug-in architecture, which makes it easily extensible. The core plug-ins are provided as part of the RENEW distribution. Many advanced features are supplied by optional plug-ins.

The editor has been improved over the last years and received many small usability enhancements and has evolved into an integrated development environment (IDE) for net based software development. It contains a syntax check during editing and debugging tools, such as breakpoints or manual transitions. Furthermore the editor features desktop integration, a file navigator and image export to various formats.

The simulator is capable of handling different formalisms. The main formalism is the Java reference net formalism, for which different extensions exist, such as inhibitor, reset and timed arcs. The workflow net formalism, provided by an optional plug-in, adds a task transition, which can be canceled during execution, so that its effect on the net can be reverted. Other formalisms provide simulation of P/T nets, feature structure nets and bool nets. Simulation is available in different modes. In the interactive simulation mode the user may control the simulation by choosing the transitions to fire and inspect each single step. The automatic simulation mode is usable for system execution and can be run with and without graphical feedback. RENEW features dynamic loading of nets on demand and configurable logging of simulation events. The monitoring plugins facilitate the inspection of remote simulations. With an integration of the LoLA verification tool [1] RENEW is also suited for verification tasks during modeling.

The first official version of RENEW was released in 1999 and has since then been continuously developed as a Petri net editing and simulation environment. The plugin system, introduced with the major release 2.0 in 2004 [4], enabled

¹ RENEW web page: <http://www.renew.de/>.

the extension of RENEW into various directions. Many of the newly developed plug-ins are related to agent-oriented software engineering. Additionally, RENEW was utilized to provide a workflow management engine and clients. Besides using RENEW primarily for modeling Petri nets, plugins provide support for different modeling techniques, i.e. diagrams from UML or BPMN.

In the future we like to further improve RENEW as an IDE for modeling and implementation with Petri nets. Anyhow, our plans in using RENEW's graphical framework as a modeling environment are not restricted to Petri nets. One of our current research projects aims at advancing RENEW to a framework for meta-modeling domain specific modeling languages [5]. Further research topics are concerned with providing the facilities to enable distributed simulations across multiple instances of RENEW and in distributed networks [6]. To furthermore qualify RENEW as an IDE for model based software engineering in a distributed software development environment, we are currently developing a plugin to integrate project management features. Another research project is concerned with utilizing RENEW as a library or service to other applications. Additional enhancements aim at improving the editor capabilities of RENEW. Drag and drop support for the navigator will support the usability by providing easy to use facilities to managing files. Our release plan includes improving the quick fix feature to provide better proposals for automated code completion. A re-designed console plugin enables interactive command line processing with history and command completion.

References

1. Hewelt, M., Wagner, T., Cabac, L.: Integrating verification into the PAOSE approach. In: Duvigneau, M., Moldt, D., Hiraishi, K. (eds.) Petri Nets and Software Engineering. International Workshop PNSE'11, Newcastle upon Tyne, UK, June 2011. Proceedings. CEUR Workshop Proceedings, vol. 723, pp. 124–135. CEUR-WS.org (Jun 2011)
2. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
3. Kummer, O., Wienberg, F., Duvigneau, M., Cabac, L.: Renew – User Guide (Release 2.4.2). University of Hamburg, Faculty of Informatics, Theoretical Foundations Group, Hamburg (Jan 2015), <http://www.renew.de/>
4. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An extensible editor and simulation engine for Petri nets: Renew. In: Cortadella, J., Reisig, W. (eds.) Applications and Theory of Petri Nets 2004. 25th International Conference, ICATPN 2004, Bologna, Italy, June 2004. Proceedings. Lecture Notes in Computer Science, vol. 3099, pp. 484–493. Springer, Berlin Heidelberg New York (Jun 2004)
5. Mosteller, D., Cabac, L., Haustermann, M.: An Approach to Meta-Modeling with Petri Nets. In: Moldt, D., Rölke, H., Störrle, H. (eds.) Petri Nets and Software Engineering. International Workshop PNSE'15, Brussels, Belgium, June 2015. Proceedings. CEUR Workshop Proceedings, vol. 1372. CEUR-WS.org (Jun 2015)
6. Simon, M.: Concept and Implementation of Distributed Simulations in RENEW. Bachelor thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg (Mar 2014)

Queue-less, Uncentralized Resource Discovery: Formal Specification and Verification

Camille Coti, Sami Evangelista, and Kais Klai

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030
F-93430, Villetaneuse, France
{first.last}@univ-paris13.fr

A New Fully Distributed Resource Management System In this paper, we present a formal approach for the specification and the verification of a fully distributed resource reservation system. Our system is made of two parts: the *launcher*, which is executed by the user who wants to run a job on a set of computing nodes, and the *agent*, which is a daemon running on all the resources that exist in the system.

Clients must have an exclusive access to the resources that are allocated for them. Under the requirement that clients have reasonable requirements, all the clients' requests are answered positively in a finite time and all the jobs are executed completely. In order to ensure the correctness of our system regarding such properties, we describe it using a Petri net model, we express formally the desired properties and we perform their formal verification successfully.

Our algorithm relies on the service discovery tools provided by the Zeroconf protocol. Computing nodes declare themselves on the Zeroconf bus. However, this simple discovery service is not sufficient to ensure that the computing resources will not be used by several jobs at the same time.

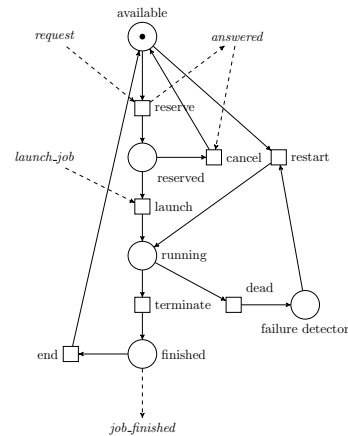


Fig. 1. Model for handling resource volatility with a failure detector

Modelling The Petri net model of a machine is presented on Fig. 1. A machine can be reserved when it is *available*. It answers the client and switches into *reserved* mode. When the local process is done, the machine switches to state *finished*, signals to the client that its part of the job is done, and then returns back to state *available*. There is only one *available* place on each resource, and this place contains only one token in the initial marking. Hence, a machine can answer positively to one client only.

A model where 2 clients issue concurrent resource allocation requests on the same set of resources is represented on Figure 2. Each client has its own reservation system. We represented 2 clients, one requesting n resources and the other m resources.

The *cancel* transition is very important here to release some resources in case of a deadlock caused by a conflict between applications occurring for instance when all the available machines are reserved but no application is able to start. Therefore, after a certain time, if no additional resources appear on the Zeroconf bus, the machines reserved for at least one application will be freed and become available for the other one.

Analysis We analyzed both generic and specific properties. As generic properties, we were interested in deadlock freeness, boundedness and soundness. The deadlock freeness ensures that no dead state (a state from which no transition is fireable), except the final state (all the jobs are done), is reachable. The boundless property ensures that the number of reachable states is finite. This has been ensured by finding out that the state space of the system has been fully and successfully built in a finite time. Finally the soundness property implies three requirements: (1) *option to complete*, (2) *proper completion*, and (3) *no dead transitions*.

The table right below gives the execution time (in seconds) of Helena and statistical data on their state space : the number of reachable states, the number of terminal reachable states, and the number of arcs in the state space. We selected a set of 6 configurations according to their state space size. A first analysis of the state space report revealed that our model is bounded and that all transitions are executable.

Regarding specific properties, we were interested in checking the following: (1) It is never possible for a machine to be running two different applications, and (2) it is always possible to answer possibly any request (as long as the number of required resources is less than the number of the machines available in the system).

As a conclusion, the properties expected are all verified provided a few reasonable assumptions are made on the environment. First, if we assume that an infinite number of cancellations can not infinitely postpone the beginning of a scheduled job then we can ensure that any submitted job will be scheduled and executed if enough machines are available. Second, in the presence of machine failures, a scheduled job can always terminate if we assume that the pool of available machines allows it.

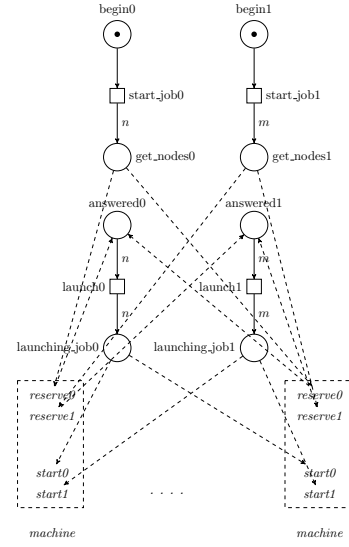


Fig. 2. Reservation system of 2 clients

Configurations				Analysis results			
J	M	P	F	Time	States	Term.	Arcs
4	6	4	no	3.92	1,369,236	1	2,849,412
6	4	2	yes	5.90	2,865,804	1,999	5,740,698
5	6	4	no	13.10	8,407,677	1	17,557,805
4	6	4	yes	20.08	12,111,398	559	27,376,192
6	6	4	no	85.25	43,094,470	1	90,124,518
5	6	4	yes	164.36	65,633,194	1,743	151,096,440

Introducing the *Quick Fix* for the Petri Net Modeling Tool RENEW

Jan Hicken, Lawrence Cabac, Michael Haustermann

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics

`{1hicken,cabac,haustermann}@informatik.uni-hamburg.de`

1 Extended Abstract

Many modern integrated development environments (IDEs) such as Eclipse [2] support developers by providing a *quick fix* feature. If the application detects syntax errors in the source code, it can propose fixes addressing the error by providing context-sensitive alternatives the developer can choose from interactively.

Considering Petri net modeling tools, similar functionality can be useful for inscriptions for high-level nets. We want to transfer the functionality to RENEW [1], where the Java-based inscriptions may be treated analogously to source code in IDEs. This includes a mechanism for detecting and highlighting errors for developer on the one hand and algorithms for suggesting possible fixes to these errors on the other hand. Furthermore, the feature shall apply these fixes automatically, so that the developer has to adjust as little as possible source code manually.

The first category of suggestions consists of Java fields: The choices consist of the declared and accessible fields for a denoted class in the source code. If an entered field is unknown, the quick fix can provide a list of fields using the Java Reflect API. However, this can lead to a lot of choices being presented to the developer, when the class defines a great amount of fields. In order to address excessive lists of suggestions, lists may be filtered to fields having the same prefix as entered by the developer.

Similar to the fields, methods can also be searched by using the Java Reflect API. Filtering the list of methods declared by a class may also be done by comparing prefixes. In addition to that, the entered parameters need to be considered when providing suggestions to the user. A method giving the wrong parameter types, the feature can suggest possible method overloads.

Variables, that are used in the Petri net can be declared in RENEW's declaration node. If the net contains a declaration node with at least one declared variable, all used variables have to be declared. An undeclared variable causes the quick fix to suggest possible types for that variable and adding the corresponding statement in the declaration node.

When assigning a value to a variable, the value can either be a literal, a newly constructed object or the returned value from a method call. Determining

the type in the first case is trivial, because the corresponding literal's primitive type can be evaluated by the compiler immediately. To determine the type of a newly constructed object, the constructor's class name has to be well-known, which means the class name is either fully qualified or an already imported class. Giving the case of a well-known class name, the corresponding declaration statement can also be generated easily. Moreover, the information of a fully-qualified class name can be used to construct an import statement for that class or its whole package right away. The last case is also rather trivial, because the return type of a method and its corresponding class object can be determined using the Java Reflect API.

With the quick fix feature, we achieved a further integration of modeling support, which helps the developer reaching his goals. Errors in the source code are explicitly highlighted and possible fixes can be applied interactively. In addition to the management of development errors, the feature can be used similar to an autocomplete feature. The developer does not have to look up every class member he wants to use in the API documentation but can choose from any alternatives the quick fix provides. We streamlined the development process by enabling a much faster development in an integrated environment, which is easily extensible.

When suggesting fields or methods for a class, the quick fix can provide all known members. If the amount of class members is high, the filtering only applies to suggestions with the same prefix as the entered text. It is possible to extend this feature to filter for matching types and parameters, which also enables a prioritization of suggestions. The suggestions for variables are reliable when it comes to assignments, where the right part of the expression is parseable and the type is known. However, when it comes to arc inscriptions and variable types, which have not been imported, the current algorithm does not give sufficient results. The former case may require an analyzation of other in- and outgoing arcs regarding the particular place or transition and its inscriptions. Supporting unimported types depends highly on the ability to automatically import classes found in the classpath. In addition to that, the class hierarchies are not part of the suggestion mechanism at all.

Furthermore, the quick fix feature not only is applicable to Java source code within inscriptions but also affects modeling errors. Common Petri net properties are yet verifiable through algorithms. These may be used to address unwanted discrepancies between wanted and actual net properties using the quick fix.

References

1. Kummer, O., Wienberg, F., Duvigneau, M.: Renew – the Reference Net Workshop (Jun 2015), <http://www.renew.de/>, release 2.4.2
2. The Eclipse Foundation: Eclipse: The Eclipse Foundation open source community website (Jun 2015), <http://www.eclipse.org/>

Process-oriented Worksheets for the Support of Teaching Projects

Dennis Schmitz and Lawrence Cabac

University of Hamburg, Department of Informatics
<http://www.informatik.uni-hamburg.de/TGI/>

Keywords: Teaching, Software Engineering, Agents, Petri Nets, Process, Learnflows

Extended Abstract

Teaching students how to develop a single software application in a large team (e.g. with 25 people) is a challenging task. Especially if the software development process is distributed and concurrent, the participants have heterogeneous basic knowledge of software development and are not acquainted with the basics of the used software development approach.

We are teaching the Petri net-based, agent- and organisation-oriented software development approach (PAOSE) [2,3] in a period of one semester in the form of a project. The first six weeks cover the introduction phase. In this work we concentrate on this first phase only. The students are learning the basics of the PAOSE approach and how to apply it to develop multi-agent systems. In this context they get to know the corresponding methods, techniques and tools and have the opportunity to become familiar with these. The rest of the project the tutors and students cooperatively develop a multi-agent application.

From the previous projects, improvements to this approach emerged and the technical complexity of the environment increased. One example is the WebGateway [1], which enables the development and integration of web-based services for multi-agent systems. Due to the increased technical complexity, we experienced that the teaching complexity increased, too. Also, the large amount of new information can be confusing and frustrating for the students. We have observed, that in order to minimize this confusion and frustration it is important, to bring the information into a sensible order. This means, to provide the students with all necessary information in the adequate context. If the information is presented too early, the students could forget it until they need it. If the information is presented too late, the difficulty of the exercise unintentionally increases and this could lead to frustrated students.

In order to improve the structure of the existing worksheets, we analyzed these in regard of the inherent processes, which are necessary to solve the exercises [4]. As modeling technique we used workflow Petri nets [5]. We call these workflows *learnflows*. Based on these learnflow models, we restructured and augmented the worksheets. During this restructuring process it occurs, that we also extended the learnflow models. An interdependency of the refined texts and the

learnflow models emerged. This procedure leads to refined worksheets containing logically ordered and detailed instructions, which the students can follow step by step. The use of the workflow Petri nets helps the tutors, to rearrange the worksheet text in a better order. It shows them, which information appears in the wrong place (e.g. the information is necessary for an exercise on the current page, but appears only on the subsequent page). Also, it shows the tutors which information is missing and helpful for the students during processing the exercises.

In the latest teaching project we have already applied these worksheets and the workflow Petri net models as visual support along with the texts. We have noticed less frustration and confusion among the students. They are now able to process the worksheets faster and more independently. Therefore, the tutors are able to concentrate on teaching the actual objectives. Also the process-oriented worksheets support the communication between the tutors and the students. If a student needs some help, it is easy to identify which steps he has accomplished and to find out where precisely the challenge occurs.

On the poster we will present the old non-process-oriented and the new process-oriented version of an exercise of a worksheet of our teaching project. In the previous exercises the students already implemented a one-to-one message communication between agents. The objective of this exercise is, to extend this functionality to a one-to-all (broadcast) message communication between agents. For illustration we present the text and the corresponding visualized process model of both versions of the exercise. The visualization of the process of the non-process-oriented exercise did not exist before this work. We use it, to enable a visual comparison to make the differences obvious. These mentioned resources and also an extract of this text can be accessed on the PAOSE homepage [3] via <http://www.paose.net/wiki/ProcessOrientedWorksheets>.

The new worksheets represent the first stage of a process-oriented learning environment, which we are designing for teaching projects. The environment will provide a web-based support for the process-oriented creation and processing of worksheets, based on an agent-oriented workflow management system.

References

1. T. Betz et al. Integrating web services in Petri net-based agent applications. In D. Moldt and H. Rölke, editors, *Petri Nets and Software Engineering. International Workshop PNSE'13, Milano, Italia, June 2013. Proceedings*, volume 989 of *CEUR Workshop Proceedings*, pages 97–116, June 2013.
2. L. Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2010.
3. PAOSE homepage. <http://www.paose.net/>. Last accessed on 31st May, 2015.
4. D. Schmitz. Unterstützung von Lehr- und Lernprozessen am Beispiel des PAOSE-Lehrprojekts. Master thesis, August 2015, University of Hamburg, Department of Informatics, 2015.
5. W. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer Berlin Heidelberg, 1997.

Integrating Network Technique into Distributed Agent-Oriented Software Development Projects

Christian Röder and Lawrence Cabac

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, <http://www.informatik.uni-hamburg.de/TGI/>

Keywords: Project Management, Network Technique, Software Engineering, Petri Nets, Modeling

Extended Abstract

The management of local software projects is challenging, due to its complexity. In case of distributed development projects, the complexity in project management increases even more [1]. In this publication we introduce and adapt the well-proven *network technique*¹ into PAOSE, a distributed agent-oriented software development approach, by directly integrating a modeling tool for network technique into the development environment of PAOSE. We support the project participants in modeling their interdependent project activities and reason about them more easily, using an illustrative, graphical syntax. By providing the mapping of network technique diagrams onto Petri nets, we utilize formal semantics and improve integration into our development approach, which is based on Petri nets.

The participants of distributed software development projects have to tackle several challenges: Project members, which are organized in sub-teams, perform activities at spacial and temporal distance from one another. These distances have an impact on communication, coordination and control [1]. If, in order to counter these challenges, concepts from agile project management that value self-responsibility are applied, the sub-teams are required to self-organize and self-manage their own activities. Therefore, the sub-teams require support.

Naturally, a sub-team has to plan and perform a large number of activities. These activities may depend logically and temporally. Furthermore, not only may dependencies exist between the activities of one single sub-team, but also between activities of multiple sub-teams. Quantities and interdependencies of activities complicate the sub-teams capabilities of planning and scheduling: Statements regarding the duration of the overall project or about efficiently scheduling activities are not made easily by participants of the sub-teams. In order to ease

¹ The terms *network technique* or *network scheduling* (German: *Netzplantechnik*) subsume - amongst other concepts - the more known methods *CPM* (critical path method), *MPM* (metra potential method), *PERT* (program evaluation and review technique) or *PDM* (precedence diagramming method). For a short overview about network technique, see [6, pp. 101 - 108].

the sub-teams' project management, participants of sub-teams require support in modeling the complex system of linked activities.

Distributed software development projects can be executed by following the **Petri-net** and **agent-oriented software engineering** (PAOSE) approach [2,4]. In prior works, an issue tracking system and continuous integration support were integrated into PAOSE in order to support the participants in performing project management activities [3]. To allow scheduling of activities and reasoning about time properties further work is required. The current research of Röder [5] is concerned with the integration of time planning techniques into PAOSE.

The discipline of project management provides - among other things - the well-proven *network technique*. Using network technique, project managers are able to graphically model projects. Thereby, project activities and relationships are explicated as *net schedules* and elements of project flows. In addition, project managers can make use of the *critical path method* to obtain prioritization guidance by identifying activities that are critical in regard to the project duration.

In this poster we present a prototypical tool for applying concepts of a network technique to model sets of project activities as net schedules, enabling the identification of sequences of time-critical activities. The modeler can map net schedules to Petri nets, whereby the semantics can be altered in principle. This mapping has three benefits. Petri net analysis can be applied indirectly on net schedules. Project participants obtain explicit semantics about the net schedules they created. Furthermore, net schedules can indirectly be incorporated as executable Petri nets into the Petri net-based implementations of PAOSE.

In the future, the modeling tool can be enhanced in different ways: More concepts of the network technique can be implemented into the tool, to enrich the usable syntax. More complex semantics for the mapping of net schedules to Petri nets can be provided, to generate Petri net-based implementations for various purposes.

References

1. Pär Ågerfalk et al. A framework for considering opportunities and threats in distributed software development. In *Proceedings of the International Workshop on Distributed Software Development. Austrian Computer Society*, pages 47–61, August 2005.
2. Lawrence Cabac. *Modeling Petri Net-Based Multi-Agent Applications*, volume 5 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2010.
3. Matthias Güttler. Integration einer agilen Projektmanagementumgebung in ein verteiltes Team. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, November 2013.
4. PAOSE web page. <http://www.paose.net/>. Last accessed on 31st May, 2015.
5. Christian Röder. Prototypische Integration von Netzplantechnik in einen agentenorientierten und verteilten Softwareentwicklungsprozess. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, planned for September 2015.
6. Stefano Tonchia. *Industrial Project Management: Planning, Design, and Construction*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2008.

Applying Petri Nets to Approximation of the Euclidean Distance with the Example of SIFT

Jan Henrik Röwekamp and Michael Haustermann

University of Hamburg, MIN Faculty, Department of Informatics, Theoretical Computer Science Group and Cognitive Systems Group, Hamburg, Germany

<http://www.informatik.uni-hamburg.de/TGI/>

<http://kogs-www.informatik.uni-hamburg.de/>

Abstract. SIFT (Scale Invariant Feature Transform) is a complex image processing procedure for matching objects or patterns in images. Involving the computation of Euclidean distances in high dimensional space of many pairs of points and matching them against a threshold, this work proposes a speedup for the procedure utilizing colored Petri nets. Being sped up more pairs can be evaluated within reasonable time leading to better overall results.

Keywords: SIFT, object recognition, image processing, Euclidean distance, performance evaluation, binary squaring, complexity reduction, colored Petri nets

1 Introduction

Looking at the recent development of society towards omnipresence of computers coming in form of smart phones, handhelds, embedded systems and more the necessity of semi and fully automated processing of images to handle the amount of data and related requests produced by people becomes more and more severe. Efficient algorithms – in both running time as well as quality regard – cut down time spent by users to achieve their goals. SIFT [1](short for Scale Invariant Feature Transform) is a very powerful process for recognition of real world objects in an – from the queries point of view – unknown, heterogeneous image environment. At one critical point within the procedure it is necessary to compute the Euclidean distance between several pairs of high dimensional vectors and compare it against a certain threshold. Since this task is rather demanding in regards to computation time when done in the traditional way by just computing the distance, this work presents a method using Petri nets to lower the computation time for each distance calculation and comparison.

Object recognition as well as image processing in general is an extensive field of research with a lot of applications. Examples of applications benefiting from improved running times are **panorama photos** which a lot of modern mobile devices (smart phones) are able to take, **Optical character recognition (OCR)** in which an application for a live translator that automatically translates text seen in the real world is imaginable, **automated counting** of cars, people, coins, ... using images and/or video streams and also **assignment of photos** in a sense where objects depicted on a photo known to an online database, but unknown to the user, are matched.

As several other image processing algorithms, the procedure of SIFT involves calculating the Euclidean distance in high dimensional spaces of a significant amount of points.

Binary Squaring

One of the basic ideas in this work is the bit-wise squaring $bsq(n)$ of binary numbers which will be called "binary squaring". Assume $n = [42]_{10} = [0000101010]_2 = 2^1 + 2^3 + 2^5$. The binary square would be: $bsq(n) = [1092]_{10} = [0100100100]_2 = 2^2 + 2^6 + 2^{10}$. Obviously this value does not coincide with the correct squared value of 42 (1792).

There will follow some propositions related to binary squaring, which (analytical) proofs are omitted due to space constraints. For the factor $R_f(n)$ between binary square and conventional square of an integer n it holds: $1 \leq R_f(n) < 3$. The average sum of binary squares of uniform distributed random integers between two powers of two differs from the sum of conventional squares by factor 1.944 with high probability. This also holds for values between 0 and a power of two. The computation of the Euclidean distance can be approximated using the sum of binary squares times 1.944. The major advantage of the binary square version of this computation is, that summands (on the binary level) can be interchanged *before* computing the squared value. Thus the computation of the Euclidean distance using binary squares can be rearranged to handle most significant bits first being able to recognize distances above the threshold very fast.

The Petri Net Model

The general idea is to model the computation of the binary square based Euclidean distance with a colored Petri net. By doing so some of the computations can be done by the net design itself, for example using edge weights / (computations) and also the distance calculation can be split up and parallelized utilizing Petri nets' inherent concurrency by using the concept of binary squaring. The net consists of three major parts: The bit-splitting component, the bit analyzer component and a decision component (which decides whether the points are below or above the threshold). The bit splitter utilizes the results above and splits the differences in each dimension of two points into its bits, subsequently placing all bits of same significance i over all dimensions into the same place of the net k_i . The analyzer component contains a place which again contains the (classic) squared value divided by 1.944 of the desired threshold. The analyzer component continuously removes marks from the pool for each mark removed from a pool k_i . As soon as the pool runs out of marks, but there are still marks in one of the k_i to be processed the decision component decides "false" otherwise it decides "true".

As there are only a very few pairs below the threshold and the majority above (depends on the algorithm, that requires to calculate the Euclidean distance, but in most cases this holds), most of the computations will end in only a few firings instead of d squares and additions when looking at a d -dimensional space.

References

1. David G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, ICCV '99, pages 1150–1157, Washington, DC, USA, 1999. IEEE Computer Society.

Coordination Rules Generation from Coloured Petri Net Models

Adja Ndeye Sylla, Maxime Louvel, François Pacull

Univ. Grenoble Alpes, CEA, LETI, MINATEC Campus, F-38054 Grenoble, France
AdjaNdeye.sylla@cea.fr, maxime.louvel@cea.fr, francois.pacull@cea.fr

Abstract. This paper presents an environment to automatically generate coordination rules from coloured Petri nets models.

1 Introduction

Today's systems such as building automation or industrial control processes are composed of many heterogeneous and interacting components. These interactions raise problems such as data sharing and concurrent accesses. Coordination models and languages [4] are essential to provide a simple way to handle these interactions. LINC [3] is a rule based coordination environment. It is used to develop and deploy distributed applications. A LINC application is a set of rules enacted in distributed rule engines. LINC relies on powerful mechanisms such as transactions to ensure the normal execution of a rule. However, this does not prevent from writing conflicting rules. Currently, these conflicts are detected at execution time, when bugs are observed.

2 Contribution

The proposed approach consists in generating LINC rules from coloured Petri Net (CPN) models [2]. An application is first modelled using CPNs. This helps exchanges among all team members. Then the refined CPNs are verified to avoid undesired behaviours. Finally, the corresponding LINC rules are automatically generated and directly executed by LINC.

An application developed using LINC is a set of rules manipulating resources in several bags (bags are a distributed associative memory [1]). A resource is a tuple of **strings** and is manipulated using three operations: **rd**, **get**, and **put** to respectively verify the presence of resources, consume and insert resources. A rule consists of a precondition (i.e. verification of conditions) and a performance (i.e. actions to perform, atomically when the conditions are verified). To enable the automatic generation of LINC rules, we limit the colours. Colours defining states of the system are enumeration. Other tokens are tuples of strings.

To generate the rules, we define a transformation to move from CPN to LINC rules (Table 1a). Tokens in places are mapped to resources in bags. A transition is mapped to a performance. Indeed they both verify conditions and atomically

perform actions. An arc is mapped to an operation: both direction to **rd**, place to transition to **get** and transition to place to **put**. In LINC rule, the precondition explicitly specifies that the performance is triggerable when the specified resources are present. This is implicit in a CPN (i.e. a transition is enabled as soon as the specified tokens are present). Thus precondition is generated using the incoming arcs of a transition. To enable the graphical representation and the verification of LINC applications which were manually developed, we define the reverse transformation to move from LINC rules to CPN (Table 1b).

Coloured Petri net	LINC rule	LINC	Coloured Petri net
<i>Token</i>	<i>Resource</i>	<i>Resource</i>	<i>Token</i>
<i>Place</i>	<i>Bag</i>	<i>Bag</i>	<i>Place</i>
<i>Transition</i>	<i>Performance</i>	<i>Operation</i>	<i>Arc</i>
<i>Arc</i>	<i>Operation</i>	<i>type</i>	<i>orientation</i>
<i>orientation</i>	<i>type</i>	<i>Performance</i>	<i>Transition</i>
<i>Arcs incoming in a transition</i>	<i>Precondition</i>	<i>Precondition</i>	<i>Implicit in CPN</i>

(a) CPN model to LINC model

(b) LINC model to CPN model

Table 1: Defined transformations

Verification of CPN is limited in existing tools. Thus a CPN is first transformed to a PN by unrolling the colours with enumeration. Other places are left as is. Then, the generated PN is verified using existing model checkers. This enables to verify undesired behaviours such as deadclok and livelock.

3 Conclusion

This paper has presented a method to automatically generate rules from validated Coloured Petri Nets. CPNs verification ensures the global behaviour and LINC ensures that each step is actually executed according to the CPN. Hence distributed applications can safely be executed in actual distributed and embedded systems. This has been validated in a smart parking solution including several off-the-shelves hardware and software components.

References

1. N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 1989.
2. K. Jensen. *Coloured Petri nets: basic concepts, analysis methods and practical use*. Springer Science & Business Media, 2013.
3. M. Louvel and F. Pacull. Linc: A compact yet powerful coordination environment. In *Coordination Models and Languages*. Springer, 2014.
4. G. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in computers*, 1998.

