Proceedings of the 8th
Transformation Tool Contest

TTC 2015

Louis M. Rose
Tassilo Horn
Filip Křikava

Preface

The aim of the Transformation Tool Contest (TTC) series is to compare the expressiveness, the usability, and the performance of transformation tools along a number of selected case studies. A deeper understanding of the relative merits of different tool features will help to further improve transformation tools and to indicate open problems.

This contest was the eighth of its kind. For the third time, the contest was part of the Software Technologies: Applications and Foundations (STAF) federation of conferences. Teams from the major international players in transformation tool development have participated in an online setting as well as in a face-to-face workshop.

In order to facilitate the comparison of transformation tools, our programme committee selected the following three challenging cases via single blind reviews: the model execution case (for which eventually four solutions were accepted), the Java refactoring case (for which eventually six solutions were accepted), and the train benchmark case (for which eventually five solutions were accepted).

These proceedings comprise descriptions of the three cases and descriptions of all of the solutions to these cases. In addition to the solution descriptions contained in these proceedings, the implementation of each solution (tool, project files, documentation) is made available for review and demonstration via the SHARE platform (http://share20.eu).

TTC 2015 involved open (i.e., non anonymous) peer reviews in a first round. The purpose of this round of reviewing was that the participants gained as much insight into the competitors' solutions as possible and also to identify potential problems. At the workshop, the solutions were presented. The expert audience judged the solutions along a number of case-specific categories, and prizes were awarded to the highest scoring solutions in each category. A summary of these results for each case are included in these proceedings. Finally, the solutions appearing in these proceedings were selected by our programme committee via single blind reviews. The full results of the contest are published[1] on our website.

Besides the presentations of the submitted solutions, the workshop also comprised a live contest. That contest involved a set of tasks for processing Java annotations in source level transformations. The live contest was announced to all STAF attendees and participants were given four days to design, implement and test their solutions.

The contest organisers thank all authors for submitting cases and solutions, the contest participants, the STAF local organisation team, the STAF general chair Alfonso Pierantonio, and the program committee for their support.

24th July, 2015
l'Aquila, Italy

Louis M. Rose
Tassilo Horn
Filip Křikava

[1] `http://www.transformation-tool-contest.eu`

Organisation

TTC 2015 was organised by the Universita degli Studi dell'Aquila, Italy.

Program Committee

Rubby Casallas	University of los Andes, Columbia
Coen De Roover	Vrije Universiteit Brussel, Belgium
Jeff Gray	University of Alabama, USA
Tassilo Horn	University Koblenz-Landau, Germany
Ákos Horvth	BME, Hungary
Christian Krause	SAP Innovation Center Potsdam, Germany
Filip Krikava	Czech Technical University, Czech Republic
Sonja Maier	Universitaet der Bundeswehr Muenchen, Germany
Richard Paige	University of York, UK
Arend Rensink	University of Twente, Netherlands
Louis Rose	University of York, UK
Bernhard Schaetz	TU München, Germany
Eugene Syriani	University of Montreal, Canada
Tijs Van Der Storm	Centrum Wiskunde & Informatica, Netherlands
Pieter Van Gorp	Eindhoven University of Technology, Netherlands
Gergely Varro	Technische Universität Darmstadt, Germany
Bernhard Westfechtel	University of Bayreuth, Germany

Table of Contents

The Model Execution Case

The Java Refactoring Case

The Train Benchmark Case

Part I.
The Model Execution Case

The TTC 2015 Model Execution Case

Tanja Mayerhofer and Manuel Wimmer

Business Informatics Group, Vienna University of Technology, Austria
{mayerhofer,wimmer}@big.tuwien.ac.at

Abstract. This paper describes a case study for the Transformation Tool Contest (TTC) 2015 concerning the execution of models. The case foresees the specification of the operational semantics of a subset of the UML activity diagram language with transformation languages. In particular, the computation of the end result of the execution of the activity diagrams is targeted as well as the provisioning of a precise trace for the complete execution. The evaluation concerns the correctness of the operational semantics specifications, its understandability and conciseness, as well as its performance.

1 Introduction

Executable models are being used for decades in computer science since the introduction of Petri nets and state machines to name just a few prominent examples. In the past years, they also became with Executable Domain Specific Modeling Languages (xDSMLs) and executable UML (xUML) an important research line in Model-Driven Engineering (MDE) [1, 2]. As a prerequisite for having executable models, the operational semantics of the modeling languages have to be explicated. In general, there are two approaches used for defining the operational semantics of models [5]. First, one may incorporate the runtime concepts into the metamodel of the modeling languages to represent execution states and define transformation rules for evolving the execution states of a model. Second, one may delegate the execution of models by mapping the modeling languages to some existing formalisms which already provide execution support. The second approach has been already covered in the past by previous TTC cases. However, the first approach has not been subject to investigation for a TTC case yet.

We believe that having a dedicated TTC case for the direct specification of the operational semantics within a language's metamodel is of major interest for the transformation community due to two reasons. First, there is already a large body of work discussing how to implement the operational semantics for modeling languages using different kinds of languages including also several model transformation languages (cf. for instance [3,4,6]). Second, with efforts such as fUML [8] and xDSMLs [1], language engineers have to reside on mature techniques to define the operational semantics for their modeling languages in a concise, reusable, and scalable way.

To shed some light on the current state-of-the-art of defining operational semantics with model transformations, we propose in this case description the task of specifying the operational semantics of a subset of the UML activity diagram language covering several control flow concepts and a simple expression language. The provided metamodel for this subset of the UML activity diagram language already contains the necessary runtime concepts. So the task for TTC attendees is to define a transformation,

which specifies the operational semantics of the UML activity diagram language by updating the runtime state of executed UML activity diagrams. Thus, the input model for the transformation is a UML activity diagram in its initial runtime state and the output model of the transformation is the final runtime state of the UML activity diagram including a trace of the execution. Due to these characteristics, the transformation is considered to be in-place and endogenous [7]. Please note that the transformation is only concerned with the abstract syntax of the models, which may trigger modifications in the concrete syntax, e.g., for model animation. However, the latter is not in the focus of this case.

2 The Transformation

This section describes the artifacts needed for solving this case, namely the UML activity diagram metamodel, a description of its operational semantics, as well as an example transformation trace.

2.1 Metamodel

Figure 1 shows an excerpt of the metamodel of the activity diagram variant considered in this case. It shows the basic concepts for modeling activities. Activities (metaclass Activity) consist of variables (metaclass Variable), activity nodes (metaclass ActivityNode), and activity edges (metaclass ActivityEdge).

Variables. For variables we distinguish between Integer variables and Boolean variables (metaclasses IntegerVariable and BooleanVariable). Variables may define an initial value (reference initialValue), where we again distinguish between Integer values and Boolean values (metaclasses Value, IntegerValue, and BooleanValue). Variables can serve as local variables or input variables of an activity (references locals and inputs of Activity).

Activity Nodes. There are two types of activity nodes available, namely control nodes (metaclass ControlNode) and actions (metaclass Action).

Control nodes can be used to define the start of an activity (metaclass InitialNode), the end of an activity (metaclass ActivityFinalNode), alternative branches of an activity (metaclasses DecisionNode and MergeNode), and concurrent branches of an activity (metaclasses ForkNode and JoinNode).

Actions constitute the fundamental unit of executable behavior and their execution represents some processing in the modeled system. In this case we consider so-called opaque actions (metaclass OpaqueAction), which can define an ordered sequence of expressions (metaclass Expression). Which kinds of expressions are supported, can be seen in the excerpt of the metamodel depicted in Figure 2.

We distinguish between Integer expressions and Boolean expressions (metaclasses IntegerExpression and BooleanExpression). An Integer expression processes two Integer variables (references operand1 and operand2). Integer calculation expressions (metaclass IntegerCalculationExpression) either perform a summation or subtraction

of these variables (enumeration IntegerCalculationOperator) and assign the resulting value to another Integer variable (reference assignee). Integer comparison expressions (metaclass IntegerComparisonExpression) compare the variables according to the defined comparison operator (enumeration IntegerComparisonOperator) and assign the resulting value to a Boolean variable (reference assignee). For Boolean expressions we distinguish between unary expressions and binary expressions (metaclasses BooleanUnaryExpression and BooleanBinaryExpression) that assign the resulting value to a Boolean variable (reference assignee). Unary Boolean expressions apply the logical operator *NOT* (enumeration BooleanUnaryOperator) on a Boolean variable (reference operand). Binary expressions apply the logical operators *AND* and *OR* on two Boolean variables (references operand1 and operand2). Please note that computed values cannot be assigned to input variables of activities.

Activity Edges. Activity edges are used to connect activity nodes with each other. Control flow edges (metaclass ControlFlow) define the flow of control among activity nodes. They may define a Boolean variable whose value serves as guard condition for the control flow (reference guard). Guard conditions are only allowed for outgoing control flow edges of decision nodes.

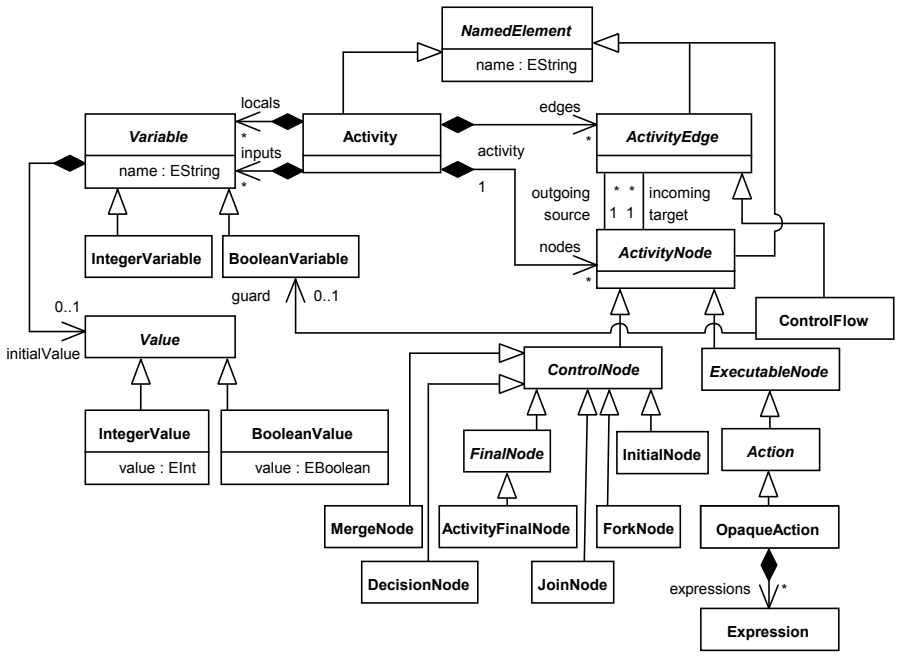


Fig. 1. Metamodel for UML activity diagrams (activities, variables, edges, nodes)

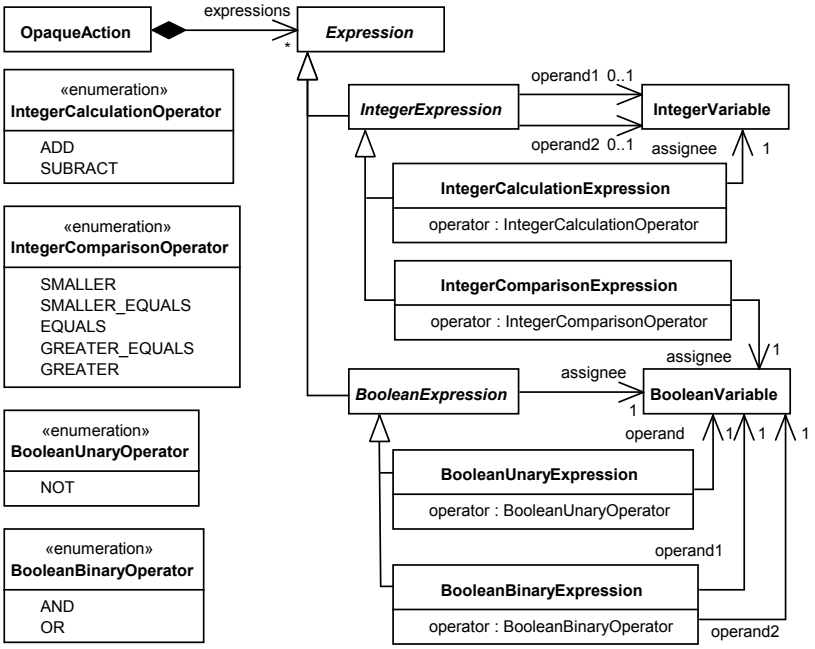


Fig. 2. Metamodel for UML activity diagrams (expressions)

2.2 Operational Semantics

An operational semantics defines the semantics of a modeling language by specifying the steps of computation required for executing a model conforming to the modeling language. This means, that an operational semantics defines an interpreter for the modeling language, which can be regarded as state transition system defining how an executing model progresses from state to state. Therefore, an operational semantics consists of two parts: *(i)* the definition of the *runtime concepts* needed for capturing the state of an executing model and *(ii)* the definition of the *steps of computation* involved in performing transitions of the executing model from one state to another state. Please note that runtime concepts may include also the definition of additional input values required for the execution of a model.

While the runtime concepts needed for defining the state of an executing model can be defined by applying metamodeling techniques, the steps of computation progressing the executing model to a new state has to be defined with transformation languages.

In the following, we discuss the runtime concepts and steps of computation defining the operational semantics of the UML activity diagram language. Please note that the defined operational semantics are based on the semantics of fUML [8].

Runtime Concepts. Figure 3 shows the metamodel defining the runtime concepts for the UML activity diagram language considered in this case. The runtime concepts are

depicted in orange color, while the metaclasses are depicted in white color. We distinguish between four types of runtime concepts: runtime concepts for capturing *(i)* the token flow among activity nodes, *(ii)* the current values of variables, *(iii)* the trace of an activity diagram, and *(iv)* input values that may be provided to an activity diagram.

The semantics of activity diagrams is based on a definition of token flow semantics similar to the token flow semantics of Petri nets (cf. [9] for a formalization of the semantics of UML activity diagrams using Petri nets as semantic domain). Informally speaking, an activity node is executed, when all required control tokens are available through incoming control flow edges. After the execution of an activity node is completed, control tokens are offered to the successor nodes via outgoing control flow edges. The runtime concept Token and its subclasses ControlToken and ForkedToken define how tokens are represented during execution. Thereby, forked tokens originate from the execution of fork nodes, splitting a control flow (reference baseToken) into multiple concurrent flows. Tokens are always owned by the activity node (reference heldTokens of ActivityNode) offering the tokens via activity edges to successor nodes (reference offers of ActivityEdge). The representation of token offers is defined by the runtime concept Offer.

Variables defined for an activity are initialized with their initial value (cf. Figure 1, reference initialValue of Variable), meaning that the value is copied and set as current value of the variable (reference currentValue). The current value of variables is updated during the execution of an activity by the execution of opaque actions defining assignments to these variables.

For capturing tracing information, the runtime concept Trace is defined keeping an ordered list of executed activity nodes (reference executedNodes).

For input variables of an activity (cf. Figure 1, reference inputs of Activity), input values to be processed by the execution of the activity may be provided. For representing these input values, the runtime concepts Input and InputValue are defined.

Please note that the defined runtime concepts extends the metamodel of the UML activity diagram language, i.e., they extend the metaclasses defined in the metamodel with additional attributes and references, and add additional metaclasses to the metamodel. This can, for instance, be done with the *package merge* operation known from UML and MOF. In our reference implementation, we introduced them directly into the metamodel of the UML activity diagram language.

Steps of Computation. The following steps of computation are defined for executing activity diagrams.

1. *Initialization of Variables.* The executed activity receives input values (cf. Figure 3, metaclass InputValue) for its defined input variables and initializes the current values of the input variables accordingly (cf. Figure 3, reference currentValue of Variable). Furthermore, also the current values of the local variables are initialized according to their initial values (cf. Figure 1, reference initialValue of Variable).
2. *Setting Activity Nodes as Running.* All nodes contained by the executed activity are set as being running (cf. Figure 1, attribute running of ActivityNode).
3. *Execution of Initial Node.* The initial node of the activity is executed. The execution of the initial node consists of producing a single control token, adding this control

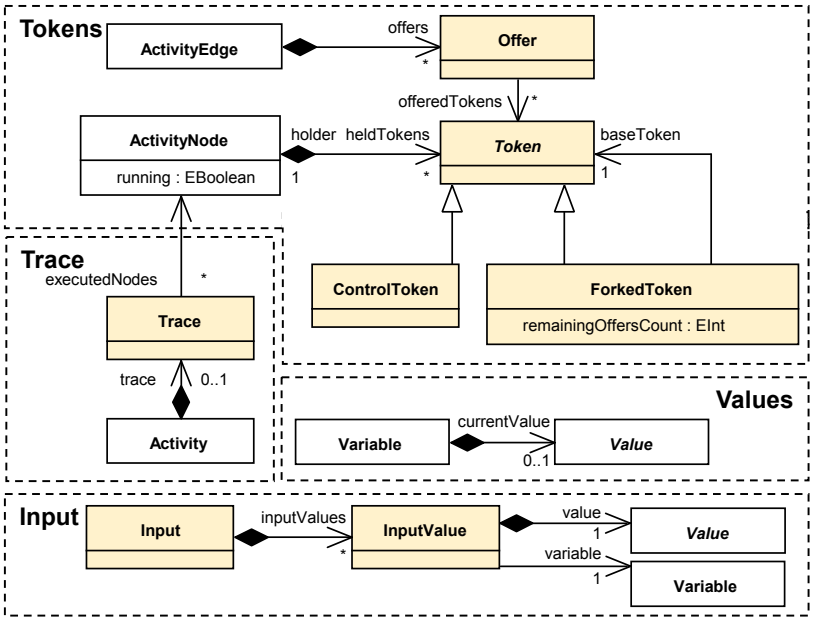


Fig. 3. Runtime concepts of UML activity diagrams

token to the initial node (cf. Figure 3, reference heldTokens of ActivityNode), and offering it via the outgoing control flow edges of the initial node to successor nodes (cf. Figure 3, reference offers of ActivityEdge). Please note that an activity has to contain exactly one initial node.

4. *Determination of Enabled Nodes.* The currently enabled nodes of the activity are determined. An activity node is enabled if it is set as being running and if all incoming control flow edges provide token offers. In the case of merge nodes, only one of the incoming control flow edges has to provide a token offer.
5. *Selection and Execution of One Enabled Node.* One of the enabled nodes is selected and executed. Executing an enabled node consists of three steps:
   i *Consumption of Offered Tokens.* All tokens offered to the node via incoming control flow edges (in the following also referred to as *incoming tokens*) are consumed. This leads to the removal of all token offers of all incoming control flow edges (cf. Figure 3, reference offers of ActivityEdge). Furthermore, in the case of a consumed control token, the control token is removed from the offering (i.e., preceding) node (cf. Figure 3, reference heldTokens of ActivityNode). In the case of a consumed forked token, the forked token's remaining offers count is decremented by one (cf. Figure 3, attribute remainingOffersCount) and only if the remaining offers count is then equal to zero (i.e., every successor node of a fork node has processed the forked token), the forked token is removed from the offering node. Furthermore, if the forked token's base token (cf. Figure 3, reference baseToken) has not be removed from its offering node

(cf. Figure 3, reference holder of Token), the base token is removed from its offering node (independent of the remaining offers count).
Please note, that the operational semantics does not support implicit forking. For instance, consider an initial node having two succeeding actions (i.e., the initial node has two outgoing control flow edges each leading to a distinct action). In this case, the single control token produced and held by the initial node is offered by two offers—one offer for each outgoing control flow edge—to the two succeeding actions. However, only one of the actions can be executed, because the execution of one action will remove the token from the initial node making the offer to the other action obsolete, i.e., the token cannot be consumed by this second action anymore and it can thus not be executed.

ii *Execution of the Node's Behavior.* After the consumption of offered tokens, the behavior of the node is executed. Depending on the type of the node, control tokens may be produced, which are added to the node (cf. Figure 3, reference heldTokens of ActivityNode) and offered to successor nodes via the node's outgoing control flow edges (cf. Figure 3, reference offers of ActivityEdge).

iii *Tracing.* Furthermore, the executed node is added to the trace kept by the executed activity (cf. Figure 3, reference trace of Activity and reference executedNodes of Trace).

The behavior of the different types of activity nodes executed in Step *(ii)* is defined as follows:

(a) *Opaque Actions.* The expressions defined by an opaque action are executed in sequential order. The semantics of expressions consists of applying the defined operator on the current values of the defined operand variables and assigning the resulting value to the defined assignee variable as current value.
After all expressions have been executed, one control token is created for each outgoing control flow edge and offered to successor nodes via the respective edge.

(b) *Fork Nodes.* A fork node produces for each incoming token a forked token, whose base token is set to the corresponding incoming token and whose remaining offers count is set to the number of outgoing edges (cf. Figure 3, reference baseToken and attribute remainingOffersCount of ForkedToken). The created forked tokens are offered via all outgoing control flow edges of the fork node.

(c) *Decision Nodes.* A decision node evaluates the guard conditions of its outgoing control flow edges, i.e., it determines whether the Boolean variables defined as guard conditions have the Boolean value *true* set as current value. The incoming tokens are offered via the edge whose guard condition is fulfilled. Please note that only one guard condition is allowed to be fulfilled.

(d) *Join Nodes and Merge Nodes.* Join nodes and merge nodes offer the incoming tokens on all outgoing control flow edges.

(e) *Activity Final Nodes.* An activity final node terminates the execution of the containing activity causing all nodes contained by the activity to be set as not running (cf. Figure 3, attribute running of ActivityNode).

6. *Repetition of Steps 4 and 5 until Termination.* Step 4 and Step 5 are repeated until no activity nodes are enabled anymore constituting the termination of the activity.

Please note that in case an activity final node has been executed, no node is enabled anymore, because all nodes are set as not running.

In the provided reference implementation, the steps of computation are implemented using Java and EMF. Therefore, we introduced operations into the Java classes generated by EMF for the presented metamodel of the UML activity diagram language. In the following, we provide an overview of the most important operations.

Activity

```
/*
 * Receives input values for the activity's input variables
 * and starts the execution of the activity.
 */
void main(List<InputValue> inputValues);
/*
 * Initializes the activity's local and input variables.
 */
void initialize(List<InputValue> inputValues);
/*
 * 1. Sets all nodes of an activity as running.
 * 2. Fires the initial node.
 * 3. Determines the currently enables nodes.
 * 4. Selects one of enabled nodes and fires it.
 * 5. Repeats steps 3 and 4 until no node is enabled anymore.
 */
void run();
```

Activity Node

```
/*
 * Returns true if the node is enabled, false otherwise.
 */
boolean isReady();
/*
 * Consumes all tokens provided via incoming edges.
 */
List<Token> takeOfferedTokens();
/*
 * Adds tokens to node (heldTokens).
 */
void addTokens(List<Token> tokens);
/*
 * Executes the behavior of the node.
 */
void fire(List<Token> tokens);
/*
 * Offers tokens via all outgoing edges.
 */
void sendOffers(List<Token> tokens);
/*
```

```
 * Removes token from node (heldTokens).
 */
void removeToken(Token token);
/*
 * Returns true, if all incoming edges offer tokens,
 * false otherwise.
 */
boolean hasOffer();
```

Activity Edge

```
/*
 * Returns all offered tokens and destroys all offers.
 */
List<Token> takeOfferedTokens();
/*
 * Creates offer for provided tokens.
 */
void sendOffer(List<Token> tokens);
```

Action

```
/*
 * Executes an action.
 */
void doAction();
```

Expression

```
/*
 * Executes expression.
 */
void execute();
```

2.3 Example Transformation Execution Trace

Figure 4 shows an example of an activity diagram in UML notation. Please note that we provide with our reference implementation a textual concrete syntax for the UML activity diagram language, which is used for defining the UML activity diagrams used in our test suite (cf. Appendix A).

The activity shown in Figure 4 defines two variables: the input variable *internal* and the local variable *noninternal* both of type Boolean. The local variable *noninternal* is initialized with the value *false*. Furthermore, the activity consists of one initial node, one decision node, one fork node, one join node, one merge node, one activity final node and eight opaque actions, which are connected by 15 control flow edges. Noteworthy about the activity is, that the opaque action *register* defines one expression notinternal = ! internal and that the outgoing edges of the decision node define the two variables as guard conditions.

Figure 5 and Figure 6 visualize and explain the execution of this example activity diagram in the case that the value *true* is provided as input for the input variable *internal*. These figures illustrate the execution of the activity nodes contained by the UML

activity diagram one by one. Control tokens and offers of control tokens are shown in orange color. Forked tokens and offers of forked tokens are shown in blue color. Updates of important features are also highlighted in color. The execution is shown until the execution of the action *manager interview*. The complete trace of the example is as follows: initial node *initial* - opaque action *register* - decision node *decision* - opaque action *get welcome package* - fork node *fork* - opaque action *assign to project* - opaque action *add to website* - join node *join* - opaque action *manager interview* - opaque action *manager report* - merge node *merge* - opaque action *authorize payment* - activity final node *final*. Please note, that the opaque actions *assign to project* and *add to website* could also be executed in reverse order.

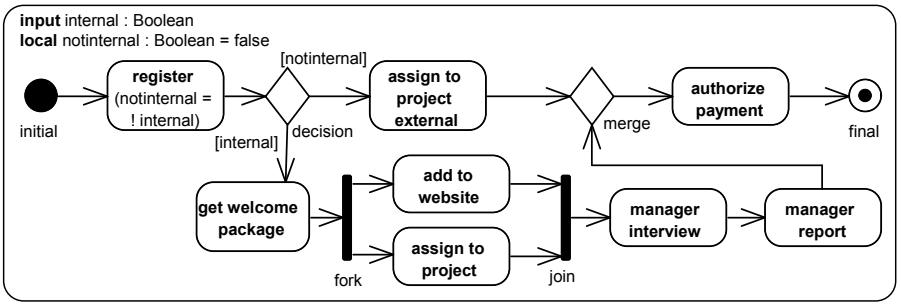


Fig. 4. Example activity diagram (UML notation)

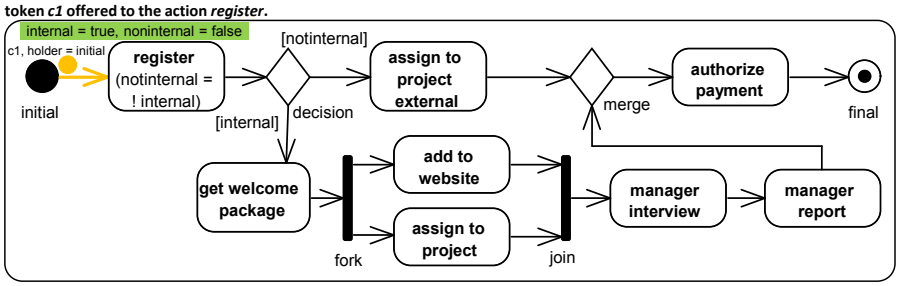
2.4 Variations

As the presented UML activity diagram language is quite extensive, solution developers may choose to implement it only partially. We foresee the following three case variations.

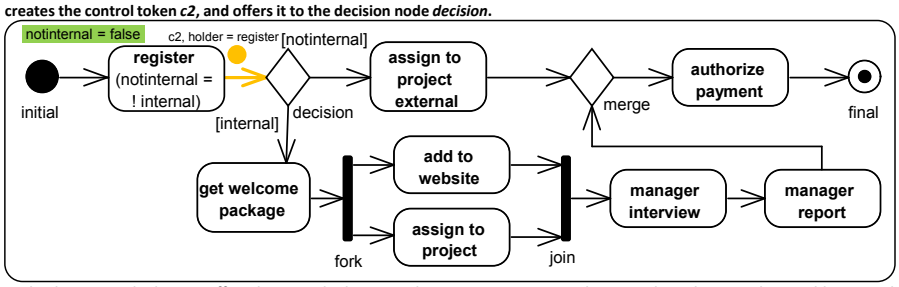
Variant 1: Simple Control Flow. The first variant considers only the following concepts of the UML activity diagram language: Activity, initial node, activity final node, opaque action (without expressions), control flow edge. This means that only the operational semantics of these concepts has to be implemented by solution developers choosing this case variant. The following runtime concepts have to be implemented for this variant: Offer, token, control token, trace. We consider this subset of concepts to be the minimal one that should be implemented by all solution developers.

Variant 2: Complex Control Flow. The second variant considers compared to the first variant the following additional concepts: Fork node, join node, decision node, merge node, local Boolean variables, and Boolean values. Only the runtime concept forked token as well as current values of Boolean variables have to be implemented additionally compared to the first variant.

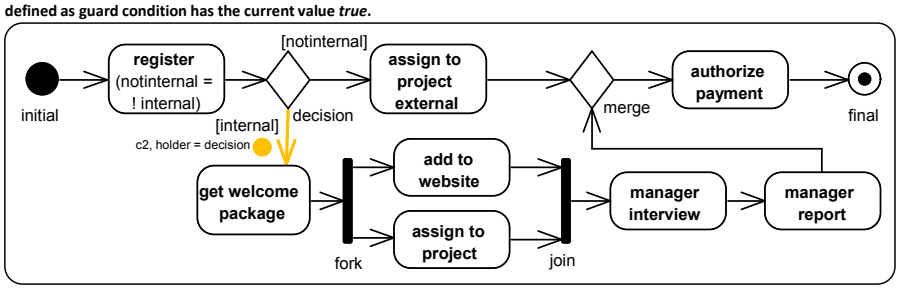
1. The variables are initialized, the nodes are set as running, the initial node is executed leading to the creation of the control token *c1* offered to the action *register*.



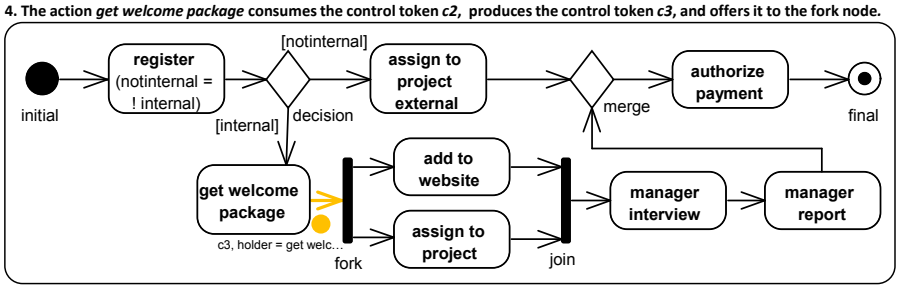
2. The action *register* consumes the token *c1*, executes the defined expression leading to an update of the variable *non-internal*, creates the control token *c2*, and offers it to the decision node *decision*.



3. The decision node *decision* offers the control token *c2* to the opaque action *get welcome package*, because the variable *internal* defined as guard condition has the current value *true*.



4. The action *get welcome package* consumes the control token *c2*, produces the control token *c3*, and offers it to the fork node.



5. The fork node *fork* produces the forked token *c4* for the incoming control token *c3* (i.e., the forked token's base token). The remaining offers count is set to 2, because the fork node has two outgoing control flow edges. The forked token c4 is offered to the successor actions via two distinct offers.

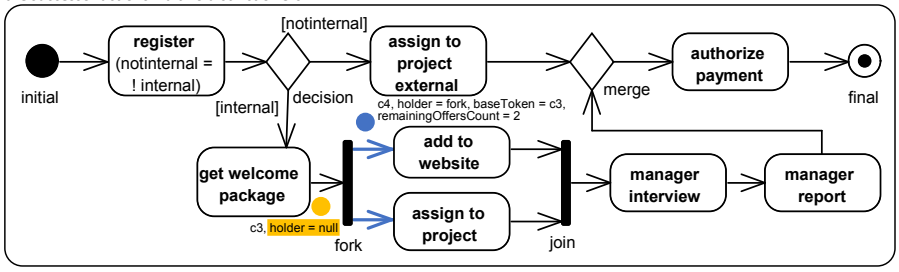
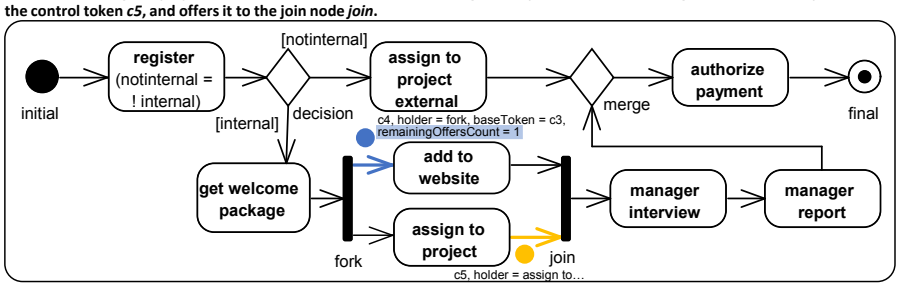
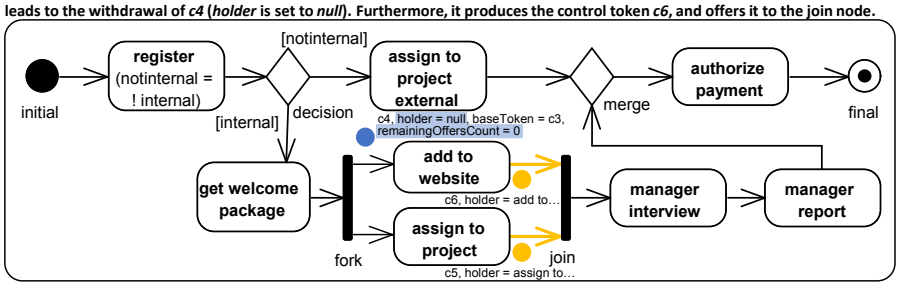


Fig. 5. Visualization of the execution of the example activity diagram (part 1)

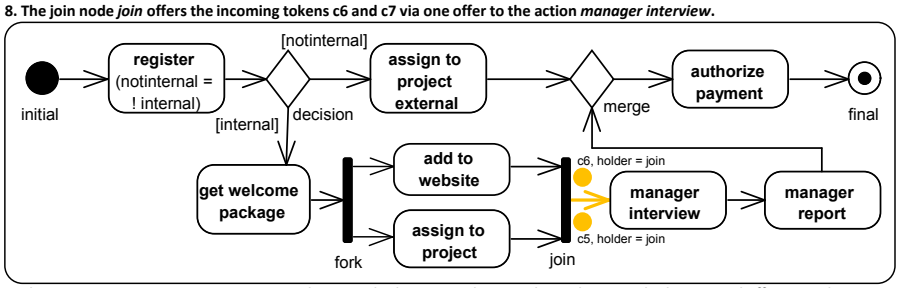
6. The action *assign to project* consumes its token offer for *c4* leading to an update of *c4*'s remaining offers count to 1, produces the control token *c5*, and offers it to the join node *join*.



7. The action *add to website* consumes its token offer for *c4* leading to an update of *c4*'s remaining offers count to 0, which in turn leads to the withdrawal of *c4* (*holder* is set to *null*). Furthermore, it produces the control token *c6*, and offers it to the join node.



8. The join node *join* offers the incoming tokens c6 and c7 via one offer to the action *manager interview*.



9. The action *manager interview* consumes the control tokens *c5 and c6*, produces the control token *c7*, and offers it to the action *manager report*.

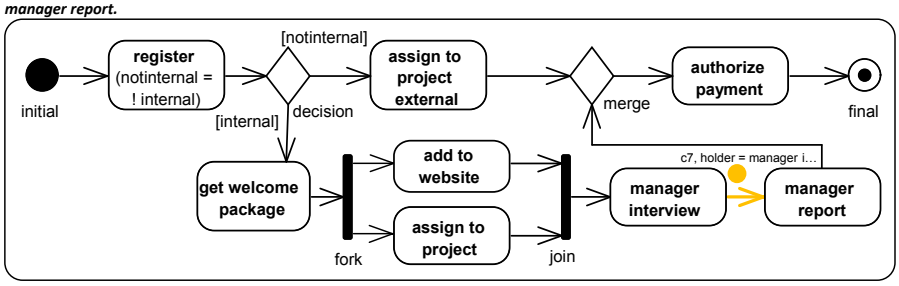


Fig. 6. Visualization of the execution of the example activity diagram (part 2)

Variant 3: Expressions The third variant considers the complete UML activity diagram language. Thus, the following additional concepts have to be implemented compared to the second variant: Input variables, Integer variables, Integer values, and all expression types. Also the runtime concepts input and input value have to be implemented in addition.

2.5 Reference Implementation

The reference implementation for this case may be found at the following open source code repository: `https://github.com/moliz/moliz.ttc2015`. It consists of the following Eclipse plug-in projects.

org.modelexecution.operationalsemantics.ad: Contains the metamodel of UML activity diagram language including definitions of the runtime concepts required as part of its operational semantics, and the Java code generated by EMF for the metamodel including the steps of computation of the operational semantics.

org.modelexecution.operationalsemantics.ad.test: Provides a test suite for verifying the correctness of the implemented operational semantics. The test suite is explained in Section 3.

org.modelexecution.operationalsemantics.ad.grammar,
org.modelexecution.operationalsemantics.ad.grammar.ui,
org.modelexecution.operationalsemantics.ad.input.grammar,
org.modelexecution.operationalsemantics.ad.input.grammar.ui: Xtext projects implementing a textual concrete syntax for activity diagrams and input values.

For running the reference implementation, the *Eclipse Modeling Tools (version Luna)*[1] are required. Furthermore, the *Xtext plug-in* for Eclipse[2] has to be installed additionally.

3 Test Suite

As part of the reference implementation, we provide a test suite meant for evaluating the correctness and performance of the implemented operational semantics. Each test case contained by the test suite executes a single activity and asserts the resulting trace. If local variables are manipulated by the activity, also their final values are asserted. The provided test cases are described in the following.

test1: Tests the operational semantics of initial nodes, opaque actions, activity final nodes, and control flow edges.

test2: Tests the operational semantics of fork nodes and join nodes.

[1] Eclipse Modeling Tools may be downloaded from `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/lunasr2`.
[2] Xtext can be installed in Eclipse using the menu *Help - Install Modeling Components*.

test3: Tests the operational semantics of decision nodes, merge nodes, and local Boolean variables.

test4: Tests the operational semantics of expressions.

test5: Tests the operational semantics of input variables.

test6: Defines the example activity diagram explained in Section 2.3.

testperformance_variant1: Performance test for variant 1 of this case. The UML activity diagram of this test cases comprises 1,000 sequential opaque actions.

testperformance_variant2: Performance test for variant 2 of this case. The UML activity diagram of this test cases comprises 100 concurrent branches each one comprising 10 opaque actions.

testperformance_variant3_1: Performance test for variant 3 of this case. Like the test case *testperformance_variant2*, the UML activity diagram of this test cases comprises 100 concurrent branches each one comprising 10 opaque actions. Furthermore, for each concurrent branch one variable exists that is incremented by the opaque actions lying on this branch.

testperformance_variant3_2: Performance test for variant 3 of this case. The UML activity diagram of this test case comprises 18 activity nodes including a loop introduced through decision and merge nodes. Due to the loop, 1,001 activity node executions occur.

Solution developers, who chose to implement variant 1 of the case only have to consider the test cases *test1*, and *testperformance_variant1*. For variant 2, the test cases *test2*, *test3*, and *testperformance_variant2* have to be considered additionally. Solutions for variant 3 have to consider all test cases.

A run configuration for executing all test cases is provided by the test project (*org.modelexecution.operationalsemantics.ad.test/TestSuite.launch*). In case a solution builds upon the Eclipse Modeling Tools and uses the provided metamodel, it is only required to override or modify the operation *executeActivity(String modelPath, String inputPath)* of the class *TestSuite* located in the project *org.modelexecution.operationalsemantics.ad.test*. This operation is responsible for loading the UML activity diagram to be executed as well as the activity diagram's input values, executes the UML activity diagram, and provides the trace captured during the execution as output. However, if a solution does not build upon the Eclipse Modeling Tools or does not use the provided metamodel, the test suite has to be re-implemented accordingly by the solution developers to demonstrate the solution's correctness and assess its performance.

The test project also contains textual representations of the traces obtained by the reference implementation for the UML activity diagrams of all test cases (folder *trace*). This textual representation comprises the execution order of the activity nodes of the respective UML activity diagram as well as the final values of local variables. Please note, that in the case of concurrent branches in the activity diagram, the execution order captured by the provided trace represent only *one* valid execution order.

4 Evaluation Criteria

The task of this case is to use a model transformation tool to implement the described operational semantics for a subset of the UML activity diagram and to execute models conforming to this language. Submissions are evaluated according to the following criteria.

4.1 Correctness

It is mandatory that solutions demonstrate that they have the intended behavior for all test cases covered by the test suite described in Section 3.

4.2 Understandability and Conciseness

Peer reviews will be used to assess qualitatively the conciseness and understandability of all solutions. We envision online reviews involving multiple rounds in order to reach consensus among all participants. Understandability and conciseness are used as measures to reason also about the implementation effort for the operational semantics specifications.

4.3 Performance

Performance is measured by logging how long the developed transformations need to execute the models, i.e., to produce the final runtime state. Therefore, the test cases *testperformance_variant1*, *testperformance_variant2*, *testperformance_variant3_1*, and *testperformance_variant3_2* described in Section 3 are used, depending on the case variant chosen by the solution developers. The execution time should be measured in milliseconds, e.g., in Java using java.lang.System.nanoTime(). Please note that reading the input model and writing the output model is not considered to be part of this performance evaluation.

4.4 Overall Assessment

The final score for submitted solutions will be calculated by summing up a maximum of 100 points in total, while for the correctness, understandability and conciseness, and performance, 25, 50, 25 points are reserved, respectively. Therewith, we want to strongly emphasize the importance of having concise and understandable operational semantics specifications in order to best support language engineers in developing, maintaining, and extending such kind of specifications as they are considered to be one of the most critical artifacts in the language engineering process.

References

1. Bryant, B.R., Gray, J., Mernik, M., Clarke, P.J., France, R.B., Karsai, G.: Challenges and Directions in Formalizing the Semantics of Modeling Languages. Computer Science and Information Systems 8(2), 225–253 (2011)

2. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (2009)
3. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Proceedings of the 3rd International Conference on The Unified Modeling Language (UML). LNCS, vol. 1939, pp. 323–337. Springer (2000)
4. Jézéquel, J., Barais, O., Fleurey, F.: Model Driven Language Engineering with Kermeta. In: Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE). LNCS, vol. 6491, pp. 201–221. Springer (2009)
5. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Model Transformation Intents and Their Properties. Software & Systems Modeling pp. 1–35 (2014)
6. Mayerhofer, T., Langer, P., Wimmer, M., Kappel, G.: xMOF: Executable DSMLs Based on fUML. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE), LNCS, vol. 8225, pp. 56–75. Springer (2013)
7. Mens, T., Van Gorp, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science (ENTCS) 152, 125–142 (2006)
8. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1 (August 2013), http://www.omg.org/spec/FUML/1.1
9. Syriani, E., Ergin, H.: Operational Semantics of UML Activity Diagram: An Application in Project Management. In: Proceedings of the 2nd Workshop on Model-Driven Requirements Engineering Workshop (MoDRE). pp. 1–8. IEEE (2012)

A Textual Concrete Syntax

For defining UML activity diagrams more conveniently, we provide a textual concrete syntax implemented with Xtext. Listing 1 shows the exemplary UML activity diagram discussed in Section 2.3 and shown in Figure 4 in this textual concrete syntax.

```
activity Test7 (bool internal) {
  bool notinternal = false

  nodes {
    initial initialNode7 out(edge42),
    action register comp {notinternal = !internal} in(edge42) out(edge43),
    decision decisionInternal in(edge43) out(edge44, edge45),
    action assignToProjectExternal in(edge44) out(edge56),
    action getWelcomePackage in(edge45) out(edge46),
    fork forkGetWelcomePackage in(edge46) out(edge47, edge48),
    action assignToProject in(edge47) out(edge49),
    action addToWebsite in(edge48) out(edge50),
    join joinManagerInterview in(edge49, edge50) out(edge51),
    action managerInterview in(edge51) out(edge52),
    action managerReport in(edge52) out(edge53),
    merge mergeAuthorizePayment in(edge53, edge56) out(edge54),
    action authorizePayment in(edge54) out(edge55),(*final finalNode7 in(edge55)
  }

  edges {
    flow edge42 from initialNode7 to register,
    flow edge43 from register to decisionInternal,
    flow edge44 from decisionInternal to assignToProjectExternal [notinternal
        ],
    flow edge45 from decisionInternal to getWelcomePackage [internal],
    flow edge46 from getWelcomePackage to forGetWelcomePackage,
```

```
    flow edge47 from forkGetWelcomePackage to assignToProject,
    flow edge48 from forkGetWelcomePackage to addToWebsite,
    flow edge49 from assignToProject to joinManagerInterview,
    flow edge50 from addToWebsite to joinManagerInterview,
    flow edge51 from joinManagerInterview to managerInterview,
    flow edge52 from managerInterview to managerReport,
    flow edge53 from managerReport to mergeAuthorizePayment,
    flow edge54 from mergeAuthorizePayment to authorizePayment,
    flow edge55 from authorizePayment to finalNode7,
    flow edge56 from assignToProjectExternal to mergeAuthorizePayment
  }
}
```

Listing 1. Example activity diagram (textual concrete syntax)

A Solution to the TTC'15 Model Execution Case Using the GEMOC Studio

Benoit Combemale
INRIA
benoit.combemale@inria.fr

Julien DeAntoni
UNS, I3S/INRIA
julien.deantoni@polytech.unice.fr

Olivier Barais
University of Rennes 1, IRISA
olivier.barais@irisa.fr

Arnaud Blouin
INSA Rennes, IRISA
arnaud.blouin@irisa.fr

Erwan Bousse
University of Rennes 1, IRISA
erwan.bousse@irisa.fr

Cédric Brun
OBEO
cedric.brun@obeo.fr

Thomas Degueule
INRIA
thomas.degueule@inria.fr

Didier Vojtisek
INRIA
didier.vojtisek@inria.fr

We present in this paper a complete solution to the Model Execution case of the Transformation Tool Contest 2015 using *the GEMOC Studio*. The solution proposes an implementation of the most complete version (variant 3) of the UML Activity Diagram language. The implementation uses different technologies integrated into the GEMOC Studio for implementing the various concerns of the language: *Kermeta* is used to modularly implement the operational semantics and to weave it into the provided metamodel, *Melange* is used to build the overall language runtime seamlessly integrated to EMF, *Sirius Animator* is used to develop a graphical animator, *the GEMOC execution engine* is used to execute the conforming models according to the operational semantics and to build a rich and efficient execution trace that can be manipulated through a powerful *timeline*, which provides common facilities like, for instance trace visualization, and step-by-step execution (incl. breakpoint, step forward and step backward). Finally, *MoCCML* is used to provide an alternative to the implementation with an explicit and formal concurrency model for activity diagrams supported by a solver and analysis tools. We evaluate our implementation with regard to the evaluation criteria provided in the case description and give evidence of the correctness, understandability, conciseness and performance of our solution.

1 Introduction

Executability of models opens many possibilities in terms of early dynamic verification and validation (V&V) of models, such as debugging [1, 5], model checking [3] and runtime verification [9]. In recent years, a lot of efforts have been made to provide facilities to design executable Domain-Specific Modeling Languages (xDSMLs) [4, 10, 12]. To establish an overview of the state of the art in terms of tools and methods to design and implement xDSMLs, the Transformation Tool Contest (TTC) 2015[1] has proposed a dedicated case about Model Execution [11]. This case describes a part of the execution semantics for the UML Activity Diagram language in the form of an operational semantics.

In this paper, we present a solution to the most complete variant of this case (*i.e.,* variant 3) using the GEMOC Studio[2]. The variant 3 of the Model Execution case considers the complete Activity Diagram metamodel provided. It includes various kinds of nodes (initial node, final node, fork node, join node,

[1] Cf. http://www.transformation-tool-contest.eu
[2] Cf. http://gemoc.org/studio

decision node, merge node), opaque actions, Boolean and integer variables (either local or as input), and various Boolean and integer expression types.

In the rest of this paper we first present in Section 2 an overview of the solution using the GEMOC language workbench to design and implement the UML Activity Diagram language as defined in the variant 3 of the Model Execution case, as well as the resulting environment in the GEMOC modeling workbench. Then, in Section 3, we evaluate our implementation with regard to the evaluation criteria provided in the case description and provide evidence of the correctness, understandability, conciseness and performance of our solution. Finally, Section 4 concludes and gives some perspectives for the GEMOC Studio. Annex A gives a detailed description of the solution.

2 Solution Overview

Our solution uses the GEMOC Studio, an Eclipse package atop the *Eclipse Modeling Framework* (EMF)[3], which includes both a language workbench to design and implement tool-supported xDSMLs, and a modeling workbench where the xDSMLs are automatically deployed to allow system designers to edit, execute, simulate, and animate their models. As a result, our solution not only provides a model interpreter conforming to the proposed operational semantics of the UML Activity Diagram language, but also provides a graphical model animator, an advanced trace manager, as well as an alternative version that offers an explicit and formal model of computation supporting concurrency. All resources are available from `http://gemoc.org/ttc15`.

For designing and implementing the various concerns of an xDSML, the language workbench put together the following tools seamlessly integrated into EMF:

- *Kermeta*, which offers specific annotations for Xtend[4] to support the modular implementation of an operational semantics (both runtime concepts and steps of computation) and its weaving into an EMF-based metamodel (*i.e.,* an Ecore model).
- *Melange*[7], to build the overall language runtime seamlessly integrated into EMF and to ensure interoperability between the legacy metamodel without the operational semantics, and the metamodel extended with the operational semantics.
- *Sirius Animator*, an extension of the model editor designer Sirius[5] to create graphical animators for xDSMLs.
- *MoCCML*, a tool-supported meta-language to specify a Model of Concurrency and Communication (MoCC) and its mapping to a specific metamodel and associated operational semantics of a xDSML.

The language workbench also includes a generative approach, which provides a rich and efficient domain-specific trace metamodel for any xDSMLs (for more details, we refer the reader to [2]).

Once an xDSML is implemented with the aforementioned tools of the language workbench, the xDSML is automatically deployed into the modeling workbench, which provides an advanced environment integrated into the Eclipse debugger for model execution. In particular, the modeling workbench provides the following tools:

- A Java-based *execution engine* (parameterized with the specification of the operational semantics), possibly coupled with *TimeSquare*[6] (parameterized with the MoCC), to support the concurrent execution and analysis of any conforming models.

[3] Cf. `https://www.eclipse.org/modeling/emf`
[4] Cf. `https://eclipse.org/xtend`
[5] Cf. `https://eclipse.org/sirius`
[6] Cf. `http://timesquare.inria.fr`

- A *model animator* parameterized by the graphical representation defined with Sirius Animator to animate executable models.
- A generic *trace manager*, which allows a system designer to visualize, save, replay, and explore different execution traces of their models, as well as navigating step-by-step in a given execution trace (incl. breakpoint, step forward and step backward).
- A generic *event manager*, which provides a user interface for injecting external stimuli in the form of events during the simulation (*e.g.,* to simulate the environment).

The implementation of the UML Activity Diagram language (see details in Annex A) is automatically deployed in the GEMOC modeling workbench (see Fig. 1).

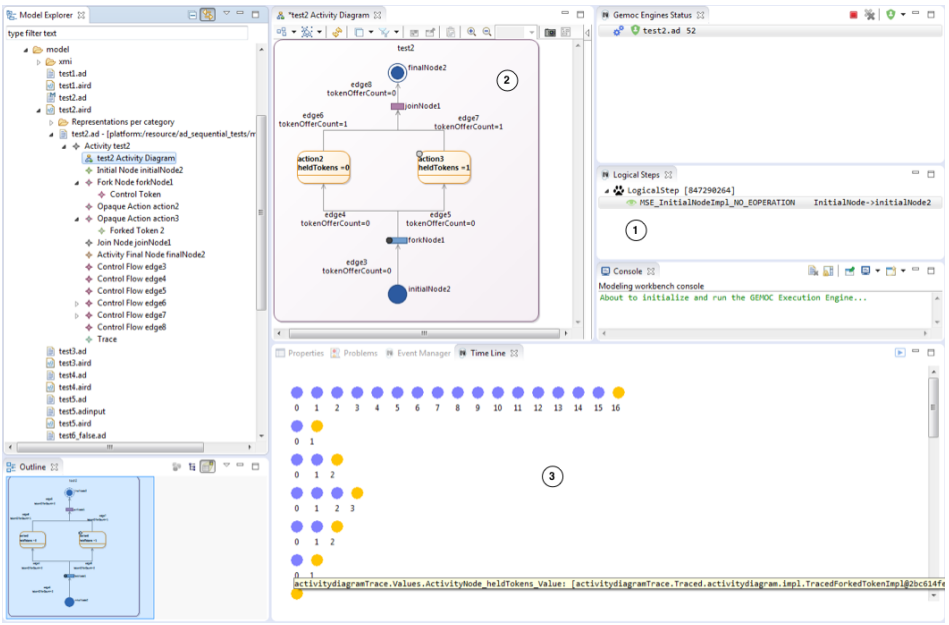


Figure 1: The GEMOC Modeling Workbench for the TTC'15 Activity Diagram Language

The modeling workbench offers a powerful environment to system engineers for controlling the execution of their models with a debugger-like control panel[7] (①), visualizing the execution of their models thanks to the graphical animator (②), and analyzing and exploring several execution traces with a graphical timeline that supports step forward and step backward (③). Finally, the modeling workbench offers several extension points that can be used to plug additional front-end or back-end, such as a timing diagram included into the modeling workbench.

3 Evaluation of the Solution

We now evaluate our solution by using the evaluation criteria proposed in the case description [11]. Each criteria is evaluated on three different versions of our solution, all implemented within the GEMOC Studio:

[7] Note that when using MoCCML, the concurrent computational steps are indicated in the control panel (the computational steps that will be executed concurrently during a given execution step). If the MoCC is non-deterministic, the control panel proposes the different permitted execution steps, one of which can be either selected manually by the system engineer, or proposed automatically by one of the built-in heuristic.

- *executionOnly*: interpreter defined with Kermeta only (incl. Section A.1);
- *withAnimationAndTrace*: execution within the GEMOC modeling workbench, with support of animation and trace management (incl. Sections A.1, A.2, A.3 and A.4);
- *withConcurrency*: execution within the GEMOC modeling workbench, with support of animation, trace management and concurrency (incl. Sections A.1, A.2, A.3, A.4 and A.5).

3.1 Correctness

The correctness of our solution is based on the test suites provided by the case. All the three versions of our solutions provide correct results.

3.2 Understandability and Conciseness

Kermeta is used to design and implement the operational semantics. Based on Xtend, Kermeta provides a powerful Java-like imperative and statically typed meta-language. This last follows an object-oriented paradigm which makes it directly aligned with the object-oriented Ecore metamodel provided by the case.

The implementation of the operational semantics follows the well-known *Interpreter* design pattern which supports a modular design of the operational semantics with regard to the initial metamodel which is reused as is and not affected. There is no translation into a third formalism, and this approach easily supports the definition of different variants of the semantics (*e.g.,* interpreter and compiler, different semantic variation points, etc.). Finally, the use of the open-class and static introduction mechanisms makes the design of the operational semantics even simpler than the interpreter pattern, avoiding to duplicate the initial structure into the interpreter. The operations of the operational semantics are directly weaved into the suitable classes of the initial metamodel.

The entire implementation of the operational semantics of the variant 3 of the Model Execution case comprises 441 LOC (version *executionOnly*). This includes the entire implementation of the interpreter sufficient for the execution of any conforming models. The other technologies Melange, Sirius Animator, and MoCCML are optional, and can be used only to provide the additional features such as model interoperability, trace management, model animation, and formal concurrency specification and analysis. Note also that the analysis tools that provide the modeling workbench are not only useful for the system engineer to analyse the models, but also for the language designer to analyse the language semantics implementation.

3.3 Performance

	executionOnly	withAnimationAndTrace	withConcurrency
Test perf 1	0.29	0.87	226183
Test perf 2	0.33	0.78	$\perp$
Test perf 3_1	0.37	1.01	$\perp$
Test perf 3_2	0.13	0.19	5219

Table 1: Execution time (in ms) of the performance tests (using *System.nanoTime()*)

Table 1 shows the execution time (without load and save times) of the models provided for the performance evaluation (*Test perf 1*, *Test perf 2*, *Test perf 3_1* and *Test perf 3_2*) in . The execution time is provided for all the models, according to the three versions of our solution. Performance evaluation has been performed using an Ubuntu VirtualBox image with Java 8 and the last Gemoc Studio. This virtual machine ran on top of a HP EliteBook 820 computer with an Intel Core i7 processor and 16GB of memory. Note that in

table 1, two $\perp$ appears due to the fact that there are more than 2^{100} possible inter-leavings in *Test perf 2* and *Test perf 3_1*. The goal of the concurrent version is to show and make explicit such interleavings but in these cases, it is both non valuable and impossible.

4 Conclusion and Perspective

We present in this paper our solution using the GEMOC Studio to the most complete *variant 3* of the TTC'15 Model Execution case. The solution provides not only an EMF-based interpreter for UML activity diagrams, but also comes with a well-integrated model debugging environment based on Eclipse, including advanced features for graphical model animation and execution trace management. We also propose an enhanced version of our solution which integrate into the operational semantics a formal and explicit model of concurrency supported by analysis tools. The GEMOC Studio integrates different technologies to implement the various concerns of the executability (runtime concepts, steps of computation, animator, concurrency). We evaluate our solution regarding both the benchmark provided by the case, and the criteria proposed in the case description. In particular, we give evidence for the correctness, the understandability and conciseness, and the performance of our solution.

The GEMOC Studio is a play ground for research activities related to *Software Language Engineering*, including model executability. Various studies are currently investigated on related topics, including the integration with continuous time, formal analysis, optimizing compilers, semantic variability and adaptation, and application to other domains (*e.g.,* enterprise architecture and scientific modeling).

References

[1] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray & Benoit Baudry (2015): *Supporting Efficient and Advanced Omniscient Debugging for xDSMLs*. In: Proc. of SLE'15.

[2] Erwan Bousse, Tanja Mayerhofer, Benoit Combemale & Benoit Baudry (2015): *A Generative Approach to Define Rich Domain-Specific Trace Metamodels*. In: ECMFA, LNCS, Springer.

[3] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche & Xavier Thirioux (2009): *Essay on Semantics Definition in MDE, An Instrumented Approach for Model Verification*. Journal of Software 4(9).

[4] Benoit Combemale, Xavier Crégut & Marc Pantel (2012): *A Design Pattern to Build Executable DSMLs and Associated V&V Tools*. In: APSEC, IEEE.

[5] Jonathan Corley, Brian P. Eddy & Jeff Gray (2014): *Towards Efficient and Scalabale Omniscient Debugging for Model Transformations*. In: DSM Workshop, ACM.

[6] Julien Deantoni, Papa Issa Diallo, Ciprian Teodorov, Joël Champeau & Benoit Combemale (2015): *Towards a Meta-Language for the Concurrency Concern in DSLs*. In: DATE.

[7] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais & Jean-Marc Jézéquel (2015): *Melange: A Meta-language for Modular and Reusable Development of DSLs*. In: Proc. of SLE'15.

[8] Clément Guy, Benoît Combemale, Steven Derrien, Jim RH Steel & Jean-Marc Jézéquel (2012): *On model subtyping*. In: ECMFA, LNCS, Springer.

[9] Martin Leucker & Christian Schallhart (2009): *A brief account of runtime verification*. The Journal of Logic and Algebraic Programming 78(5).

[10] Tanja Mayerhofer, Philip Langer, Manuel Wimmer & Gerti Kappel (2013): *xMOF: Executable DSMLs based on fUML*. In: SLE, LNCS 8225, Springer.

[11] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. In: TTC, CEUR.

[12] Jérémie Tatibouët, Arnaud Cuccuru, Sébastien Gérard & François Terrier (2014): *Formalizing Execution Semantics of UML Profiles with fUML Models*. In: MODELS, LNCS 8767, Springer, pp. 133–148.

A Description of the solution

In this annex we describe the design and implementation of the UML Activity Diagram language as defined in the variant 3 of the TTC'15 Model Execution case [11].

A.1 Operational Semantics

Kermeta is used to implement the operational semantics. Kermeta complements Xtend to support the definition of both the runtime concepts and the steps of computation in a separate file rather than in the initial metamodel, and to statically weave them in the initial metamodel. Kermeta provides static typing to safely define the operational semantics, and a compilation scheme of the operational semantics which results in a Java-based runtime seamlessly integrated to the Java code generated by EMF from the initial metamodel.

The runtime concepts can be additional classes that will be merged with the initial metamodel, or new structural features (attributes or references) either in the existing classes of the initial metamodel or in the newly added classes. When new structural features have to be added to a class existing in the initial metamodel, the annotation *@Aspect* is used to re-open the class.

Listing 1 shows an excerpt of the modular definition of the runtime concepts. *Token* and *ForkToken* are new concepts, while the content of *ActivityNodeAspect*, a collection of *Token*, will be merged into the concept *ActivityNode* from the abstract syntax (cf. annotation *@Aspect*). All the runtime concepts of the case have been defined similarly.

```
1  @Aspect(className=ActivityNode)
2  class ActivityNodeAspect {
3      List<Token> heldTokens = new ArrayList<Token>
4  }
5
6  abstract class Token {
7      public ActivityNode holder
8  }
9
10 class ForkedToken extends Token {
11     public Token baseToken ;
12     public Integer remainingOffersCount;
13 }
14
15 [...]
```

Listing 1: Modular definition of the runtime concepts with Kermeta

The steps of computation are defined in terms of operations weaved into the suitable classes, either from the initial metamodel or from the newly added classes of the runtime concepts. Similarly to the structural features of the runtime concepts, when an operation has to be added to a class existing in the initial metamodel, the annotation *@Aspect* is used to re-open the class.

Listing 2 shows an excerpt of the definition of the steps of computation (i.e., the interpreter), which manipulates the runtime concepts previously defined. The implementation follows the *Interpreter* design pattern [8], defining one operation *execute* per concept of the abstract syntax to be interpreted. Each method is modularly defined in an aspect, and then weaved into the suitable class of the abstract syntax. Listing 2

[8] Cf. http://en.wikipedia.org/wiki/Interpreter_pattern

shows the overall execution of an *Activity*. All the steps of computation of the most complete variant of the case have been defined similarly.

```
1  @Aspect(className=Activity)
2  class ActivityAspect {
3
4      def void execute(Context c) {
5          _self.locals.forEach[v|v.init(c)]
6          _self.nodes.filter[node|node instanceof InitialNode].get(0).execute(c)
7
8          var list =  _self.nodes.filter[node|node.hasOffers]
9          while (list!=null && list.size>0 ){
10             list.get(0).execute(c)
11             list =  _self.nodes.filter[node|node.hasOffers]
12         }
13     }
14 }
15
16 [...]
```

Listing 2: Modular definition of the steps of computation with Kermeta

The definition of the runtime concepts and the steps of computation in a separate file offers a modular mechanism to implement the operational semantics. In addition to support the separation of concerns (abstract syntax and operational semantics), this is also a way to support different implementations of the operational semantics for the same abstract syntax (*e.g.,* in case of semantic variation points).

A.2 Language Assembling

Once the operational semantics is defined with Kermeta, *Melange*[9] can be used from the language workbench for assembling the initial metamodel and the chosen operational semantics into an xDSML.

```
1  language UMLActivityDiagram {
2      syntax "platform:/resource/.../activitydiagram.ecore"
3      with org.gemoc.ad.sequential.dynamic.*
4      exactType UMLActivityDiagramMT
5  }
```

Listing 3: Assembling an xDSML with Melange

As a result, Melange provides the xDSML as well as a structural interface (*aka.* model type [8]) in the form of a new metamodel that can be used to define additional tooling such as model transformations (*e.g.,* trace manager and execution engine) or animators, which use the operation semantics (runtime concepts or steps of computation). In addition to provide the assembling of the expected xDSML as well as the interoperability between the initial metamodel and the metamodel with the operational semantics, Melange also provides other features not required in this solution such as language inheritance and model transformation reuse.

A.3 Trace Management

Based on the resulting xDSML, the language workbench includes a generative approach that automatically provides a rich and efficient domain-specific trace metamodel. Instead of relying on complete snapshots

[9] `http://melange-lang.org`

of the executed model to construct a trace, this metamodel precisely captures what the execution state of a model conforming to the xDSML is through an efficient object-oriented structure based on the runtime concepts of the xDSML. In addition, the structure provides rich navigation facilities to browse a trace according to various dimensions (*e.g.* the value of a field or the occurrences of an event). For more details we refer the reader to [2].

A.4 Animation Facilities

Optionally, *Sirius Animator* can be used to complement the xDSML with a graphical model animator. *Sirius Animator* allows to either extend the graphical representation of an existing model editor defined with Sirius, or to define a separate graphical representation, based on the runtime concepts. This graphical representation is then used to visualize the state of a model during its execution.

In our solution, we defined a new graphical representation (called *viewpoint specification* in Sirius) on top of the provided metamodel for UML activity diagrams, augmented with the runtime concepts to be visualized at runtime.

A.5 Explicit and Formal Concurrency Model

Because concurrency is a more and more important concept, one can use *MoCCML* to specify the MoCC. A MoCC specifies the possibly timed causalities and synchronizations among the steps of computation in a formal way. Based on *MoCCML*, non-determinism and parallelism are clearly and formally identified in the operational semantics and can be varied or refined [6]. Analysis tools are also provided in the GEMOC Studio to analyze the MoCC.

Listing 4 shows an excerpt of the MoCC specification. Lines 1 and 2 define an event in the context of an *ActivityNode* (*i.e.*, for all its instances). For each occurrence of this event the *execute* function is called. All these events are constrained by some relations. For instance, in the classical case, the execution of a node is done after its predecessor has been executed (see the *Precedes* relation line 6). In the context of *Activity* appears a kind of loop since the activity can not start if not stop (line 11) and its *start* actually executes the initial node of the activity (line 15), *i.e.*, the starting point of the causality chain written in Line 6. From such a specification, and for a specific model, a symbolic event structure is automatically derived.

```
1  context ActivityNode
2    def : executeIt : Event = self.execute()
3    inv waitControlToExecute:
4          (not self.oclIsKindOf(MergeNode)) implies
5          Relation Precedes(self.incoming.source.executeIt, self.executeIt))
6
7  context Activity
8    def : start : Event = self.initialize()
9    def : finish : Event = self.finish()
10   inv NonReentrant:
11       Relation Alternates(self.start, self.finish)
12
13 context InitialNode
14   inv startedWhenActivityStart:
15       Relation Precedes(self.activity.start, self.executeIt  )
```

Listing 4: Excerpt of the explicit and formal model of concurrency for activity diagrams

fUML Activity Diagrams with RAG-controlled Rewriting
– A *RACR* Solution of *The TTC 2015 Model Execution Case* –

Christoff Bürger
Department of Computer Science, Faculty of Engineering, LTH, Lund University
Lund, Sweden
christoff.burger@cs.lth.se

This paper summarises a *RACR* solution of *The TTC 2015 Model Execution Case*. *RACR* is a metacompiler library for *Scheme*. Its most distinguished feature is the seamless combination of reference attribute grammars and graph rewriting combined with incremental evaluation semantics. The presented solution sketches how these integrated analyses and rewriting facilities are used to transform *fUML Activity Diagrams* to executable Petri nets. Of particular interest are (1) the exploitation of reference attribute grammar analyses for Petri net generation and (2) the efficient execution of generated nets based on the incremental evaluation semantics of *RACR*.

1 Prerequisites and Contents

The following document describes a *RACR*-based [1] solution of the *Model Execution Case* [6] of the *8th Transformation Tool Contest* which was part of the *Software Technologies: Applications and Foundations* (*STAF*) conference 2015. It assumes readers are familiar with the contest task (cf. [6]); no further previous knowledge is required, although a basic understanding of reference attribute grammars [5] and familiarity with the *Scheme* programming language [3] will be helpful. The presented solution is part of *RACR's* source code repository at `https://github.com/christoff-buerger/racr`; a deployed *SHARE* [8] demonstrator is provided at `https://is.ieis.tue.nl/staff/pvgorp/share/`.

The structure of this document is as follows: Section 2 gives a short overview of the solution. It first presents the implemented analyses in Section 2.1, concluding in a sketch of the intended abstract syntax graphs used to execute *fUML Activity Diagrams* [4]. Afterwards, Sections 2.2 sketches the implementation of execution semantics by means of rewrites reusing the implemented analyses. An evaluation follows in Section 3. The actual source code is investigated in the appendix; readers are highly encouraged to closely follow it and consult *RACR's* reference manual [1] as required.

2 Solution Overview

The activity diagram interpreter presented in the following is realised in the form of two language processors. The first analyses the actual activity diagram and its inputs and translates them to a Petri net [7]. The second executes generated Petri nets – it is a Petri net interpreter.

2.1 RAG-based Analyses: From Activity Diagrams to Petri Nets

Figure 1 sketches the abstract syntax graph of an exemplary activity diagram. Our interpreter is implemented in terms of such graphs; they represent the original input diagram, its current execution state and analysis results, including static *and* dynamic analyses like diagram well-formedness or whether

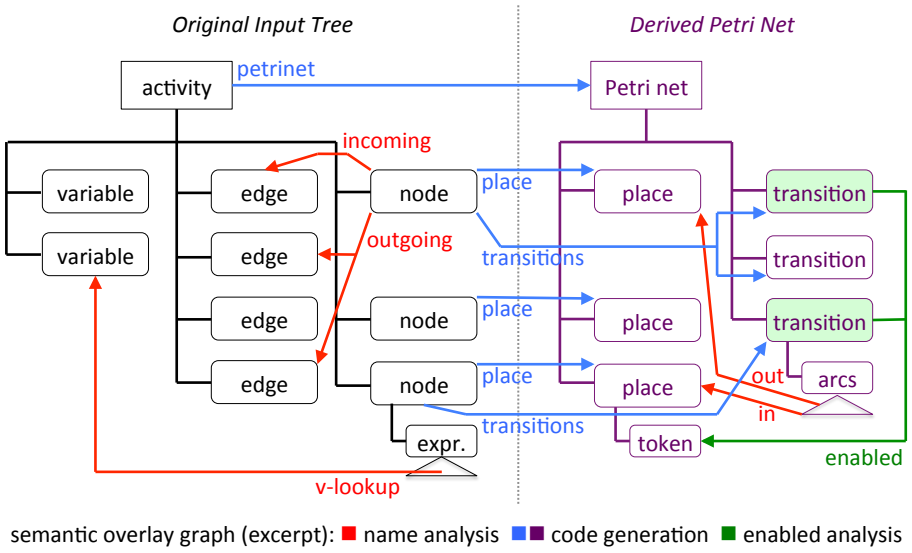


Figure 1: Example abstract syntax graph of the activity diagram interpreter ©Christoff Bürger

activities are ready for execution. The graph consists of two abstract syntax trees (black and purple nodes and edges). The black one on the left encodes the actual activity diagram; it is the original input of the interpreter[1] and constructed by a hand written recursive-decent parser (the parser is straightforward and not investigated in the following). The purple right tree encodes the Petri net used to execute the diagram; code generation derives it from the original input tree (blue edges). Name analysis extends both trees to the actual diagram each represents. In case of the original input tree, it resolves the symbolic names of activity edges to the target and source activity nodes they refer to, such that each node knows all its in- and outgoing edges (*Original Input Tree*, red edges). In case of the derived Petri net, name analysis resolves the symbolic names of the arcs of transitions to the actual places they refer to (*Derived Petri Net*, red edges). Enabled analysis finally associates transitions with the tokens they consume if fired or no token if disabled (green edges). It just is a special kind of name analysis, searching for consumable tokens and returning the tokens consumed if a transition is enabled and *false* if it is disabled.

Everything in Figure 1, except the original input tree encoding the activity diagram, is derived by the interpreter. The interpreter computes an semantic overlay graph that extends its input tree to a graph well-suited for digram execution. The required analyses are implemented using *RACR's* reference attribute grammar facilities, each by a set of attributes. Figure 1 shows only an excerpt of the actually implemented analyses and the resulting abstract syntax graph. The names of reference attributes are labeled next to the edges they induce; for example, the `v-lookup` attribute finds the variables the assignee and operands of expressions refer to. Not shown are non-reference attributes like type and well-formedness analysis, parts of the code generation, for example for expressions of executable nodes, and minor query-support analyses like lookup of activity nodes and edges by name.

Important for the development-effort-benchmarks in Section 3 is, that analyses can be interdependent, fostering reuse and modularisation. *RACR's* demand-driven evaluation strategy automatically deduces correct evaluation orders, easing the implementation of complex or mutually dependent analyses. For example, code generation can reuse name analysis, execution reuses code generation and the runtime

[1] Input for the interpreter are textual diagram specifications as given by the tool contest [6]. Parsing such specifications yields abstract syntax trees like the one labeled *Original Input Tree* in Figure 1; they satisfy the scheme in Appendix A.1.

lookup of tokens (i.e., the name and enabled analyses of Petri nets) reuses the places code generation generated. From the perspective of a user – whether interpreter or Petri net developer – a common interface for querying analyse results is provided: abstract syntax graphs as shown in Figure 1. Moreover, analyses are automatically memoized; deduced abstract syntax tree parts are only re-evaluated if required.

2.2 Rewriting-based Transformations: Incremental Execution of Petri Nets

Given abstract syntax graphs as in Figure 1 and all the deduced analyse results they encode, the specification of execution semantics boils down to simple transformations manipulating their tokens. After all, convenient means to find enabled activity nodes considering the state of execution are already provided (enabled analysis reasons about the current marking of the generated Petri net). Execution therefore can be realised by a simple loop that reuses the enabled analysis to find an enabled transition and deletes its consumed and adds its produced tokens using *RACR's* primitive rewrite functions `rewrite-delete` and `rewrite-add` [1]; if no transition is enabled, execution terminates.

Important for the performance-benchmarks in Section 3 are the automatic incremental evaluation semantics of *RACR*. When an abstract syntax graph information is queried throughout attribute evaluation, *RACR* maintains a dependency to remember that the value of the attribute depends on the queried information. If an abstract syntax graph information changes, *RACR* invalidates all attributes transitively depending on it. The enabled analysis of the Petri net language is no exception since it is implemented using attributes. It depends on tokens that would be consumed or are missing, including the special case of tokens encoding variable values. All these dependencies are automatically tracked by *RACR*, such that the enabled status of incoming arcs is only re-evaluated if it could be changed by a fired transition, otherwise the cached is used. Likewise, the enabled status of transitions is only re-evaluated if the enabled status of any of their incoming arcs was invalidated. For example, if any of the two enabled transitions of Figure 1 (highlighted green) is fired, the enabled attributes of both are invalidated since each depends on the token deleted according to firing semantics. Similarly, when a new value is assigned to a variable via `rewrite-terminal` (cf. Appendix A.3.2), the enabled status of transitions depending on its value is re-evaluated, if either, they were enabled or, although all tokens they consume are provided, still were disabled. Without special implementation efforts, *RACR* optimises the implemented execution semantics.

The activity diagrams of the tool contest result in very simple and restricted Petri nets with just a single token type (except tokens encoding variable values; cf. Appendix A.3.2) and at most one token per place. The developed Petri net language is much more expressive however, supporting coloured, weighted Petri nets with arbitrary input arc conditions and output computations; it was developed before the tool contest for more general applications. In case of the tool contest, the restricted type and number of tokens, and therefore simple enabled decisions, preclude major performance benefits from *incremental* enabled analysis. If there are only few tokens and conditions to check, caching the results of such checks does not pay-off as much as in more complex cases. Of course, the execution semantics could be optimised for such less expressive nets. For example, the transitions of the Petri nets generated for most activity diagrams never compete for tokens (this holds for example for all test cases given by the tool contest). In this case, all enabled transitions can be fired in one pass (enabled pass); only thereafter, for the next iteration of the execution loop, enabled analysis has to be repeated[2]. In general however, Petri net transitions can compete for tokens. For example, in Figure 1 the two enabled transitions highlighted green compete for the same token; their enabled attributes point to the same token to consume if fired. To fire one of the two enabled transitions disables the other one.

[2]Enabled passes still sequentially execute parallel fork branches; they perform no multi threaded execution. They execute one activity *of each* active branch in each iteration step instead of *a single* activity of some active branch.

Source Code File	Solution Part (language task)	LOC	
Activity diagram language (507):		*499*	
`analyses.scm`: *255*	AST specification	18	4%
	ASG accessors (constructors, child & attribute accessors)	65	13%
	Name analysis	32	6%
	Type analysis	23	5%
	Well-formedness	32	6%
	Petri net generation	90	18%
`parser.scm`: *219*	Parsing	214	43%
`user-interface.scm`: *33*	Initialisation & execution	25	5%
Petri net language (255):		*200*	
`analyses.scm`: *102*	AST specification	9	5%
	ASG accessors (constructors, child & attribute accessors)	32	16%
	Name analysis	13	7%
	Well-formedness	10	5%
	Enabled analysis	29	15%
`execution.scm`: *43*	Running and firing semantics	31	16%
`user-interface.scm`: *80*	Initialisation & Petri net syntax	33	17%
	Read-eval-print-loop interpreter	19	10%
	Testing nets (marking & enabled status)	24	12%

Figure 2: Solution size (lines of code, LOC)

3 Evaluation

Development-effort-benchmarks Figure 2 summarises the size of the implementation in terms of lines of code, excluding empty lines and pure comments. The difference between the size of the solution parts and their source code files is due to boilerplate code for library imports and exports not being accountable to any certain task. Also, the abstract syntax graph accessors are boilerplate code that could be generated and should not be counted. They are mostly one liners to introduce convenient functions for node constructions and child and attribute querying. For example, in the listings of Appendix A we will write `(->target n)` to query the target of an activity edge. *RACR* provides generic query functions however, such that the query would be `(ast-child 'target n)` (cf. reference manual [1, Chapter: *Abstract Syntax Trees*]). To this end we specify the abstract syntax graph access function `(define (->target n) (ast-child 'target n))` which is obviously boilerplate. Finally, note that the implementation of user interface functionality makes up huge parts of the implementation (in case of the activity diagram language 48%; for the Petri net language 39%). To develop language user interfaces is not subject of *RACR* however; input parsing and abstract syntax tree instantiation therefore should also be excluded.

Performance-benchmarks Figure 3 presents the results of benchmarking the performance test cases given by the tool contest. The benchmarks have been executed on a *MacBook Air (Mid 2011)* with a 1.7GHz *Intel Core i5* CPU, 4GB 1333MHz DDR3 RAM and *Mac OS 10.10.3*. As *Scheme* system *Larceny 0.98 (General Ripper)*[3] was used. Times were measured using the `time` command of *UNIX* without warming up the *Larceny* virtual machine just by execution from *Bash*. Each test case was performed with increasing numbers of translation tasks, such that the actual times spend for parsing, well-formedness checks, Petri net generation and their actual execution can be investigated. For example,

[3] `http://www.larcenists.org` and `https://github.com/larcenists/larceny`

Tasks Performed *(later tasks include previous ones)*	Test Cases (`testperformance_variant`) 1	2	3_1	3_2	Time Spend *(lowest / highest / average)*
Activity diagram parsing	831 / 831	871 / 871	875 / 875	718 / 718	41% / 86% / 50%
Activity diagram well-formedness	926 / 95	1017 / 146	1079 / 204	739 / 21	3% / 11% / 7%
Petri net generation	1042 / 116	1061 / 44	1196 / 117	741 / 2	0% / 6% / 4%
Petri net well-formedness	1220 / 178	1230 / 169	1466 / 270	746 / 5	1% / 14% / 10%
Petri net execution	2026 / 806	1776 / 546	1912 / 446	831 / 85	10% / 40% / 29%
Petri net execution (enabled passes)	2618 / 1398	1344 / 114	1572 / 106	836 / 90	7% / 53% / 27%

Figure 3: Time measurements (times in ms: total / task-only)

`testperformance_variant2.ad` spend 169ms on checking the well-formedness of its Petri net making a total of 1230ms with Petri net execution excluded. Of this 1230ms, 44ms where spend to generate the Petri net, 146ms to check well-formedness of the activity diagram and 871ms to parse the test file and construct an abstract syntax tree. The activity diagram parsing time includes loading the *Larceny* virtual machine, *RACR* and the activity diagram and Petri net languages. The percentage of time spend for a certain task is w.r.t. a test case's total execution time. It is only shown for the test cases with the lowest and highest percentage spend for each task (highlighted by colouring the time of the respective test case). The average percentage is the sum of all test cases to perform a certain task divided by the sum of their total execution times. Again, readers should exclude parsing times when judging *RACR*.

The last row in Figure 3 presents the execution times of a variant with enabled passes. The implementation of this variant requires three more lines of code. As described in Section 2.2, it just fires all enabled transitions each execution loop iteration instead of a single. Of course, if there are no forks the enabled pass variant wastes time to filter all enabled transitions. If there are parallel branches however, enabled passes improve execution performance a lot. Thanks to the incremental enabled analysis, the execution without enabled passes nevertheless performs surprisingly well.

References

[1] Christoff Bürger (2012): *RACR: A Scheme Library for Reference Attribute Grammar Controlled Rewriting*. Technical Report TUD-Fl12-09, Lehrstuhl Softwaretechnologie, Technische Universität Dresden. Updated version distributed with *RACR* at `https://github.com/christoff-buerger/racr`.

[2] Christoff Bürger, Sven Karol, Christian Wende & Uwe Aßmann (2011): *Reference Attribute Grammars for Metamodel Semantics*. In Brian Malloy, Steffen Staab & Mark van den Brand, editors: *Software Language Engineering: Third International Conference*, *Lecture Notes in Computer Science* 6563, Springer, pp. 22–41.

[3] R. Kent Dybvig (2009): *The Scheme Programming Language*, 4 edition. MIT Press.

[4] Object Management Group (2013): *Semantics of a Foundational Subset for Executable UML Models (fUML)*. Technical Report, Object Management Group. Version 1.1.

[5] Görel Hedin (2000): *Reference Attributed Grammars*. *Informatica (Slovenia)* 24(3), pp. 301–317.

[6] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. Technical Report, Business Informatics Group, Vienna University of Technology.

[7] Wolfgang Reisig (2013): *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer. English translation of *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*.

[8] Pieter Van Gorp & Steffen Mazanek (2011): *SHARE: a web portal for creating and sharing executable research papers*. *Procedia Computer Science* 4, pp. 589–597.

A Activity Diagram Language

The abstract syntax graph of the activity diagram language corresponds to the metamodel given in the task description [6, Figure 1].

A.1 Abstract Syntax Tree Scheme

The metaclasses and their composite relations determine the solution's abstract syntax tree scheme. For example, the following excerpt of the abstract syntax tree scheme specifies the metaconcepts `Activity`, `Variable`, `ActivityEdge` and `ControlFlow`:

```
1 (ast-rule 'Activity->name-Variable*-ActivityNode*-ActivityEdge*)
2 (ast-rule 'Variable->name-type-initial)
3 (ast-rule 'ActivityEdge->name-source-target)
4 (ast-rule 'ControlFlow:ActivityEdge->guard)
```

Note, that names starting lowercase on right-hands (following the `->`) denote terminal children – i.e., ordinary properties – whereas names starting uppercase denote non-terminals – i.e., composite relations. Unbounded composites (Kleene closures/unbounded repetitions) are denoted by a `*` following the respective non-terminal. Analogous to the task description's metamodel, `ControlFlow` inherits from `ActivityEdge` denoted by `:ActivityEdge`. By doing so control-flow edges not only inherit the name, source and target properties of activity edges, but also their attributes and therefore semantic analyses (in terms of metamodelling the attributes of a reference attribute grammar are derived properties and methods [2]).

A.2 Name, Type and Well-formedness Analyses

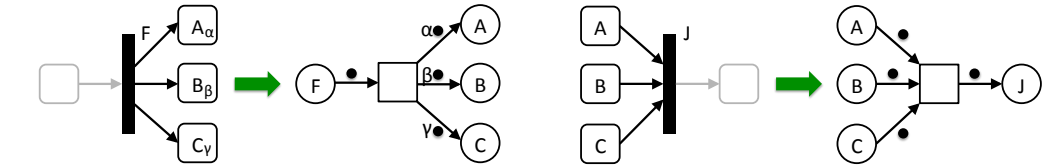
The main purpose of the attribute-based semantic analyses of the activity diagram language is, besides the actual generation of Petri nets, the provision of information convenient for such code generation. This comprises the construction of a graph structure encoding all information required for code generation (name analysis) and checks that ensure diagrams are also valid such that the generated Petri nets do not misbehave (type and well-formedness analyses).

As a name analysis example consider the association of activity edges with nodes (`incoming` and `outgoing` attribute). To do so, hash maps from node names to their respective incoming and outgoing edges are constructed. Given these maps, each node can just lookup its own name to determine its edges:

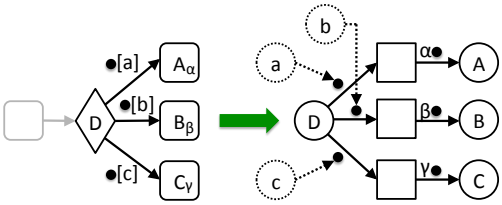
```
1 (ag-rule
2  incoming ; List of incoming edges of a node.
3  (Activity        (lambda (n) (make-connection-table ->target (=edges n))))
4  (ActivityNode    (lambda (n) (hashtable-ref (=incoming (<- n)) (->name n) (list)))))
```

To query an attribute for its value we just write `(=attribute-name n)`; to query an abstract syntax tree child or parent we just write `(->child/terminal-name n)` and `(<- n)` respectively. In all three cases, `n` is the context node, i.e., the node the attribute is associated with/the node which has the child/the node whose parent is queried respectively. The lookup of incoming edges at an activity node `n` works as follows (Line 4): get the diagram's hash table via `(=incoming (<- n))` and query it with the activity node's name; if it has no entry, return the empty list (the last `(list)` on Line 4). To construct the actual table (Line 3), we just call a support function which given an access function `->` and list of abstract syntax tree nodes queries all its elements and adds them to a newly constructed hash table according to their `->` values[4]. In our case the arguments are just all edges of the diagram (supported by the `=edges` attribute) and the

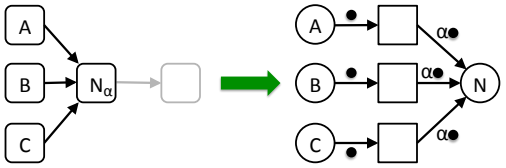
[4]The implementation is straightforward and based on `hashtable-update!` provided by *Scheme* [3].



(a) fork and join transformation



(b) decision transformation



(c) initial, final, merge & executable node transformation

Figure 4: Activity Diagram to Petri net transformation rules ©Christoff Bürger

target query function `->target`. Likewise, the name analysis provides attributes to lookup variables, nodes and edges (`v-lookup`, `n-lookup` and `e-lookup` attribute), the source and target of edges (`source` and `target` attribute) and the initial node (`initial` attribute).

Given the name analysis, type analysis is easy to implement (`well-typed?` attribute). Consider for example unary expressions, which, according to the metamodel, must be negations:

```
1 (UnaryExpression
2  (lambda (n)
3    (define ass (=v-lookup n (->assignee n)))
4    (define op (=v-lookup n (->operand1 n)))
5    (and ass op (eq? (->type op) Boolean) (eq? (->type ass) Boolean)))))
```

First we lookup the variable to write the result to and the negated operand (Lines 3 & 4). Afterwards we ensure both exist and are indeed of type Boolean (Line 5).

Based on type and name analyses we can check well-formedness. As an example consider decisions and executable nodes:

```
1 (DecisionNode     (lambda (n) (and (in n = 1) (out n >= 1) (guarded n #t))))
2 (ExecutableNode   (lambda (n) (and (in n = 1) (out n = 1) (guarded n #f)
3                                    (for-all =well-typed? (=expressions n)))))
```

In both cases we use three support functions. The `in` and `out` functions ensure the node has a certain number of incoming and outgoing edges. The `guarded` function asserts, depending on its boolean argument, whether outgoing edges must be control-flows (in case of *true* they must be, otherwise not). Decisions must have a unique incoming edge, at least one outgoing edge and their outgoing edges must be control-flows (Line 1). Executable nodes must have a unique incoming and outgoing edge which is not a control-flow (Line 2). Furthermore, all their expressions must be type correct (Line 3).

A.3 Code Generation

A.3.1 Places, Transitions & Arcs

Figure 4 summarises the code generation rules. For each activity node and variable a Petri net place is constructed (`places` attribute). In case of variables, the place contains their respective initial value as token. Otherwise, only the place of the initial node has a token. The general rule for generating transitions (`transitions` attribute) is, that given an activity node, a transition is constructed for each of its predecessor nodes. The transition just consumes a token from the predecessor's place and puts it into the node's place (Figure 4 (c)).

Special means in case of control-flow edges and executable node's expressions have to be taken however. Consider Figure 4 (b). In case of control-flow edges, the respective guard must be checked before any token is consumed. To do so, it is sufficient to lookup the value encoded in the token of the place which encodes the variable the guard refers to. Further, before a token is placed by an outgoing arc, all expressions of the node its destination place represents must be executed. In Figure 4, these two actions are represented by dashed arcs from variable places to guarded input arcs and by Greek letters representing the expressions to execute.

Forks and joins are exceptions form these default rules however, because of their parallelising and synchronising semantics. In case of a fork, all its outgoing edges yield a single transition. Likewise, all incoming edges of a join are translated to a single transition (Figure 4 (a)). As an example consider the implementation of the `transitions` attribute of joins:

```
1 (JoinNode
2  (lambda (n)
3    (define incoming (=incoming n))
4    (list
5     (pn::Transition
6      (->name (car incoming))
7      (map >>? incoming)
8      (list (n>> (car incoming)))))))
```

Based on the join's incoming edges (Line 3) a new transition named like the "first" incoming edge is constructed (Lines 5 & 6). The transition has a single outgoing arc (Line 8) and for each incoming edge of the join one incoming arc[5](Line 7). These arcs are constructed by the `>>?` and `n>>` support functions which given an activity edge construct a new incoming or outgoing arc respectively. Incoming arcs consist of a single symbolic name referencing the source place the arc is consuming tokens from and a list of functions, each selecting a token to consume. Outgoing arcs consist of a single symbolic name referencing the target place the arc is producing tokens to and a single function that given all consumed tokens computes the produced ones. Consider the construction of incoming arcs via `>>?`:

```
1 (define (>>? n) ; Construct incoming Petri net arc for activity edge.
2   (if (ast-subtype? n 'ControlFlow)
3       (pn::Arc (->source n) (list (=v-accessor (=v-lookup n (->guard n)))))
4       (pn::Arc (->source n) (list (lambda (t) #t)))))
```

First, it is checked if the given activity edge is a control-flow (Line 2). If it is, the consumption function has to query the value of its guard, i.e., given a consumable token the arc is enabled if, and only if, the guard's value is *true*. To enable the querying of variable values at runtime (i.e., during Petri net execution), we construct special access functions that return the value of the token of the variable's place (`v-accessor` attribute). In case of a control-flow, `>>?` therefore finds the guard variable in the activity

[5]Incoming and outgoing arcs are consuming and producing tokens when a transition is fired respectively.

diagram via `=v-lookup` and defines its access function to be the consumption function of the arc (Line 3). If the argument of `>>?` is not a control-flow, the consumption function just returns *true*, i.e., whenever a consumable token is given the arc is enabled (Line 4). In both cases, the place to consume a token from is the given activity edge's source, i.e., `(->source n)`. All of this happens before runtime. When the generated Petri net is executed the consumption function and source are already settled by the code generation; no runtime lookup is required.

A.3.2 Variables, Expressions & The Execution of Executable Nodes

As already explained, each variable is translated to a place containing a single token encoding its value. The `v-token` attribute refers for each variable to the respective token encoding its runtime value. Its implementation queries the place representing the variable (`places` attribute), its list of tokens and finally the list's first and only child:

```
1 (ag-rule
2  v-token ; The Petri net token encoding the runtime value of the variable .
3  (Variable      (lambda (n) (ast-child 1 (pn:->Token* (=places n))))))
```

Remember, that *RACR* is incremental and caches all attributes. As long as information `places` depends on is not changed – like in the given tool contest scenario – it will construct a new Petri net place only the first time queried; further queries will evaluate to this very place. This caching behaviour holds for all attributes of the activity diagram language. Based on `v-token`, implementing `v-accessor` is straightforward:

```
1 (ag-rule
2  v-accessor ; Function returning the runtime value of the variable .
3  (Variable      (lambda (n) (define token (=v-token n)) (lambda x (pn:->value token)))))
```

First, lookup the token representing the variable's value using `v-token`. Afterwards, return a function in whose closure the token is and which uses the Petri net language to query its value via `pn:->value`.

After investigating how runtime values of variables are encoded and can be accessed, it remains to show how they are changed by expressions. The `computation` attribute generates for each expression a function assigning its left-hand the value of its right-hand. For example, consider unary expressions:

```
1 (UnaryExpression
2  (lambda (n)
3    (define assignee (=v-token (=v-lookup n (->assignee n))))
4    (define op1 (=v-accessor (=v-lookup n (->operand1 n))))
5    (define op (->operator n))
6    (lambda () (rewrite-terminal 'value assignee (op (op1))))))
```

First, the token representing the assignee is looked up (Line 3); afterwards, the access function of the operand variable and the operation to perform (Lines 4 & 5). These information are the closure of the function to construct. The function itself uses *RACR's* `rewrite-terminal` function to change the value of the assignee to the one computed by applying the operator on the value the operand's value access function returns (Line 6). Again, all lookups are at generation time of the Petri net and not runtime.

The `computation` attribute is defined for every activity node. It generates a function whose execution represents the execution of the respective activity node at runtime. This comprises three runtime actions: (1) tracing the node's execution, (2) computing its expressions if any (i.e., if the node is an executable node) and (3) establishing its offers for successor nodes:

```
1 (ActivityNode
2  (lambda (n)
3    (define executed (->name n))
```

```
4   (lambda x (trace executed) (list #t))))
5 (ExecutableNode
6  (lambda (n)
7    (define executed (->name n))
8    (define computations (map =computation (=expressions n)))
9    (lambda x (trace executed) (for-each (lambda (f) (f)) computations) (list #t))))
```

Note, that the computation functions generated by the `computation` attribute accept arbitrary many arguments and always return a singleton list with element *true*. Their tracing and expression execution is obvious (Lines 4 & 9); how token offers are established we still have to clarify however.

As already explained, for each activity node a place is generated. A token in such a place indicates that the activity node provides an offer to its successors. According to the semantics of activity diagrams, the offers of an activity edge are provided immediately *after* executing its expressions. The computation function of an activity node therefore has to be executed immediately *before* a token is put into its respective place, i.e., whenever an outgoing arc of a transition places a token in its place. Thus, outgoing arcs must apply the computation function of their target. The implementation of `>>n` therefore is:

```
1 (define (n>> n) ; Construct outgoing Petri net arc for activity edge.
2   (pn::Arc (->target n) (=computation (=target n))))
```

As explained before, an outgoing arc consists of a symbolic name referencing the target place and a production function that given the consumed tokens computes the ones placed in its target place. The functions generated by the `computation` attribute are valid production functions; they accept arbitrary many consumed tokens and place a single *true* token.

The SDMLib solution to the Model Execution Case for TTC2015

Stefan Lindel, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

slin|zuendorf@cs.uni-kassel.de

This paper describes the SDMLib solution to the Model Execution case for the TTC2015 [1]. We solved all case variants and did all performance tests. For this case we generated the Java implementation of the activity diagram classes with SDMLib in order to have an efficient model representation. Then we modeled the operations using SDMLib model transformations. These model transformations were embedded into methods of the activity diagram classes leveraging the overriding of methods for the distinction of different behavior for different kinds of activity nodes. Our solution deviates from the case description in the handling of tokens: instead of consuming and recreating tokens we use just one token and allow it to be at several places at a time and we just move the token forward through the activity diagram. This results in more elegant modeling and faster execution.

1 Introduction

We assume that the reader is familiar with the description of the TTC2015 model execution case [1]. This paper describes the SDMLib [2] solution to the TTC2015 model execution case. The task is to execute activity diagrams via model transformations. One shall show, how model transformations fit for this purpose. The case descriptions comes with an example implementation that uses a token game for execution that is borrowed from Petri Nets. Basically, the example implementation suggests that an activity node may be executed if there is a token offered at each incoming control flow arc and that the activity node consumes all these tokens, executes any inner action and creates new token offerings on each outgoing arc. Fork and join nodes get a special treatment using a sub-token that counts how many of the parallel activities have been executed already and to deduce when the join is complete.

We think the proposed token handling is pretty complicated and inefficient. To come up with a simpler solution, we removed all token related classes from the example solution and replaced them with a new Token class that has a to-many association `currentElements` to class `NamedElement`, cf. Figure 1. We use only a single `Token` object that may have multiple `currentElements` at a time. On execution, one of the `currentElements` is chosen and the corresponding link is moved forward to the next `NamedElement`. In addition, the `token` is attached to the current `Activity` via a to-one association named `token`. To count how many parallel actions have reached a join node, we use attribute `noOfVisitors` provided by class `ActivityNode`. Actually, only objects of class `JoinNode` need this attribute, but by providing it generally, the modeling of the interpreter becomes simpler.

Figure 2 shows an object diagram depicting the activity diagram of test 2 of the model execution case during execution. The `InitialNode i14` and the `ForkNode f3` have already been added to the `Trace t15`. `Activity a1` has a `Token t2` currently pointing to `ForkNode f3`. On execution, the `ForkNode` will remove itself from the set of `currentElements` of the `Token` and will add its outgoing `ControlFlow` objects `c12` and `c4` to the `currentElements` instead. In the next turn, one of the control flows (e.g. `c12`)

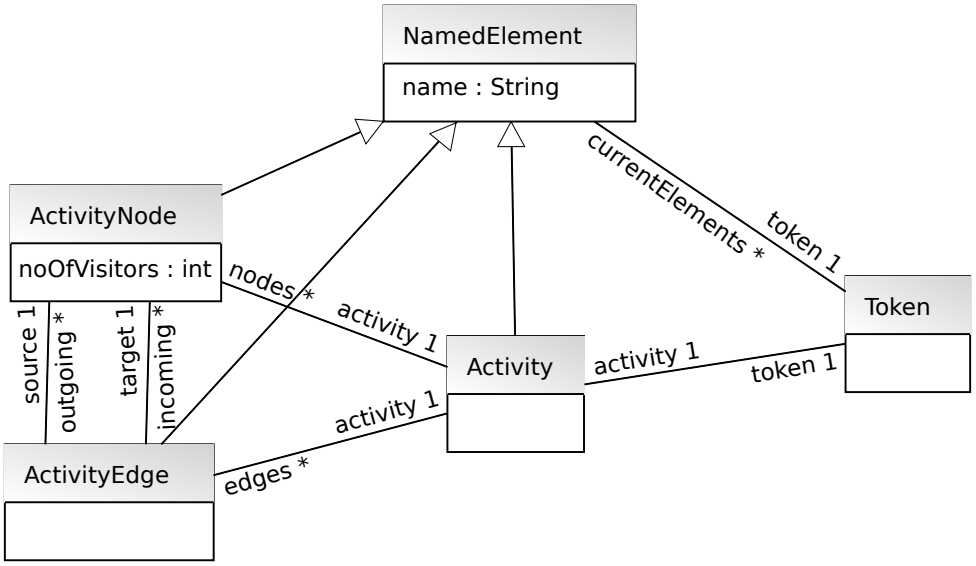


Figure 1: Simplified Token Handling

will remove itself from the `currentElements` and add its target object (e.g. `o11` instead. In addition, the `noOfVisitors` attribute of the target object is incremented. Later on, when the `JoinNode j7` is executed, `j7` checks its `noOfVisitors`. If this is lower than the number of incoming `ControlFlows`, not all parallel executions have reached the `JoinNode` yet and thus, the `JoinNode` deletes the `currentElements` link but does not forward it. Only when `noOfVisitors` indicates that all parallel branches have reached the `JoinNode`, the `currentElements` link is forwarded to the `outgoing ControlFlow`.

2 The model execution transformations

The initialization of the model execution, the handling of variables and expressions, and the overall run loop are described in the 5. To give an idea of SDMLib model transformations we discuss a generalized version of method `ActivityNode.run()`.

The overall execution identifies the current Activity node or ControlFlow edge and calls the run method of the active object. Thereby new elements become active and this is iterated until the final node is reached. Method `run()` of class `NamedElement` is overridden within its subclasses to achieve specific behavior for the various activity diagram elements. Listing 1 and Figure 3 show the general behavior of activity nodes.

```
1  class ActivityNode {
2      public void run(){
3        ActivityNodePO activityNodePO = new ActivityNodePO(this);
4
5        // add to trace
6        TracePO tracePO = activityNodePO.hasActivity().hasTrace();
7        tracePO.createExecutedNodes(activityNodePO);
8
9        // consume token
10       TokenPO tokenPO = activityNodePO.hasToken();
```

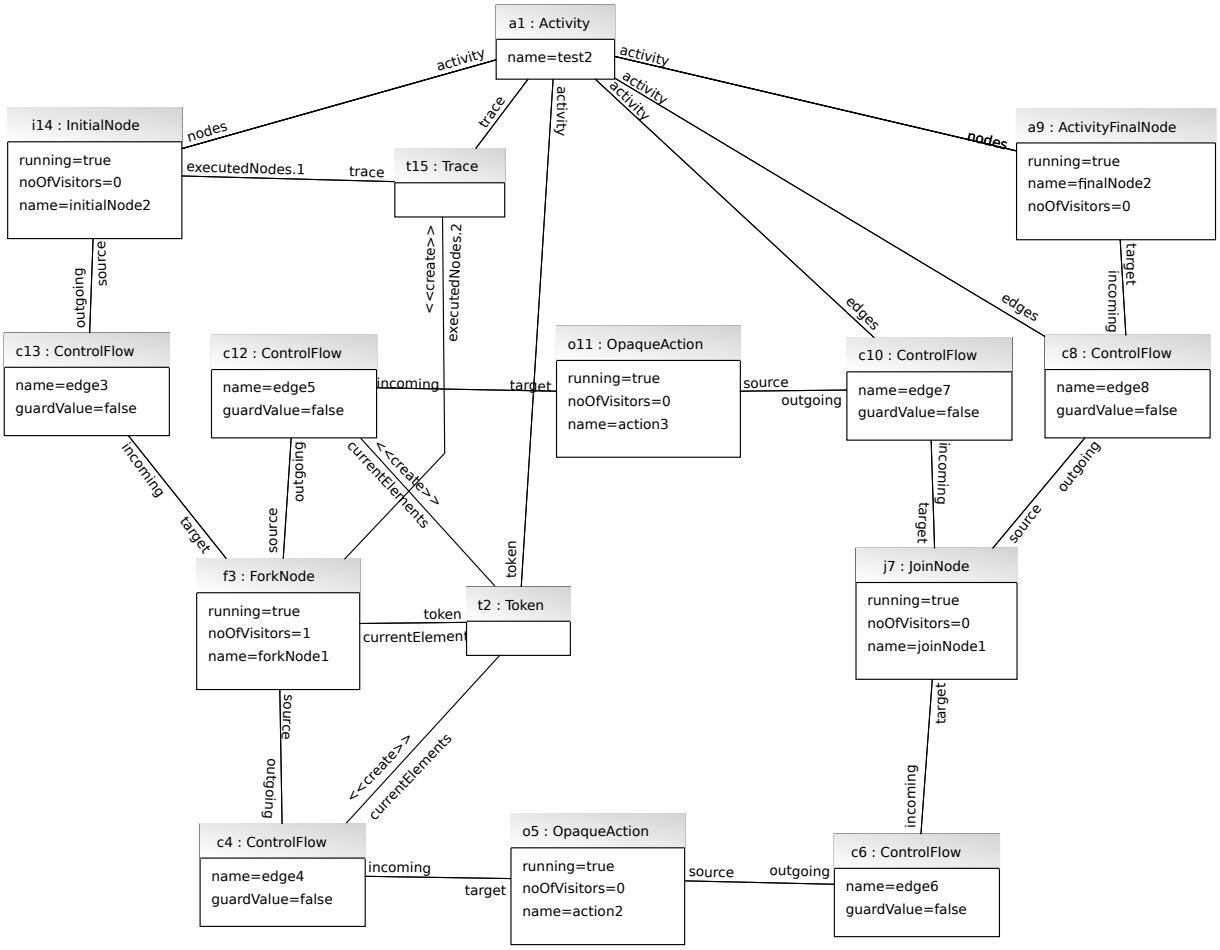


Figure 2: Moving the Token through the Activity Diagram

```
11          tokenPO.destroyCurrentElements(activityNodePO);
12
13          // forward token to all outgoing edges
14          ActivityEdgePO activityEdgePO = forkNodePO.hasOutgoing();
15
16          tokenPO.createCurrentElements(activityEdgePO);
17
18          activityEdgePO.doAllMatches();
19      }
```

Listing 1: Method ActivityNode.run() in Java

Generally, the model transformation executing an `ActivityNode` starts with an activityNodePO Pattern Object bound to the model object `this`, cf. line 3 of Listing 1. Then, line 6 uses a chain of `has` operations to look-up the owning `Activity` and the attached `tracePO`. Line 7 adds the current `ActivityNode` to the `Trace`. Then, we look up the `tokenPO` that is attached to the current `ActivityNode` (line 10) and remove the corresponding `currentElements` link (line 11). Now we forward the token. Thus, line 14 looks for

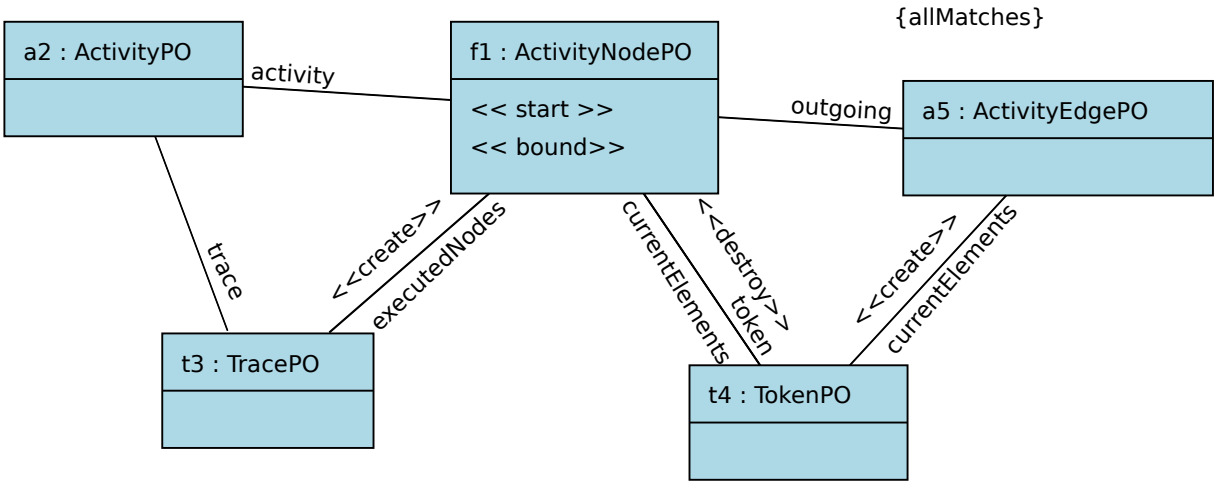


Figure 3: General ActivityNode.run() transformation

`outgoing activityEdgePO` matches and line 16 adds such `ActivityEdge` objects to the current `Token`. As there may be multiple `outgoing ActivityEdge` objects, line 18 asks the current Pattern to apply on all matches. Thus all `outgoing ActivityEdges` are added to the `currentElements`.

Note, the activity diagrams used as test cases provided by case description have no usual activity nodes that have more than one outgoing control flow. Only, fork nodes and decision nodes have multiple outgoing edges. For fork nodes, the general behavior works fine. For decision nodes, we override the `run()` method and extend the general execution pattern by a check for the guard of the outgoing `ActivityEdge`. Only if the guard is true, the corresponding activity edge is added to the `currentElements`. For decision nodes, it is guaranteed, that only one outgoing control flow has a guard that evaluates to true. Thus, we do not need an `allMatches` for decision nodes. For `JoinNodes` we just extend the general `ActivityNode.run()` pattern with a check whether the `noOfVisitors` equals the number of incomming `ControlFlows`. Only then the `Token` is forwarded.

The SDMLib implementation of the Model Execution Case provided in the SHARE environment has dedicated run methods for each kind of activity diagram element. Only when writing this paper we recognized that a common run method in class ActivityNode would have covered most cases, elegantly. After writing the paper we just had no time to update the SHARE version.

3 Results

Once we decided to come up with our own concept for moving tokens, it was pretty straight forward to develop the corresponding model transformations. The simplified token concept also resulted in model transformations that do very little search through to-many associations. The model transformations mainly look-up the current situation and and check all kinds of conditions on it. Thus, we think the execution is reasonably fast. The following table shows our performance measurements executed on a laptop with a 64 Bit Intel Dual Core i7 CPU M620 2.67GHz with 8 GB memory.

performance test	variant 1	variant 2	variant 3.1	variant 3.2
execution time (milli seconds)	9.99 ms	9.25 ms	9.38 ms	14.05 ms

For the performance measurement we did the usual tricks like warming up the Java virtual machine hot compiler by executing each activity 1000 times before measurement. We than ran each test 5 times and computed the average runtime. Overall, we think the performance test cases are a little bit to small to measure the model transformation execution time without side effects and overheads from other things running in the virtual machine.

4 Summary

Overall, the model execution case fits very well to SDMLib. It was quite straight forward to model the different execution steps and the different steps have a complexity that justifies the usage of model transformation in comparison to hand written Java code.

To some extend both the performance of our solution and the simplicity of the deployed model transformations benefit from our simplified token handling concept. However, sticking with the predefined token handling in most cases just means that the corresponding model transformations need one more statement to delete the old token and one more statement to create a new token. Thus the complexity of the model transformations would grow only slightly. The measurement of the resulting performance would be interesting.

The class model provided with the case uses a lot of inheritance and enumeration types. Actually, SDMLib can still be improved in dealing with inheritance. This is current work. Enumerations are used e.g. for the operators in expression trees. We evaluate such expression trees with usual Java code. Model transformation seem not to give leverage here.

References

[1] TTC2015 The Model Execution Case. https://code.google.com/a/eclipselabs.org/p/moliz/source/browse/?repo=ttc2015, 2015.

[2] Story Driven Modeling Library. http://sdmlib.org/, 2014.

Appendix

As a start, Listing 2 shows the Java source code that builds and runs the SDMLib model transformation initializing the variables of an activity. Figure 4 shows this transformation graphically[1].

```
1 class Activity {
2     public void initVariables(){
3         ActivityPO activityPO = new ActivityPO(this);
4         VariablePO localVariablePO = activityPO.hasLocals();
5         ValuePO valuePO = localVariablePO.hasInitialValue();
6         localVariablePO.createCurrentValue(valuePO);
7         localVariablePO.doAllMatches();
8     }
```

Listing 2: Initialize variables transformation in Java

In SDMLib a model transformation is called a *Pattern* and it consists of *Pattern Objects* and *Pattern Links* that are matched against actual model objects. For the initialization of activity variables we use a

[1] SDMLib is able to render a model transformation as HTML or SVG.

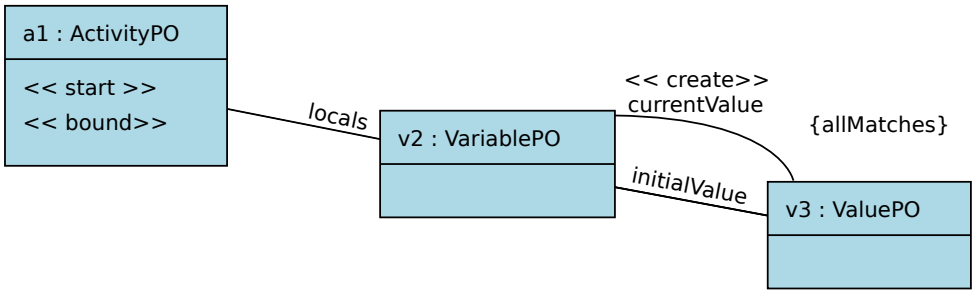


Figure 4: Initialize variables transformation

Pattern with three Pattern Objects: `activityPO`, `localVariablePO`, and `valuePO`. The constructor call `new ActivityPO(this)` creates the Pattern and adds the `activityPO` Pattern Object to it and binds `activityPO` to the current model object `this`. This means, the Pattern Object `activityPO` is directly matched against the model object `this`. It will also serve as start for the pattern matching process.

Next, the command `activityPO.hasLocals()` creates the Pattern Object `localVariablePO` and a Pattern Link of type `locals` that connects `activityPO` and `localVariablePO`. Then, the pattern matching is initiated and SDMLib tries to find model objects of type `Variable` that are connected to the current `Activity` object via a `locals` link. If there are multiple candidates, the candidates are stored for as possible matches. One of the candidates is chosen as the current match. If there is no match for a given Pattern Object, backtracking is initiated and SDMLib tries to chose other candidates for previously visited Pattern Objects and then revisits the current Pattern Object. If backtracking fails, too, the whole matching fails. In the current example case let us assume that there are two variables `v1` and `v2`. Thus Pattern Object `localVariablePO` will be matched e.g. against `v1` and `v2` will be stored as alternative candidate.

SDMLib generates the Method `hasLocals()` within class `ActivityPO` from the association `locals` between the classes `Activity` and `Variable`. For each association role such a `has` method is generated in the corresponding `PO` class. These `has` methods create a Pattern Link according to the role name and a Pattern Object according to the role's target class.

Line 5 of Listing 2 extends the search Pattern by an `valuePO` Pattern Object connected to `localVariablePO` via an `initialValue` link. Next, line 6 uses method `createCurrentValue` to extend our model transformation by an action that creates a `currentValue` link between the model objects matched by `localVariablePO` and `valuePO`. This `create` action is executed only if the Pattern has a successful match.

Finally, line 7 calls method `doAllMatches`. Method `doAllMatches` triggers the backtracking of the Pattern search, i.e. we go back to choices where still alternatives are available. In our example, this is the matching of `localVariablePO` to `var1`. Thus, `localVariablePO` is now re-matched against `v2` and the remaining pattern matching, i.e. the search for a value and the creation of a `currentValue` link is executed again. Method `doAllMatches` triggers backtracking until the Pattern search and execution fails. Overall, now all local variables of the current activity are initialized.

Model transformation `initVariables` is the first operation called within method `run()` of class `Activity`, cf. Listing 3. Similarly, method `input()` uses an `doAllMatches` transformation to assign input values to variables. Lines 5 and 6 each look-up the set of all `ActivitNode` model objects within the current activity. To implement to-many associations SDMLib generates special set classes for all model classes as in this case class `ActivityNodeSet`. These set classes inherit from a general container class and in addition for each method of the model class SDMLib generates a similar method in the corresponding set class. For example the method `withRunning(boolean)` of class `ActivityNode()` results in a similar

method in class `ActivityNodeSet`. In the set class, the generated method iterates through all contained elements and forwards the method call to each of them. Thus, line 5 of Listing 3 is finally calling method `withRunning(boolean)` on each `ActivityNode` in the current `Activity`. This sets the state of all activity nodes to running. Similarly, line 6 sets the `noOfVisitors` attribute of all activity nodes to `0`;

```
1  class Activity {
2      public void run(){
3        this.initVariables();
4        this.input(input);
5        this.getNodes().withRunning(true);
6        this.getNodes().withNoOfVisitors(0);
7
8        ActivityPO activityPO = new ActivityPO(this);
9        ActivityNodePO activityNodePO = activityPO.hasNodes();
10       InitialNodePO initialNodePO = activityNodePO.instanceOf(new InitialNodePO());
11
12       activityPO.createTrace();
13       tokenPO = activityPO.createToken();
14       tokenPO.createCurrentElements(initialNodePO);
15
16       // run the token
17       Token token = tokenPO.getCurrentMatch();
18
19       while ( ! token.getCurrentElements().isEmpty())
20       {
21          NamedElement first = token.getCurrentElements().first();
22          first.run();
23       }
24
25       this.getNodes().withRunning(false);
26     }
```

Listing 3: Method Activity.run() in Java

Lines 8 to 14 of Listing 3 build and run the central model transformation employed in method `Activity.run()`. This model transformation is shown graphically in Figure 5. Again, the Pattern starts with an `activityPO` Pattern Object bound to the current `Activity` model object, cf. line 8. This is extended by a `nodes` link to an `activityNodePO`, cf. line 9. This time we especially look for an activty node of type `InitialNode`. In the current version of SDMLib we have to use a special `instanceOf()` method to model this type check in our Pattern. This results in another Pattern Object of the desired type in line 10. In the graphical visualization this is rendered by an `instanceOf` link to another Pattern Object of the desired type. However, these two Pattern Object will match against the same model object. As this is somewhat intricate, we plan to enhance SDMLib to generate specific `hasNodesOfTypeInitalNode` methods that include the type check, internally.

Once we have identified the initial node, we create a `Trace` object (line 12) and a `Token` object (line 13). Finally, the method call `createCurrentElements(initialNodePO)` creates a `currentElements` link between the model objects matched by `tokenPO` and `initialNodePO` (line 14).

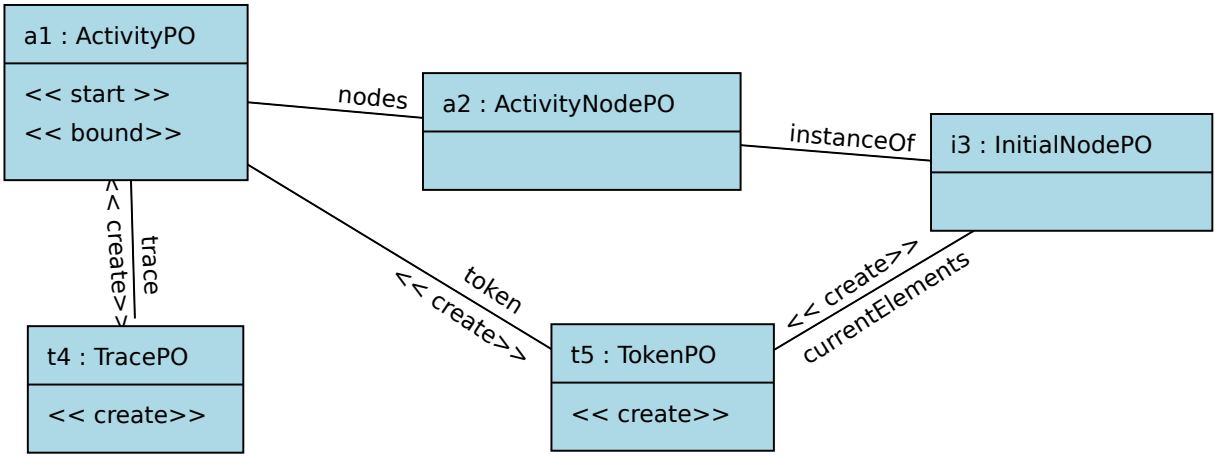


Figure 5: Starting Activity.run() transformation

Generally, the described model transformation searches through all nodes of the given activity in order to find the node of type `InitialNode`. This has a runtime complexity of O(n) in the number of activity nodes. However, in the example cases, the initial node is always the first node in the list of activity nodes. Thus, the pattern search always succeeds on the first activity node it visits and thus the actual runtime is O(1).

Once the `Trace` and the `Token` object are created, the actual execution of the activity diagram is driven by lines 17 through 23 of Listing 3. First, we look up the model object `token` that correspond to the Pattern Object `tokenPO` (line 17). The loop of line 19 uses the `currentElements` link of our `token` object as a queue, it looks-up the first element and calls `run()` on it. The `run` method will remove the corresponding `currentElements` link and add new (successor) elements to the `currentElements` instead. Note, `currentElements` may point to `ActivityNode` objects as well as to `ActivityEdge` objects. Thus, loop variable `first` uses the common super type `NamedElement`.

Listing 4 and Figure 6 show the execution of `ControlFlow` objects. Line 4 starts with a `controlFlowPO` Pattern Object bound to the current `ControlFlow` model object. Line 5 adds the current `tokenPO`. In any case, we destroy the `currentElements` link to the `Token` as the `ControlFlow` is now executed. Now we want to ensure that the guard of the `ControlFlow` allows the execution. Actually, this is not necessary as the decision node does not add a `ControlFlow` to the `currentElements` unless its guard is true. However, for completeness, `ControlFlow.run()` checks this condition, too. Unfortunately, there are two different cases to consider: first the `ControlFlow` may have no guard at all. Then it shall be consider to be true. And second, if the `ControlFlow` has a guard, than the value of that guard has to be true. To cover both cases at once, we ensure that the `ControlFlow` has no guard with value false. This may fail if there is no guard or if the guard is true. If it fails, we move the token forward. In our model transformation we use a negative application condition *NAC*, cf. line 11 through 18. The sub pattern within the NAC tries to find a match. If that succeeds, the NAC fails and the overall pattern is not executed, any more. Line 13 and 14 look-up a `Guard` at the `controlFlowPO` and test that this `Guard` is an instance of a `BooleanVariable` and that this `BooleanVariable` has a `currentValue`. Line 16 then ensures that the `currentValue` is instance of a `BooleanValue` and that the `BooleanValue` has the `value false`.

```
1 public class ControlFlow extends ActivityEdge{
2     @Override
```

```
3     public void run(){
4         ControlFlowPO controlFlowPO = new ControlFlowPO(this);
5         TokenPO tokenPO = controlFlowPO.hasToken();
6
7         // in any case remove from currentElements
8         tokenPO.destroyCurrentElements(controlFlowPO);
9
10        // add successor if guard allows
11        controlFlowPO.startNAC();
12
13        ValuePO valuePO = controlFlowPO.hasGuard()
14                              .instanceOf(new BooleanVariablePO()).hasCurrentValue();
15
16        valuePO.instanceOf(new BooleanValuePO()).hasValue(false);
17
18        controlFlowPO.endNAC();
19
20        // OK, move token
21        ActivityNodePO targetPO = controlFlowPO.hasTarget();
22
23        tokenPO.createCurrentElements(targetPO);
24
25        // count visits
26        targetPO.exec((node) -> node.incrementNoOfVisitors(1));
27    }
```

Listing 4: Method ControlFlow.run() in Java

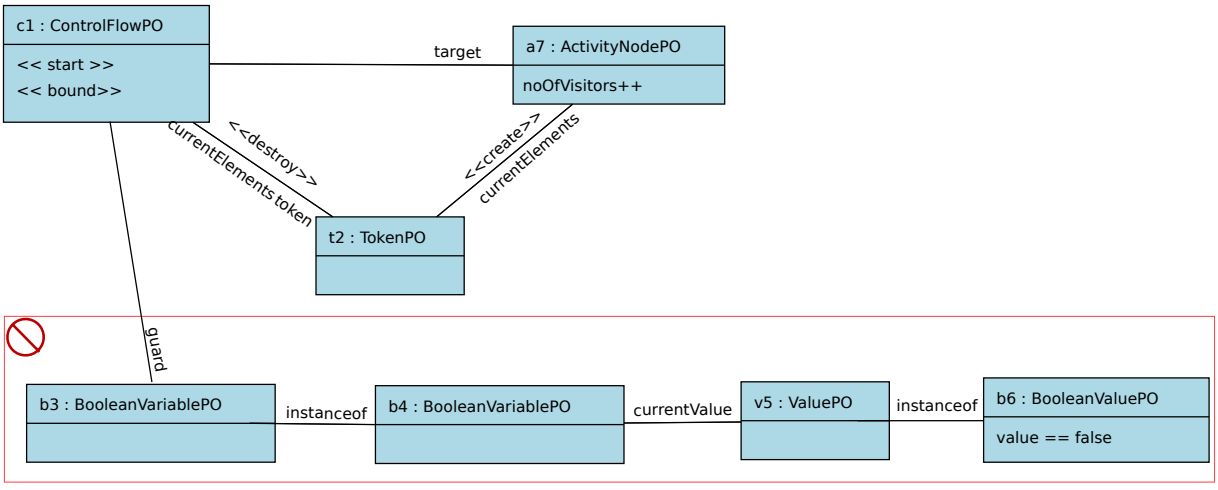


Figure 6: General ActivityNode.run() transformation

If there is no guard preventing it, line 21 of Listing 4 identifies the `target` of our `ControlFlow` and line 23 adds this target to the `currentElements`. Finally, line 26 uses a lambda expression to add an

operation to our model transformation that on execution increments the `noOfVisitors` of the `target`.

`OpaqueAction` nodes may have a number of `expressions` attached to them. `Expression` objects provide their own `run()` methods executing them. Thus, for `OpaqueAction` nodes we override the `ActivityNode` `run()` method to call the `Expression.run()` method on each expression. The expressions use various subclasses and various enumeration types to distinguish between different operations. Thus, each subclass provides its specific `run()` method and these specific `run()` methods use traditional switch statements to deal with the corresponding enumeration types, cf. Listing 5. Alternatively, we might have provided Model Patterns for each case, however evaluating expression trees is not really the application domain for model patterns.

```
1  public class IntegerCalculationExpression extends IntegerExpression
2  {
3      @Override
4      public void run()
5      {
6          IntegerValue val1 = (IntegerValue) this.getOperand1().getCurrentValue();
7          IntegerValue val2 = (IntegerValue) this.getOperand2().getCurrentValue();
8          int op1 = val1.getValue();
9          int op2 = val2.getValue();
10
11         int result = 0;
12
13         switch (this.getOperator())
14         {
15         case ADD:
16             result = op1 + op2;
17             break;
18
19         case SUBRACT:
20             result = op1 + op2;
21             break;
22
23         default:
24             throw new UnsupportedOperationException(""+this.getOperator());
25         }
26
27         this.getAssignee().setCurrentValue(new IntegerValue().withValue(result));
28     }
```

Listing 5: Method IntegerCalculationExpression.run() in Java

Solving the TTC Model Execution Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

This paper describes the FunnyQT solution to the TTC 2015 Model Execution transformation case. The solution solves the third variant of the case, i.e., it considers and implements the execution semantics of the complete UML Activity Diagram language. The solution won the *most correct solution award*.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Model Execution Case [3]. It implements the third variant of the case description, i.e., it implements the execution semantics of the complete UML Activity Diagram language. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3]. The solution has won the *most correct solution award*.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are Clojure programs using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSL) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF models and JGraLab TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases, e.g., model management, functional querying, polymorphic functions, relational querying, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and some more. For solving the model execution case, only the model management, the functional querying, and the polymorphic functions APIs have been used.

2 Solution Description

The explanations in the case description about the operational semantics on UML Activity Diagrams suggest an algorithmic solution to the transformation case. The FunnyQT solution tries to be almost a literal translation of the case description to Clojure code.

FunnyQT is able to generate metamodel-specific model management APIs. This feature has been used here. The generated API consists of element creation functions, lazy element sequence functions, attribute access functions, and reference access functions. E.g., `(a/create-ControlToken! ad)` creates a new control token and adds it to the activity diagram model `ad`, `(a/isa-Token? x)` returns true if and

[1] `http://funnyqt.org`
[2] `https://github.com/tsdh/ttc15-model-execution-funnyqt`
[3] The SHARE image name is `ArchLinux64_TTC15-FunnyQT_2`
[4] `http://clojure.org`

only if `x` is a token, `(a/all-Inputs ad)` returns the lazy sequence of input elements in `ad`, `(a/running? n)` and `(a/set-running! n true)` query and set the node `n`'s running attribute, and `(a/->locals a)`, `(a/->set-locals! a ls)`, `(a/->add-locals! a l)`, and `(a/->remove-locals! a l)` query, set, add to, and remove from the locals reference of the activity `a`[5].

In the following, the solution is presented in a top-down manner similar to how the case description defines the operational semantics of activity diagrams. The following listing shows the function `execute-activity-diagram` which contains the transformation's main loop.

```
1 (defn execute-activity-diagram [ad]
2   (let [activity (the (a/all-Activities ad))
3         trace (a/create-Trace! nil)]
4     (a/->set-trace! activity trace)
5     (init-variables activity (first (a/all-Inputs ad)))
6     (mapc #(a/set-running! % true) (a/->nodes activity))
7     (loop [en (first (filter a/isa-InitialNode? (a/->nodes activity)))]
8       (when en
9         (exec-node en)
10        (a/->add-executedNodes! trace en)
11        (recur (first (enabled-nodes activity)))))
12    trace))
```

The function queries the single activity in the diagram, creates a new trace, and assigns that to the activity. The activity's variables are initialized and its nodes are set running.

Then, a `loop-recur` iteration[6] performs the actual execution of the activity. Initially, the variable `en` is bound to the activity's initial node, which gets executed and added to the trace. Thereafter, the loop is restarted with the next enabled node. Eventually, there won't be an enabled node left, and then the function returns the trace.

The first step in the execution of an activity is the initialization of its local and input variables. The corresponding function `init-variables` is shown below. Local variables are set to their initial values, and input variables are set to the input values.

```
13 (defn init-variables [activity input]
14   (doseq [lv (a/->locals activity)]
15     (when-let [init-value (a/->initialValue lv)]
16       (a/->set-currentValue! lv init-value)))
17   (doseq [iv (and input (a/->inputValues input))]
18     (when-let [val (a/->value iv)]
19       (a/->set-currentValue! (a/->variable iv) val))))
```

After initializing the variables, the main function sets the activity's nodes running, and the main loop starts with the activity's initial node.

For different kinds of activity nodes, different execution semantics have to be encoded. This is exactly the use-case of FunnyQT's polymorphic functions (polyfn). A polymorphic function is declared once, and then different implementations for instances of different metamodel types can be defined. When the polyfn is called, a polymorphic dispatch based on the polyfn's first argument's metamodel type is performed to pick out the right implementation[7].

The next listing shows the declaration of the polyfn `exec-node` and its implementation for initial nodes. The declaration only defines the name of the polyfn and the number of its arguments (just one,

[5] The `a/` prefix denotes a namespace into which the API has been generated. The API accesses models only using EMF's generic interfaces, thus this feature does not depend on code generation on the EMF side.

[6] `loop` is not a loop in the sense of Java's `for` or `while` but a local tail-recursion. The `loop` declares variables with their initial bindings, and in the `loop`'s body `recur` forms may recurse back to the beginning of the `loop` providing new bindings for the `loop`'s variables.

[7] Polyfns support multiple inheritance. In case of an ambiguity during dispatch, e.g., two or more inherited implementations are applicable, an error is signaled.

here). The implementation for initial nodes simply offers one new control token to the initial node's outgoing control flow edge.[8]

```
(declare-polyfn exec-node [node])

(defn offer-one-ctrl-token [node]
  (let [ctrl-t (a/create-ControlToken! nil)
        out-cf (the (a/->outgoing node))
        offer  (a/create-Offer! nil {:offeredTokens [ctrl-t]})]
    (a/->add-heldTokens! node ctrl-t)
    (a/->add-offers! out-cf offer)))

(defpolyfn exec-node InitialNode [i]
  (offer-one-ctrl-token i))
```

The following listing shows the `exec-node` implementations for join, merge, and decision nodes. Join and Merge nodes simply consume their input offers and pass the tokens they have been offered on all outgoing control flows. Decision nodes act similar but offer their input tokens only on the outgoing control flow whose guard variable's current value is true[9].

```
(defn pass-tokens
  ([n] (pass-tokens n nil))
  ([n out-cf]
   (let [in-toks (consume-offers n)]
     (a/->set-heldTokens! n in-toks)
     (doseq [out-cf (if out-cf [out-cf] (a/->outgoing n))]
       (a/->add-offers!
        out-cf (a/create-Offer!
                nil {:offeredTokens in-toks}))))))

(defpolyfn exec-node JoinNode [jn]
  (pass-tokens jn))

(defpolyfn exec-node MergeNode [mn]
  (pass-tokens mn))

(defpolyfn exec-node DecisionNode [dn]
  (pass-tokens dn (the #(-> % a/->guard a/->currentValue a/value)
                       (a/->outgoing dn))))
```

How offers are consumed is defined by the `consume-offers` function shown below. The offers and their tokens are calculated. Then, the offered tokens are divided into control and forked tokens. For control tokens, their holder is unset. For forked tokens, the corresponding base token's holder is unset. The forked tokens' remainingOffersCount is decremented. If it has become zero, the forked token is removed from its holder. Lastly, the offers are deleted, and the incoming tokens are returned.

```
(defn consume-offers [node]
  (let [offers    (mapcat a/->offers (a/->incoming node))
        tokens    (mapcat a/->offeredTokens offers)
        ctrl-toks (filter a/isa-ControlToken? tokens)
        fork-toks (filter a/isa-ForkedToken? tokens)]
    (doseq [ct ctrl-toks]
      (a/->set-holder! ct nil))
    (doseq [ft fork-toks]
      (when-let [bt (a/->baseToken ft)]
        (a/->set-holder! bt nil))
      (a/set-remainingOffersCount! ft (dec (a/remainingOffersCount ft)))
      (when (zero? (a/remainingOffersCount ft))
        (a/->set-holder! ft nil)))
    (mapc edelete! offers)
    tokens))
```

[8] The FunnyQT function `the` is similar to Clojure's `first` except that it signals an error if the given collection contains zero or more than one element. Thus, it makes the assumption that there must be only one outgoing control flow explicit.

[9] `(the predicate collection)` returns the single element of the collection for which the predicate returns true. If there is no or more elements satisfying the predicate, an error is signaled.

The remaining kinds of activity nodes are fork nodes, activity final nodes and opaque actions. Their `exec-node` implementations are printed in the next listing.

A fork node consumes its offers and creates one forked token per incoming token. The incoming tokens are set as the forked tokens' base tokens, and the remaining offers count is set to the number of outgoing control flows. All created forked tokens are offered on each outgoing control flow.

```
(defpolyfn exec-node ForkNode [fn]
  (let [in-toks  (consume-offers fn)
        out-cfs  (a/->outgoing fn)
        out-toks (mapv #(a/create-ForkedToken!
                         nil {:baseToken %, :holder fn,
                              :remainingOffersCount (count out-cfs)})
                       in-toks)]
    (a/->set-heldTokens! fn in-toks)
    (doseq [out-cf out-cfs]
      (a/->add-offers! out-cf (a/create-Offer!
                               nil {:offeredTokens out-toks})))))

(defpolyfn exec-node ActivityFinalNode [afn]
  (consume-offers afn)
  (mapc #(a/set-running! % false)
        (-> afn a/->activity a/->nodes)))

(defpolyfn exec-node OpaqueAction [oa]
  (consume-offers oa)
  (mapc eval-exp (a/->expressions oa))
  (offer-one-ctrl-token oa))
```

An activity final node simply consumes all offers and then sets the running attribute of all nodes contained by the executed activity to false. An opaque action also consumes all offers, then evaluates all its expressions in sequence using the `eval-exp` function, and finally offers one single control token on the outgoing control flow.

How an expression is evaluated depends on (1) its type and (2) on the value of its operator attribute. The expression's type is only important in order to separate unary from binary expressions, and the operator defines the semantics. Therefore, the `eval-exp` function shown in the next listing has a special case for boolean unary expressions which negates the expression's current value using `not`. For all binary expressions, the map `op2fn` mapping from operator enum constants to Clojure functions having the semantics of that operator is used. The function determined by looking up the expression's operator is applied to both operands to compute the new value.

```
(def op2fn {(a/enum-IntegerCalculationOperator-ADD)           +
            (a/enum-IntegerCalculationOperator-SUBRACT)       -
            (a/enum-IntegerComparisonOperator-SMALLER)        <
            (a/enum-IntegerComparisonOperator-SMALLER_EQUALS) <=
            (a/enum-IntegerComparisonOperator-EQUALS)         =
            (a/enum-IntegerComparisonOperator-GREATER_EQUALS) >=
            (a/enum-IntegerComparisonOperator-GREATER)        >
            (a/enum-BooleanBinaryOperator-AND)                #(and %1 %2)
            (a/enum-BooleanBinaryOperator-OR)                 #(or  %1 %2)})

(defn eval-exp [exp]
  (a/set-value! (-> exp a/->assignee a/->currentValue)
                (if (a/isa-BooleanUnaryExpression? exp)
                  (not (-> exp a/->operand a/->currentValue a/value))
                  ((op2fn (a/operator exp))
                   (-> exp a/->operand1 a/->currentValue a/value)
                   (-> exp a/->operand2 a/->currentValue a/value)))))
```

After executing all enabled nodes, the transformation's main function `execute-activity-diagram` recomputes the enabled nodes and resumes the execution. The enabled nodes are computed by the `enabled-nodes` function shown in the following listing. The enabled nodes are those nodes of a given

activity which are set running, are no initial nodes[10], and receive an offer on each incoming control flow, or, in the case of a merge node, on one incoming control flow.

```
95 (defn enabled-nodes [activity]
96   (filter (fn [n]
97             (and (a/running? n)
98                  (not (a/isa-InitialNode? n))
99                  ((if (a/isa-MergeNode? n) exists? forall?)
100                  #(seq (a/->offers %)) (a/->incoming n))))
101          (a/->nodes activity)))
```

These 101 NCLOC of algorithmic FunnyQT/Clojure code implement the complete operational semantics of UML Activity Diagrams (with the exception of data flows which has not been demanded by the case description).

3 Evaluation & Conclusion

The solution comes with a test suite, and during the official evaluation achieved a full *correctness* score winning the *most correct solution award*.

With 101 lines of non-commented source code, the FunnyQT solution is quite *concise*. Of course, *understandability* is a very subjective measure measure. The solution should be evident for any Clojure programmer but even without prior Clojure knowledge, the solution shouldn't be hard to follow due to the usage of the metamodel-specific API. Another strong point is that all steps in the execution of an activity are encoded in one function each whose definition is almost a literal translation of the English description to FunnyQT/Clojure code.

The following table shows the *performance* in terms of execution times of the FunnyQT solution for all provided test models. These times were measured on a normal 4-core laptop with 2.6 GHz and 2 GB of RAM dedicated to the JVM.

Model	Time	Model	Time	Model	Time
test1	1.3 ms	*test5*	0.5 ms	*performance-variant-2*	1246.5 ms
test2	0.6 ms	*test6 (false)*	3.7 ms	*performance-variant-3-1*	1159.6 ms
test3	4.1 ms	*test6 (true)*	5.4 ms	*performance-variant-3-2*	72.7 ms
test4	3.2 ms	*performance-variant-1*	1104.0 ms		

When compared with the reference Java solution, the FunnyQT solution is slightly faster for all normal and performance test models, and about 8 times faster for the *performance-variant-3-1* model.

Overall, FunnyQT seems to be very adequate for defining model interpreters. Especially its polymorphic function facility has been explicitly designed for these kinds of tasks.

References

[1] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT, Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[2] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015.*

[3] Tanja Mayerhofer & Manuel Wimmer (2015): *The TTC 2015 Model Execution Case*. In: *Transformation Tool Contest 2015.*

[10] Initial nodes have to be excluded because if they are set running, all of their (zero) incoming control flows have offers.

Part II.
The Java Refactoring Case

Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation

Géza Kulcsár, Sven Peldszus, and Malte Lochau

TU Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt
{geza.kulcsar@es|sven.peldszus@stud|malte.lochau@es}.tu-darmstadt.de

Abstract. In this case study for the transformation tool contest (TTC), we propose to implement object-oriented program refactorings using transformation techniques. The case study proposes two major challenges to be solved by solution candidates: (1) bi-directional synchronization between source/target program source code and abstract program representations, and (2) program transformation rules for program refactorings. We require solutions to implement at least two prominent refactorings, namely Pull Up Method and Create Superclass. Our evaluation framework consists of collections of sample programs comprising both positive and negative cases, as well as an automated before-after testing procedure.

1 Introduction

Challenges resulting from software aging are well known but remain open. An approach to deal with software aging is refactoring. Concerning object-oriented (OO) programs in particular, most refactorings can be formulated and applied to a high-level structure and there is no need to go down to the instruction level. Nevertheless, most recent implementations usually rely on ad-hoc program transformations directly applied to the AST (Abstract Syntax Tree). A promising alternative to tackle the challenge of identifying those (possibly concealed) program parts being subject to structural improvements is *graph-based refactoring*.

Here, the program is transformed into an abstract and custom-tailored program graph representation that (i) only contains relevant program elements, and (ii) makes explicit static semantic cross-AST dependencies, being crucial to reason about refactorings. Nevertheless, certain language constructs of more sophisticated programming languages pose severe challenges for a correct execution of refactorings, especially for detecting refactoring possibilities and for verifying their feasibility. As a consequence, the correct specification and execution of refactorings for OO languages like Java have been extensively studied for a long time in the literature and, therefore, can not serve as scope for a TTC case study to their full extent. Therefore, we propose the challenge of graph-based

refactorings to be considered on a restricted sub-language of Java 1.4, further limited to core OO constructs of particular interest for the respective structural patterns.

A solution should take the source code of a given Java program as input and apply a given refactoring to an appropriate representation of that program. Ideally, a program graph conforming to a predefined type graph is created on which the refactorings are executed and, afterwards, propagated back to the source code. However, refactorings on other representations of the source code are also allowed as long as the source code is appropriately changed. To summarize, this case has two main challenges in its full extent and a subset of these in the basic case:

I **Bidirectional and incremental synchronization of the Java source code and the PG.** This dimension of the case study requires special attention when it comes to maintaining the correlation between different kinds of program representation (textual vs. graphical) and different abstraction levels. Additionally, the code and the graph representation differ significantly w.r.t. the type of information that is displayed explicitly, concerning, e.g., method calls, field accesses, overloading, overriding etc. As the (forward) transformation of a given Java program into a corresponding PG representation necessarily comes with loss of information, the backward transformation of (re-)building behavior-preserving Java code from the refactored PG cannot be totally independent from the forward transformation – a correct solution for this case study has to provide some means of restoring those parts of the input program which are not mapped to, or reflected in the PG.

II **Program refactoring by PG transformation.** In our case study, refactoring operations are represented as rules consisting of a left-hand side and a right-hand side as usual. The left-hand side contains the elements which have to be present in the input and whose images in the input will be replaced by a copy of the right-hand side if the rule is applied. Therefore, the actual program refactoring part of our case study involves in any case (i) the specification of the refactoring rules are based on refactoring operations given in a semi-formal way, (ii) pattern matching (potentially including forbidden patterns, recursive path expressions and other advanced techniques) to find occurrences of the pattern to be refactored in the input program and (iii) a capability of transforming the PG in order to arrive at the refactored state. Note that the classical approach to program refactoring (which is used here) never goes deeper into program structure and semantics than high-level OO building blocks, namely classes, methods and field declarations; the declarative rewriting of more fine-grained program elements such as statements and expressions within method bodies is definitely out of scope of our case study for TTC.

Each challenge can be solved in a basic version (with an arbitrary intermediate representation), and in an extended version (using a separate, intermediate representation that is at least isomorphic to our proposed type graph). Two exemplary refactoring operations should be implemented when solving this case

study. The first one, `Pull Up Method` is a classical refactoring operation – our specification follows that of [1]. `Pull Up Method` addresses Challenge II to a greater extent. The second one, `Create Superclass` is also inspired by the literature, but has been simplified for TTC. It can be considered as a first step towards factor out common elements shared by sibling classes into a fresh superclass. In contrast to `Pull Up Method`, new elements have to be created and appended to the PG. `Create Superclass`, therefore, comes with more difficulties regarding Challenge I especially if a program graph is used.

In the following, we give a detailed description of the case study to be solved by specifying the constituting artifacts, (meta-)models and transformations in Section 2. The two sample refactoring operations mentioned above are elaborated (including various examples) in Section 3. The correctness of the solutions is tested concerning sample input programs together using an automated before-after testing framework containing executable program test cases. Some test cases are based on the examples of Section 3, while some of them are hidden from the user – these cases check if the refactorings have been carefully implemented such that they also handle more complex situations correctly. Further details about this framework, the additional evaluation criteria, and the solution ranking system can be found in Section 4.

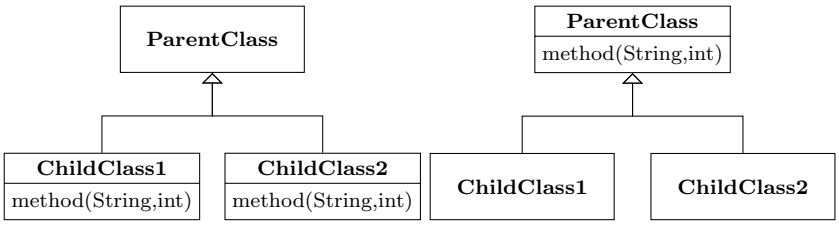
Based on the demanded functionality to be implemented by all solutions for the case study, further interesting extensions to those core tasks are mentioned in Section 5.

2 Case Description

Before diving into the details of the actual scenario to cope with, we motivate our case study once again by recalling the aim of refactorings. For this purpose, we use the very words of Opdyke, the godfather of refactorings, which say that refactoring is the same as "restructuring evolving programs to improve their maintainability without altering their (externally visible) behaviors" [2]. Hence, solutions of our case study have to (and, hopefully, want to) demonstrate the power of their chosen transformation tool by implementing refactorings as program transformation, with optional model-to-code incremental change propagation.

To describe the case study in a nutshell, we provide an intuitive example here, describing a program state where a natural need for restructuring arises.

Example. Refactoring Scenario 1 shows a basic example for a refactoring of a simple program. The source code of this program is shown in Appendix 1a. In this case, we expect that a program transformation takes place which moves `method` from all child classes of the class `ParentClass` to this same superclass. (This is a classical refactoring which is called `Pull Up Method` and builds a significant part of our case study. `Pull Up Method` will be further specified and exemplified in Section 3.)



(a) Class Diagram of Source Code 1 (b) Class Diagram after the Refactoring of Source Code 1

Refactoring Scenario 1: Structure of the Java Program before and after the Application of the Refactoring *pum(ParentClass, method(String, int))*

In the following, we give a schematic overall picture of the intended transformation chain (Figure 1) and its constituting artifacts. Solid arrows denote the extended challenge, while the dashed arrow shows the basic challenge not using a PG representation. The basic challenge can include an arbitrary intermediate representation. In Section 2.1, some details regarding the input Java code and the PG meta-model (called the *type graph*) are given, while Section 2.2 provides information on the individual transformation steps and the arising difficulties.

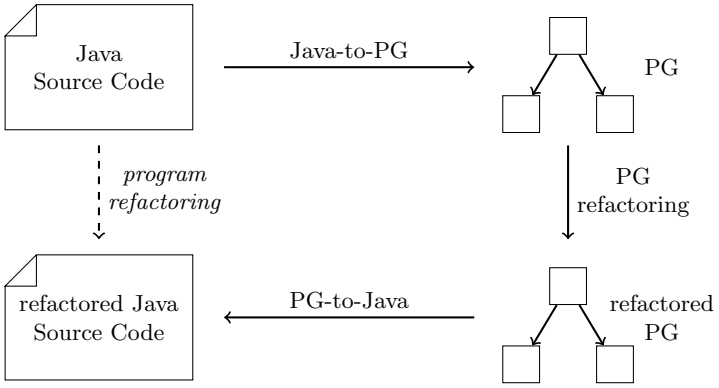


Fig. 1: Sketch of the Transformation Chain

2.1 Setting of the Case Study

Java Source Code All input programs considered for refactoring for TTC are fully functioning (although, abstract) Java programs built for the very purpose of checking the correct and thorough implementation of the given refactoring

operations. Some test input programs are openly available and will be also described later on, while some others serve as blind tests and are not accessible to the solution developers.

The Java programs conform to the Java 1.4 major version. Moreover, the following features and language elements are explicitly *out of scope* for this case study:

- access modifiers (all elements have to be `public`)
- interfaces
- constructors
- the keywords `abstract`, `static` and `final` except for `public static void main(String[] args)`
- the keyword `super`
- exception handling
- inner, local and anonymous classes
- multi-threading (`synchronized`, `volatile`, ...)

On the other hand, we would like to point out that the following Java language elements and constructs should be considered:

- inheritance
- method calls, method overloading and method overriding
- field accesses and field hiding
- libraries

To detect external libraries, editable classes must have an identical root package which is not the root package of any used library.

Type Graph for Representing Java Programs Figure 2 shows the type graph meta-model that is part of the extended case study assets as an EMF meta-model – nevertheless, other meta-modeling technologies are allowed in solutions as well. If the solution is designed for the extended challenge, a program graph is only allowed to contain the information visualized in Figure 2. For a technical realization of the shown types, references, and attributes tool depended tuning is allowed. It is not allowed to make additional information available in the PG.

In conformance with the restrictions on the considered Java programs and with the nature of classical refactoring, the type graph does not include any modeling possibilities for access modifiers, interfaces, etc. and any code constituents lying deeper than the method level. In the following, we describe the meaning of some of the most important nodes and edges of the type graph.

The type graph represents the basic structure of a Java program. The node `TypeGraph` serves as a common container for each program element as the root of the containment tree. The Java package structure is modeled by the node `TPackage` and the corresponding self-edge for building a package tree. The node `TClass` stands for Java classes and contains members (the abstract class `TMember`), which can be method and field definitions (`TMethodDefinition` or

6

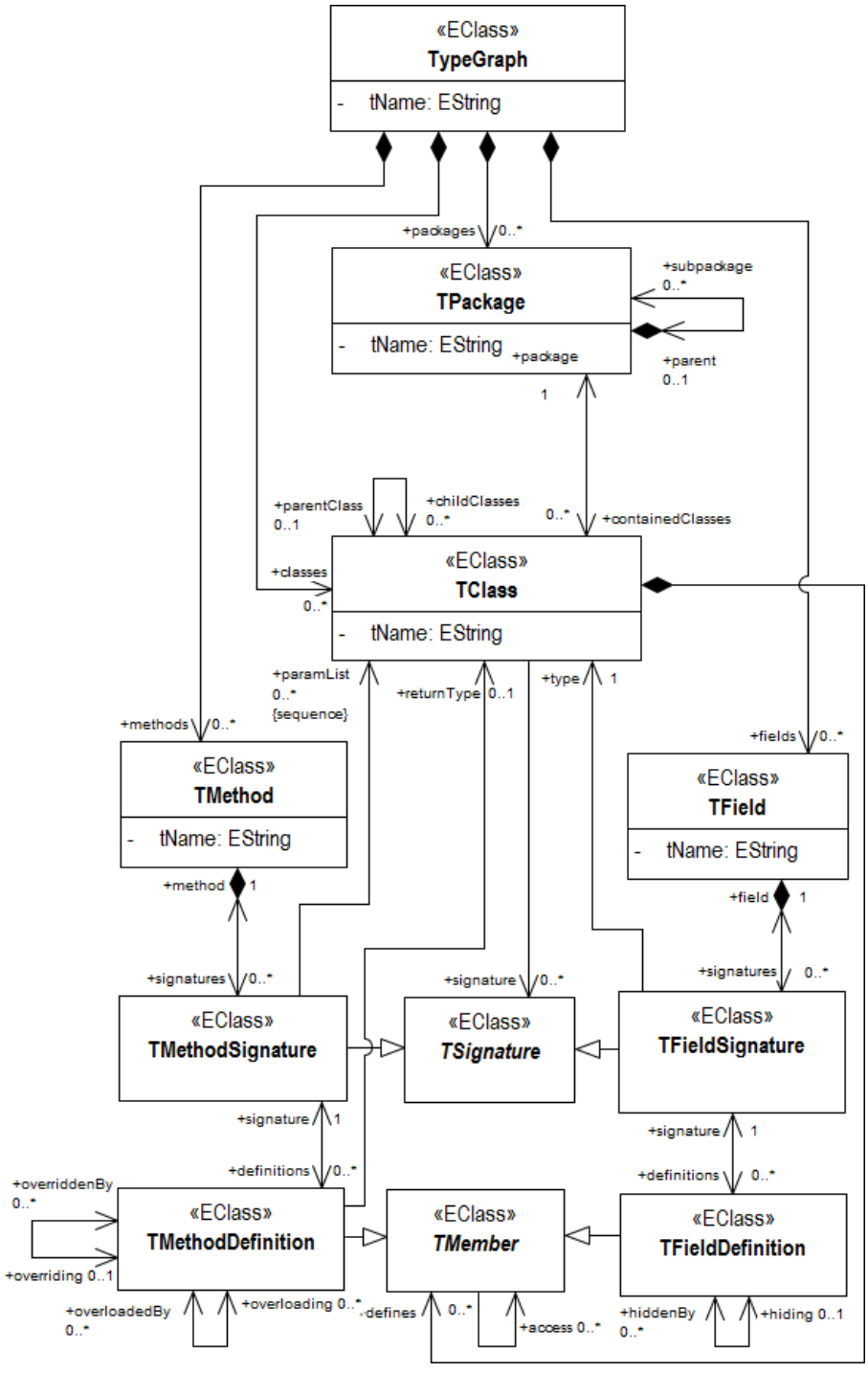


Fig. 2: Meta-model of the Proposed Type Graph

`TFieldDefinition`, respectively). In addition, a `TClass` refers to the abstract class `TSignature`, which is the common ancestor of method and field signatures.

Methods and fields are represented by a structure consisting of three elements:

- The name of the method (field) contained in the attribute `tName` of `TMethod` (`TField`), which is globally visible in the PG.
- The signatures of the methods (fields) of this name, represented by the class `TMethodSignature` (`TFieldSignature`). The signature of a method consists of its name and its list of parameter types `paramList`, while the signature of a field consists of its name and its `type`. Different signatures having the same name (i.e., a common container `TMethod` or `TField`) allow overloading. Signatures have a central role in the Java language, as all method calls and field accesses are based on signatures.
- `TMethodDefinition` (`TFieldDefinition`) is an abstraction layer representing the instruction level of Java. Relevant information is expressed by reference edges in the type graph. Overloading and overriding is declared by the corresponding edges between definition instances, although the overloading/overriding structure is also implicitly given through signatures, definitions and inheritance. The `access` edges between member instances represent dependencies between one member and the other members. This single edge type stands for all kinds of semantic dependencies among class members, namely *read*, *write* and *call*.

2.2 Transformations

The transformation chain for the extended challenge consists of three consecutive steps which are detailed here. For the basic challenge, no obligatory transformation chain is demanded.

First Step: Java Code to Program Graph Given a Java program as described in Sec. 2.1, it has to be transformed into an abstract PG representation conforming to the type graph meta-model (*ibid.*). Important note: the fact that some information necessarily disappears during this transformation calls for a solution where some preservation technique is employed, i.e., it is possible to rebuild those parts in the third step (see below) which are not present in the PG.

We remark that any intermediate program representations like JaMoPP[1], MoDisco[2], AST models etc., are allowed to facilitate the Java-to-PG and PG-to-Java transformations.

[1] `http://www.jamopp.org`
[2] `http://eclipse.org/MoDisco/`

Second Step: Refactoring of the Program Graph This step essentially consists in an endogenous (PG-to-PG) restructuring of the program graph, according to the specifications of the refactoring operations `Pull Up Method` resp. `Create Superclass`. For those specifications and actual refactoring examples, see Sec. 3.

Third Step: Program Graph to Java Code As already mentioned at the first step (Java-to-PG), one of the most difficult tasks is to create a solution which provides a means to recover the program parts not included in the PG when transforming its refactored state back into Java source code. In other words, it is impossible to implement the Java-to-PG and the PG-to-Java transformations (the first and the third step) independently of each other. Furthermore, over the challenges posed by the abstraction level of the PG, one has to pay extra attention if a newly created PG element has to appear in the refactored code.

The resulting Java code has to fulfill the requirements of (i) having those code parts unchanged which are not affected by the refactoring and (ii) retaining the observable behavior of the input program. These properties are checked using before-after testing (as usual in the case of behavior-based test criteria) provided by the automated test framework that is part of the case study and is further described in Section 4.

After this brief overview of both the static and the dynamic ingredients of the transformation scenario to be dealt with, we proceed as follows: In Section 3, we put the second step in Sec. 2.2 under the microscope and present the two aforementioned refactoring operations with associated examples to also provide an intuition how and why they are performed. Thereupon, in Section 4, we describe our automated before-after testing framework for checking the correctness of the implementations, which also serves as a basis for the solution ranking system described in the same section including further evaluation criteria.

3 Refactorings

In the following, we provide an informal specification of the requested refactorings.

3.1 Pull-up Method

First, we provide an intuition of `Pull Up Method` textually. Additionally, we give some further information and examples to clarify the requirements.

Situation and action. There are methods with identical signatures (name and parameters) and equivalent behaviours in direct subclasses of a single superclass. These methods are then moved to the superclass, i.e., after the refactoring, the method is a member of the superclass and it is deleted from the subclasses.

Graphical representation. Figure 3 shows a schematic representation of how `Pull Up Method` is performed. We use the elements of the type graph introduced in Sec. 2.1 and a notation with left- and right-hand sides as usual for graph transformation. Here, the left-hand side shows which elements have to be present or absent in the PG when applying the refactoring to it; an occurrence of the left-hand side is replaced by the right-hand side by preserving or deleting the elements of it and optionally creating some new elements and gluing them to the PG. It is implicitly given through object names which parts are preserved. In addition, we explicitly show the parts to be deleted on the left-hand side in red and marked with `--` and the parts to be created on the right-hand side in green and marked with `++`. The left-hand side also includes a forbidden pattern or *NAC*, which in this case consists of a single edge and is shown crossed through and is additionally highlighted in blue. This edge has to be absent in the input graph for the refactoring to be possible. Patterns within stacked rectangles may match multiple times.

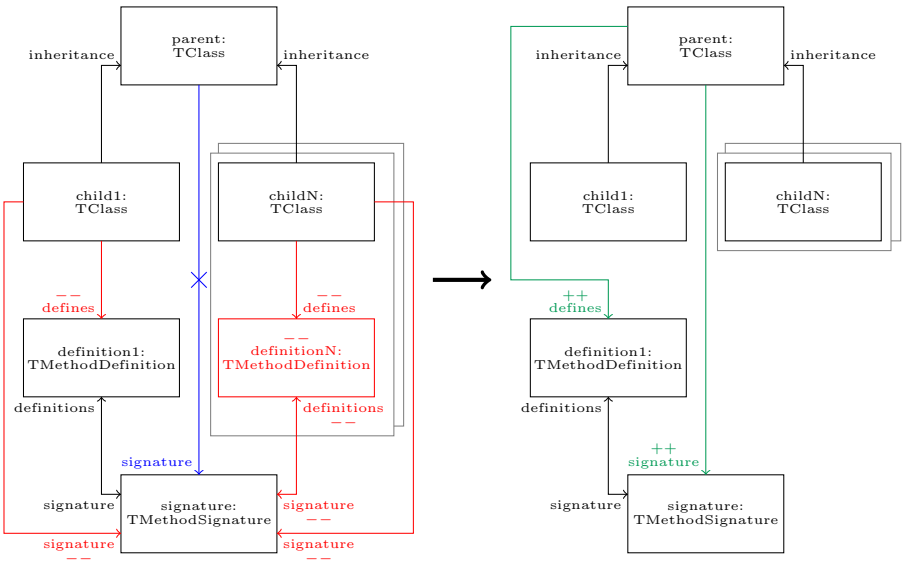


Fig. 3: Schematic Representation of a `Pull Up Method` Refactoring - Left-Hand and Right-Hand Side

Definition. In this case study, a `Pull Up Method` refactoring is specified as `pum(parent, signature)` with the following components:

– a superclass `parent`, whose direct child classes are supposed to contain at least one equivalent method implementation, and

– the method signature `signature` of such an equivalent method implementation, which represents the method to be pull-upped to `parent`.

Note that a signature consists of a name and a parameter list. The return type is not part of the signature. Anyway, within a class hierarchy, all return types of the method definitions of a signature have to be covariant.

Two equivalent implementations of a signature do not necessarily have identical implementations. Only their behavior is crucial. As proving that two implementations have identical behavior is undecidable, this decision has to be taken by a developer before initiating the refactoring.

In case the application conditions (see below) are fulfilled, the method signature `signature` as well as a corresponding method definition will be part of the `parent`. The copies of the other definitions of `signature` will be deleted from all child classes. Note that a `Pull Up Method` instance does not necessarily represent a valid refactoring - it marks merely a part of the input program where it is looked for a possible pull-up action.

Application conditions. In addition to the conditions shown in Figure 3, the following preconditions have to be fulfilled for a `Pull Up Method` refactoring instance `pum(parent, signature)`:

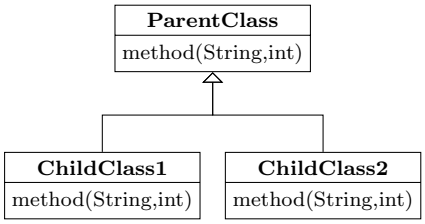
1. Each child class of the class `parent` has at least one common method signature `signature` with the corresponding method definitions ($definition_i$ for the i-th child class) having equivalent functionality.
2. Each $definition_i$ of `signature` in the child classes is only accessing methods and fields accessible from `parent`. Methods and fields defined in the child classes are not direct accessible .
3. The `parent` does not belong to a library and is editable.

Important remarks. Although it is not explicitly shown in Figure 3, all access edges in the PG pointing to a method definition deleted by the refactoring have to be redirected to point to the one which is preserved, so that subsequent refactorings are able to consider a coherent state of the PG. The actual choice of the preserved definition is irrelevant and the definitions can be arbitrarily matched, as the actual method implementations are out of scope for this case study. If methods have different return types, then a conservative behavior, such as the denial of the refactoring is allowed.

Examples

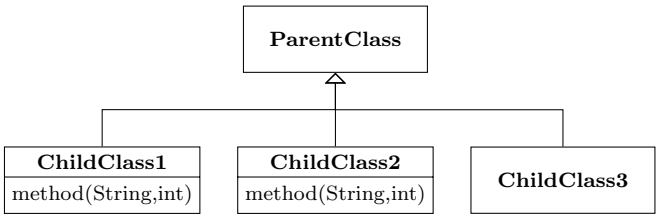
Example 1. Our first and most basic example for `Pull Up Method` is the one we have already shown as a general motivation for refactoring in the introduction part of Section 2.

Example 2. Given the program in Refactoring Scenario 2, the `Pull Up Method` refactoring `pum(ParentClass, method(String, int))` seen in the previous example is not possible. In `ParentClass`, a method with the given signature is already present which is overridden by methods in `ChildClass1` and `ChildClass2`. Accordingly, the NAC shown on the left-hand side of Figure 3 is violated.



Refactoring Scenario 2: Refactoring `pum(ParentClass, method(String, int))` not possible – `method(String, int)` already exists in `ParentClass`

Example 3. Given the program in Refactoring Scenario 3, the `Pull Up Method` refactoring `pum(parent, method(String, int))` is not possible. In this case, Precondition 1 is not fulfilled as `ChildClass3` does not contain the common method with the signature `method(String, int)`.



Refactoring Scenario 3: Refactoring `pum(ParentClass, method(String, int))` not possible – one of the child classes does not have `method(String, int)`

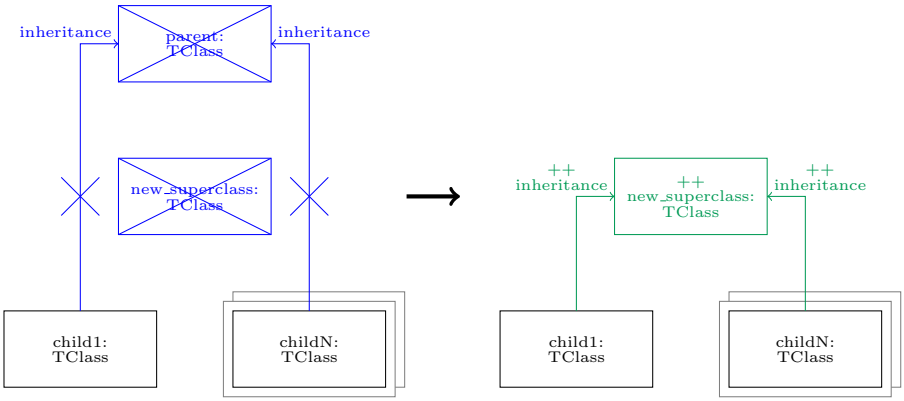
All examples shown here also have a corresponding test case in our test framework which is described in Sec. 4, with the example programs being accessible to the solution developers. In addition, there are some built-in test cases that are hidden in the framework and check trickier situations. For each of these hidden test cases, a textual hint for its purpose is provided by the test framework.

3.2 Create Superclass

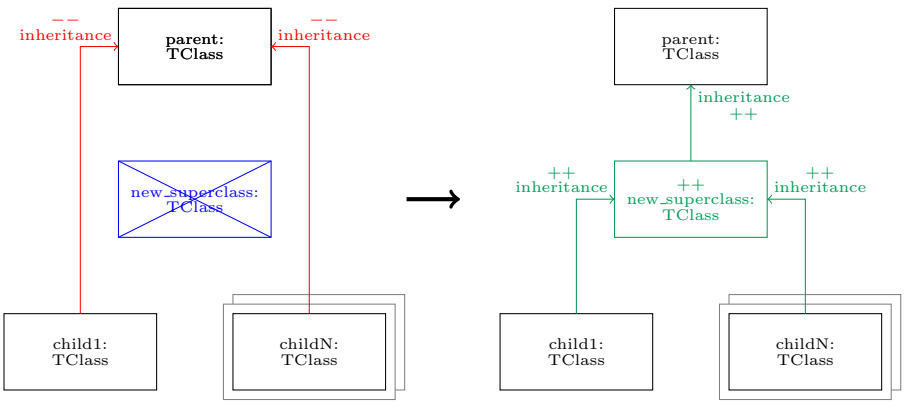
The refactoring operation `Create Superclass` is described in a similar fashion as the `Pull Up Method` refactoring above.

Situation and action. There is a set of classes with similar features. As a first step towards an improved program structure, a new common superclass of these classes is created.

Graphical representation. Figure 4 shows a schematic representation of how the `Create Superclass` refactoring is performed with the same notation as by the `Pull Up Method` refactoring above. The classes either has to have the same superclass in the PG or none of them has a superclass modeled in the PG. (Note that from a technical point of view, each Java class has a superclass. Also, the distinction above refers to the representation in the PG.) Here, both cases are shown.



(a) The classes have no superclass in the PG



(b) All classes have the same superclass in the PG

Fig. 4: Schematic Representation of a `Create Superclass` refactoring – Left-Hand Side and Right-Hand Side

Definition. In this case study, a `Create Superclass` instance is defined as `csc(classes, new_superclass)` consisting of the following components:

- a list of classes `classes`, where all classes have identical inheritance relations (i.e., each of them inherits from the same class or they do not inherit from any class in the PG), and
- a superclass `new_superclass`, which does not exist before the refactoring and has to be generated.

In case the application pre- and postconditions (see below) are fulfilled, a new class `new_superclass` will be created which becomes the superclass of the classes in `classes`. Note that a `Create Superclass` refactoring does not necessarily represent a valid refactoring - it marks merely a part of the input program where it is looked for a possible refactoring operation.

Application conditions. In addition to the conditions shown in Figure 4, the following precondition has to be fulfilled for a `Create Superclass` instance `csc(classes, new_superclass)`:

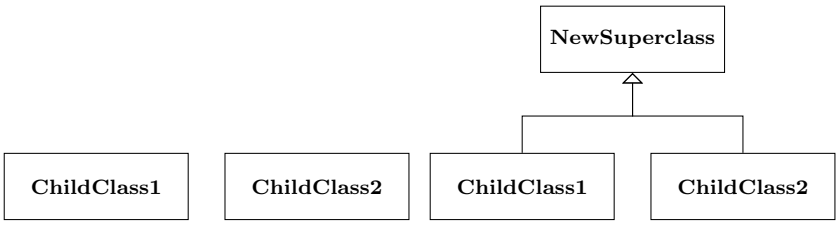
1. The classes contained in `classes` are implementing the same superclass. Note that classes with no explicit inheritance reference in Java are implementing `java.lang.Object` – modeling this class explicitly in the PG is a developer decision which does not influence the conditions for `Create Superclass`.

Additionally, the result of $csc(classes, new\_superclass)$ has to fulfil the following postconditions:

1. Each class in `classes` has an inheritance reference to `new_superclass`.
2. In case the classes in `classes` had an explicit inheritance reference to a superclass `parent` before the refactoring, their new superclass `new_superclass` has an inheritance reference to `parent`.

Examples

Example 1: Refactoring Scenario 4 shows the most basic example on which `Create Superclass` is applicable. The refactoring operation `csc({ChildClass1, ChildClass2}, NewSuperclass)` is possible as the desired new class does not exist yet.



(a) Class Diagram before the Refactoring

(b) Class Diagram after the Refactoring

Refactoring Scenario 4: Structure of the Java Program before and after the Application of the Refactoring `csc({ChildClass1,ChildClass2}, NewSuperclass)`

As demonstrated by the previous example, the `Create Superclass` refactoring itself is relatively uncomplicated, however, there are additional hidden test cases in the framework for `Create Superclass` as well. Note that, as already stated before, the main challenge by this refactoring is not to restructure the PG, respectively the chosen intermediate representation in the basic challenge, but to propagate the new element into the Java source code.

4 Evaluation

In this section, we introduce our test framework ARTE (Automated Refactoring Test Environment) for checking the correctness of implementations (Sec. 4.1) and the criteria and the scoring system which will be used to evaluate and rank the submitted solutions (Sec. 4.2).

4.1 Test Framework

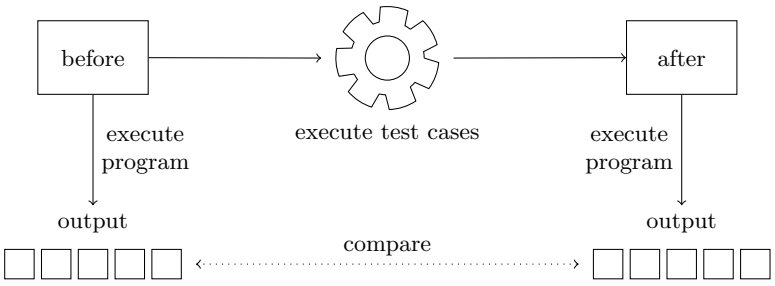


Fig. 5: Schematic Process of Before-after Testing

To enable the evaluation and ranking of the solutions for our case study, we have created an automated refactoring testing environment called ARTE,

whose mechanism is sketched in Figure 5. This test framework relies on the well-known principle of before-after testing, which is often used in behavior-critical scenarios: the behavior of the input is determined by stimulating it through the test environment and it is then checked if the output of the transformation reacts identically to the same stimulation.

In our framework, before-testing consists in compiling and executing the program and recording its console output. On the other hand, after-testing consists in compiling and executing the refactored program created by the actual solution under test, and comparing its console output to the one recorded in the before-testing phase.

The testing procedure is described in *test cases*. A test case consists of the following:

- a Java program assigned to it, on which the transformation takes place (one program can be assigned to multiple test cases) and
- a sequence of commands which can be (i) actual transformation operations or (ii) assertions to check if the transformations provided the expected result (e.g., nothing has changed if there is no correct refactoring possible). Note that a transformation operation cannot be executed without a corresponding assertion check for success.

The *execution* of a test case comprises the following steps:

- the before-testing phase as described above,
- the execution of the commands in the test case and
- the after-testing phase as described above.

For further details on how to use our testing framework ARTE and how to write individual test cases, please refer to the ARTE handbook.

Beyond the ones mentioned above, the number of imaginable extensions regarding the supported refactorings or the framework is unlimited. The reviewers can also reward some other creative extension approaches using the extension score.

4.2 Ranking Criteria and Scoring System

In this section, we propose a systematic way of evaluating and ranking the solutions for the case study.

There is a total of 100 points that can be achieved by a solution. These 100 points are composed as follows (with a detailed description of the various aspects thereafter):

- max. 60 points: correctness and completeness (successful execution of test cases)
- max. 10 points: comparison of the execution times of the solutions
- max. 30 points: quality of the solution, verified by the reviewers (15 points per reviewer assuming 2 reviewers per solution)
- ...and a maximum of 15 points *beyond* the total of 100 by comparing how much of the case extensions described in Section 5 has been implemented

Correctness and completeness - 60 points. By the final ranking of the solutions, there are three kinds of test cases considered: (i) the public ones, which are part of the test framework ARTE and have been also discussed in Sec. 3, (ii) the hidden ones, also being part of ARTE but being not further specified except for some hints within ARTE and (iii) some additional test cases which will not be announced until the final evaluation occurs. There is a fixed amount of points assigned to each test case; these numbers are not public, however, the developers may assume that the point distribution reflects the levels of difficulty. *The solution developer should provide*: a simple summary of the test cases accomplished by the solution. As the basic challenge offers fewer possibilities, we will give 2/3 of all points that can be achieved in this category.

Execution times - 10 points. The test framework ARTE provides an execution time measurement (per test case), whose result is then displayed on the console in the test summary. Based on the final test set, the fastest solution gets 10 points and the slowest 1 point, while the remaining ones will be distributed homogeneously on this scale.

Reviewer opinion - 2 x 15 points. Each of the reviewers has 15 points to award to the solution according to how much they like it.

To make the reviewers get a better insight into your solution beyond its objective correctness, it is generally a good idea to name some strong and some weak spots of the solution. It is definitely the developer itself who can contribute the most to this topic.

The soft aspects listed below serve as guidelines or hints for the solution developers to comment on their solution beyond the scope of the actual test cases in the contest. It is not mandatory, but we are excited to learn more about the way your tool works and what it can achieve!

- **Comprehensibility**: we think that the question if a solution works with an *understandable* mechanism which is not exclusively accessible for the high priests of a cult is of high importance, especially in the scope of the Transformation Tool Contest where such a comprehensible solution facilitates discussion and contributes to a profitable event.
- **Readability**: in contrast to comprehensibility, this aspect refers to the outer appearance of the tool - whether it has a nice and/or user-friendly interface, can be easily operated, maybe even with custom-tailored commands or a DSL, ...
- **Communication with the user**: although related to readability, this aspect refers to the quality, informativeness and level of detailedness of the actual messages given to the user while implementing a solution. In other words: Am I as user informed that everything went smoothly? In case of some failure or malfunction, am I thoroughly informed what actually went wrong?
- **Robustness**: this classical software quality aspect characterizes how a software behaves if put into an erroneous environment, getting malformed input,

... E.g., what happens if some out-of-scope keywords appear in a Java program to be refactored?
- **Extensibility**: this one also examines the inner structure of the solution concerning its possibilities to expand in the future. E.g., would it be easily feasible to build in the support of additional Java constructs or new refactorings?
- **Debugging**: in contrast to readability, this aspect refers solely to the debugging capabilities of the tool used to create the solution. In case a problem is uncovered through erroneous behavior, what means are provided to locate the cause of a design failure? Does the tool provide suggestions for fixing errors? How precise are the debug messages?

We have created a simple online form[3] for the reviewers to send in their opinion regarding the aspects above.

5 Case Extensions

While the core case described in Sections 1-4 is already a full-fledged refactoring use-case on its own right, it can still be extended in various ways inspired by the theory and practice of refactorings. Here, we mention some interesting possibilities for extending a solution beyond the requirements of the core case. New ideas are of course welcome and will be taken into account. There is a bonus of 15 points, which can be achieved by providing some (maybe partial) answers to one or more extensions or at least outlining a concept with relation to the core case solution. One convincing extension is enough to achieve the full bonus. These points are awarded according to the reviewers' opinion and we only give some recommendations which may serve as scoring guidelines. The final bonus score is calculated as the average of the reviewers' scores.

5.1 Extension 1: Extract Superclass

The two refactoring operations considered in the core case, namely `Pull Up Method` and `Create Superclass`, are simple actions compared to some complex operations which are still described as a single refactoring step in the literature. A classical example for such a more complex refactoring is `Extract Superclass`, which can be specified as a combination of `Create Superclass` and `Pull Up Method` (and its pendant for fields, `Pull Up Field`). After executing a `Create Superclass` for some classes, one can use `Pull Up Method` resp. `Pull Up Field` on the newly created parent class to move the common members there.

Recommendation. 15 points for a full-fledged implementation which can be executed on an appropriate example program; 9 points for a working implementation which misses `Pull Up Field`; 1-3 points for a concept sketch using the actual rule implementations.

[3] `http://goo.gl/forms/8VJuiD82Sg`

5.2 Extension 2: Propose Refactoring

In our core refactoring use-case, the hypothetical user already realized the need of refactoring and also identified the spot where it would be possible and the kind of action to be executed. Nevertheless, one can imagine a somewhat orthogonal approach where an automatic refactoring environment permanently monitors an evolving software and proactively proposes refactorings being feasible on the code base. To be more concrete, as a first step towards such a system, one might implement a method which takes as input the whole program and returns one (or more) feasible refactoring(s).

Recommendation. 15 points for a full-fledged implementation which can be executed on an appropriate example program; 5-15 points for an alternative way of implementation according to its scope and usability; 1-5 points for a plausible concept sketch according to its ambition and clarity.

5.3 Extension 3: Detecting Refactoring Conflicts

From a practical point of view, it is not unlikely that two developers of the same software might want to execute refactorings independently of each other. In this case, it can happen that the refactored code states are not compatible to each other any more and a merge is not possible. Concerning only two alternative refactorings, it is equivalent with stating that the result of the refactorings is not independent of their execution sequence. As a concrete step towards a conflict detection for refactorings, one can e.g. think of extending the framework so that it checks consequent refactoring operations and notifies the user if their execution sequence is considered as critical.

Recommendation. 15 points for a full-fledged implementation which can be executed on an appropriate pair of refactorings; 9-15 points for an alternative way of implementation according to its scope and usability; 1-9 points for a plausible concept sketch according to its ambition and clarity.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
2. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. Tech. rep. (1992)

A Appendix: Handbook of the Automated Refactoring Test Environment ARTE

ARTE is a Java terminal program which executes test cases specified in a Domain Specific Language (DSL) on solutions of the OO Refactoring Case Study of the Transformation Tool Contest 2015. A test case comprises a sequence of refactoring operations on a Java program as well as the expected results. The test cases are collected in a test suite in ARTE. The tests aim at checking the correct analysis of pre- and postconditions for refactorings and the execution of these refactorings.

For executing the provided test framework, Java JDK 1.7 is needed and the path variable has to be set to point to the JDK and not to a JRE. With a JRE and no JDK, the test framework will still start but the compilation of Java programs during testing will fail.

ARTE has been tested on Windows command line and in Bash. However, ARTE should be executable in every Java-capable terminal.

A.1 Case Study Solutions and ARTE

A solution for the case study has to implement an interface that specifies method signatures which ARTE relies on. This interface is called `TestInterface` and is provided in the file `TTCTestInterface.jar`.

Additionally, the solutions have to be exported as a simple (not executable) JAR file. This file has to contain a folder `META-INF/services` with a file called `ttc.testsuite.interfaces.TestInterface`. This latter file has to contain the fully qualified name of that class which implements the `TestInterface`. In the `TTCSolutionDummy` project, a dummy implementation of this interface is demonstrated.

An implementation fulfilling these conditions can be dynamically loaded into ARTE using the Java `ServiceLoader`. Further information can be found on the Oracle website [4][5].

The single methods which have to be implemented are:

getPluginName():
Returns the name of the actually loaded solution.

setPermanentStoragePath(File path):
Is called by ARTE to hand over a location at which data can be stored permanently by the solution.

setTmpPath(File path):
Is called by ARTE to hand over a location at which data can be stored temporally. All contents written to this location will be automatically deleted by closing ARTE.

[4] `http://www.oracle.com/technetwork/articles/javase/extensible-137159.html`

[5] `http://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html`

setLogPath(File path):
Is called by ARTE to hand over a location at which logs can be stored permanently. ARTE will store reports at this location as well.

usesProgramGraph():
Is called by ARTE to determine whether the solution uses the program graph of the extended case.

setProgramLocation(String path):
Is called in the basic case by ARTE to hand over the location of the Java program which will be refactored next.

createProgramGraph(String path):
Is called in the extended case by ARTE to instruct the solution to build the program graph for the Java program located at `path`.

applyPullUpMethod(Pull_Up_Refactoring refactoring):
Is called by ARTE for a `Pull Up Method` refactoring to be performed. The structure of the type `Pull_Up_Refactoring` is explained in the following DSL part – its fields have similar names as the corresponding keywords of the DSL. Note that `name` fields contain names of variables inside the DSL and not method or class names.

applyCreateSuperclass(Create_Superclass_Refactoring refactoring):
Is called by ARTE for a `Create Superclass` refactoring to be performed. For the type `Create_Superclass_Refactoring` holds the same as for the type `Pull_Up_Refactoring` above.

synchronizeChanges():
Is called by ARTE to instruct the loaded solution to synchronize the Java source code with the PG. This means that the changes made on the PG have to be propagated into the Java program.

A.2 Defining Test Cases

We provide a custom DSL to make the creation of new test cases more convenient. For developing test cases, we provide an Eclipse plug-in which supports syntax highlighting and basic validation of the test files. However, test files can be written using any text editor.

In the following, we show on a `Pull Up Method` and on an `Create Superclass` example how our DSL can be used to create test cases and how to perform tests using ARTE. We explain the commands within the examples in a practical, step-by-step fashion. For further information about commands not covered by these simple examples, refer to the in-line explanations and to Appendix C where a full command list is provided.

A.3 DSL Example - Pull Up Method

As `Pull Up Method` test case example, we recapitulate our Example 1 that has been used in Section 2 of the case description to motivate refactorings. The structure of the Java program used in this example is shown in Refactoring Scenario 1 and the corresponding source code in Appendix B. On this program, we are going to execute the refactoring `pum(ParentClass, method(String, int))`.

Test cases are wrapped in a `TestFile` environment that also defines the name of the test case. This name should to be identical with the name of the file containing this `TestFile` environment. If this is not the case, it is automatically renamed during import into the test framework, which can lead to failing imports with no obvious cause. The `TestFile` name has to be unique.

TestFile 1.1: PUM Example 1

```
1  TestFile public_pum_1 {
```

A `TestFile` contains everything needed for a test execution, namely classes, methods, refactorings and test cases. All elements used in the test cases have to be defined in the corresponding test file. Therefore, we define all classes we want to use in the test case.

In our example, the first class to be defined is `ParentClass`. To unambiguously identify the class in the program during test execution, the package containing the class has to be given as well. If the class is contained in the default package, the `package` parameter can be omitted.

```
2    class existing_parent {
3      package "example01"
4      name    "ParentClass"
5    }
```

As we have to check after the refactoring whether the pull-upped method is no longer contained in the child classes, those have to be defined as well.

```
6    class child1 {
7      package "example01"
8      name    "ChildClass1"
9    }
10
11   class child2 {
12     package "example01"
13     name    "ChildClass2"
14 }
```

Required primitive types and classes from libraries have to be also explicitly defined. In this example, we need these in the signature of the method to be pulled-up.

```
15  class String {
16    package "java.lang"
17    name    "String"
18  }
19
20  class int {
21    name "int"
22  }
```

For specifying a method signature, the name of the method and its parameters are necessary. The `params` command is optional as a method may have an empty parameter list. The order of the parameter list is important.

```
23  method child_method {
24    name   "method"
25    params String, int
26  }
```

According to Example 1, we are going to specify the `Pull Up Method` refactoring `pum(ParentClass, method(String,int)`. For this purpose, we have defined the necessary elements above and now, we combine them using the keyword `pullup_method`. (A refactoring can be used in multiple test cases within the test file.)

```
27  pullup_method executable_pum {
28    parent existing_parent
29    method child_method
30  }
```

Each test case has a mandatory description, which will be displayed during execution of the test case. As second argument, the name of a Java program is given. By having a look on the file structure shown in Source Code B, it can be seen that this program name refers to the folder containing the input program.

```
31  case pub_pum1_1_paper1 {
32    description "PUM-POS: (paper-ex1) Pull-up of two ..."
33    program "paper-example01"
34    testflow {
35      assertTrue(executable_pum)
36
37      existing_parent contains child_method
38      child1 ~contains child_method
39      child2 ~contains child_method
40    }
41  }
42 }
```

The single steps of a test case are defined in a list starting with the keyword `testflow`. A `testflow` environment automatically induces both before- and after-testing. As the previously defined refactoring is supposed to succeed on the given Java program, we assert a successful refactoring by using the `assertTrue` command. Refactorings can only be executed with an accompanying assertion.

After executing the refactoring, we check if the resulting Java program has the structure shown in Refactoring Scenario 1. Therefore we are checking if the method has been moved to the parent and if the child classes do not contain the method anymore.

A.4 DSL Example - Create Superclass

In the following, we describe a test case for a `Create Superclass` refactoring. For this purpose, we use again our example from the case description. The example is shown in Refactoring Scenario 4. The refactoring `csc({ChildClass1, ChildClass2}, Superclass)` is expected to succeed.

Again, we first define the necessary elements for the refactoring. The classes for which a new superclass will be created are enumerated in a list called `child` by using the `classes` keyword. One class can be added to multiple lists and lists can be used by multiple refactorings.

```
1  TestFile public_exs_1 {
2
3    class child1 {
4      package "example04"
5      name        "ChildClass1"
6    }
7
8    class child2 {
9      package "example04"
10     name        "ChildClass2"
11   }
12
13   classes child {child1, child2}
```

Elements defined in a `TestFile` do not have to exist in the input or output program. However, accessing these elements will result in a failure if they have not been created before by, e.g., a refactoring. Here, we define the variable `new_superclass` as a "placeholder" for the class `Superclass` which will be created by the `Create Superclass` refactoring.

```
14   class new_superclass {
15     package "example04"
16     name "Superclass"
17   }
```

For the definition of the `Create Superclass` refactoring, we are referencing the elements defined before.

```
18  create_superclass refactoring {
19    child childs
20    target new_superclass
21  }
22
23  case pub_exs1_1 {
24    description "EXS-POS: Create a superclass for two..."
25    program "example04"
26    testflow {
```

As we are expecting the refactoring to succeed, we use the `assertTrue` keyword. If we expect a refactoring to fail, we can use the keyword `assertFalse`. The additional keywords `expectTrue` and `expectFalse` can be used in ambiguous cases; these result in success if the expectation is fulfilled and in a warning instead of a failure otherwise. Additionally, these two keywords include an `else`-block where static tests on the unexpected outcome can be executed. For more details, refer to Appendix C.

```
27      assertTrue(refactoring)
```

The `step` keyword allows for grouping the different stages in a `testflow` but has no influence on the execution.

At this point, we have to check whether the child class extend the new superclass or not.

```
28      step{
29        child1 extends new_superclass
30        child2 extends new_superclass
31      }
32    }
33  }
34  }
```

It is possible to execute multiple refactorings in a single test case.

A.5 Using ARTE

On Windows, ARTE can be started by double-clicking `run_windows.bat`. On Linux, the file `run_linux.sh` has to be executed.

```
[foo@bar ARTE]$ sh run_linux.sh
```

If ARTE has been launched for the first time, a solution has to be loaded.

```
load --solution /home/foo/dummy.jar
```

The entered path has to be absolute. The referenced solution will be copied to the permanent storage path of ARTE. The same command has to be used again to load a different version of the solution. The previous loaded solution will be deleted.

Test cases can be loaded similarly. In contrast to the load solution command, multiple test cases can be imported.

```
load --test /home/public_pum_1.ttc /home/public_exs_1.ttc
```

The import of the Java programs is a bit more complex. In addition to the path where the program is located, the main class of the program has to be given as well. The Java programs have to be structured like the example shown in Source Code B. The referenced program folder has to contain a `src` folder. The package structure is represented by further subfolders containing classes. The referenced program folder is equivalent to an Eclipse project folder.

A Java program loaded into ARTE has to contain a class defining a `main` method that is executed during testing.

```
load --src /home/paper-example01 --main example01.ChildClass1
```

It is possible to print out each loaded test case and Java program.

```
testcases --list
```

```
programs --list
```

There are three ways to execute test cases:

1. Execute test cases by name. This is only possible for our public test cases and for self-written test cases. In this variant, multiple test cases can be chosen. If the name of a test file is entered, each test case in this file will be executed.

In the example below, all cases contained in the file `public_pum_1.ttc` and the test case `pub_exs1_1` will be executed.

```
execute --test public_pum_1.ttc pub_exs1_1
```

2. Execute all hidden test cases.

```
execute --hidden
```

3. Execute all public, hidden and self-written test cases.

```
execute --all
```

It is indispensable to use the `exit` command after using ARTE.

```
exit
```

Most of the presented commands can be executed in various ways. Feel free to find your favourite. The `help` command will help you with this. If you, e.g., want to know more about test cases, try the following:

```
help testcases
```

B Appendix: Source Code of the Java Program shown in Refactoring Scenario 1a

paper-example01/src/example01/ParentClass.java

```
1  package example01;
2
3  public class ParentClass {
4
5          public ParentClass(){}
6  }
```

paper-example01/src/example01/ChildClass1.java

```
7  package example01;
8
9  public class ChildClass1 extends ParentClass {
10
11         public ChildClass1(){}
12
13         public void method(String message, int repeat) {
14                 for(int i=0; i<repeat; i++){
15                         System.out.println(message);
16                 }
17         }
18
19         public static void main(String[] args){
20                 ChildClass1 c1 = new ChildClass1();
21                 c1.method("c1:Hello World", 3);
22
23                 ChildClass2 c2 = new ChildClass2();
24                 c2.method("c2:Hello World", 3);
25         }
26 }
```

paper-example01/src/example01/ChildClass2.java

```
27 package example01;
28
29 public class ChildClass2 extends ParentClass {
30
31         public ChildClass2(){}
32
33         public void method(String string, int k) {
34                 int i = 0;
35                 while(i++ < k) System.out.println(string);
36         }
37 }
```

C Appendix: Command Table of the DSL

Command	Subcommand	Description
TestFile *file_id* **{**		A test file always starts with this command. The file has to be called “file_id.ttc”.
	file_content	The content of a test file can be a combination of elements **class**, **classes**, **method**, **pullup_method**, **create_superclass** and **case**.
}		

Command	Subcommand	Description
class *class_id* **{**		The class command is used to describe a Java class.
	package [String]	An optional String value like ”subsubpackage.subpackage.package”.
	name [String]	The name of the class.
}		

Command	Subcommand	Description
classes *classes_id* **{**		The classes command is used to define sets of classes for further use.
	$class\_id_0, ..., class\_id_n$	A comma separated list of classes which should be grouped.
}		

Command	Subcommand	Description
method *method_id* **{**		The method command is used to describe a Java method signature.
	name [String]	The name of the method.
	param $class\_d_0, ..., class\_id_n$	Parameters are optional and are an ordered list of comma separated references to classes.
}		

Command	Subcommand	Description
pullup_method *refactoring_id* **{**		The definition of a `Pull Up Method` refactoring.
	parent *class_id*	A reference to the parent class whose childs a method should be pulled up from.
	method *method_id*	A reference to the method which should be pulled up.
{		

Command	Subcommand	Description
create_superclass *refactoring_id* **{**		The definition of a `Create Superclass` refactoring.
	classes *classes_id*	A reference to the set of classes for which a superclass should be created.
	target *class_id*	A reference to a class variable describing the superclass which will be created.
}		

Command	Subcommand	Description
case *test_case_id* **{**		A test case can be identified by the test suite through *test_case_id*.
	description [String]	A textual description of the test case. This description is also shown in the test tool.
	program [String]	The name of the program on which the test case should operate.
	testflow {	A container for the test commands.
	$test\_step_0, ..., test\_step_n$	An ordered list of test commands that can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, **∼contains**, **extends**, **∼extends**, **synchronize** and **compile**.
	}	
}		

Command	Subcommand	Description
step {		Allows for grouping, has no effect on the execution.
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps which can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
}		

Command	Subcommand	Description
assertTrue(**assertFalse(**		Checks whether a refactoring has been executed successful. The result is compared with the assertion.
	refactoring_id	The refactoring which will be handed to the solution for execution.
)		

Command	Subcommand	Description
expectTrue(**expectFalse(**		Checks whether a refactoring has been executed successful. The result is compared with the expected result. If the expected result is not matched, the execution can still be successful.
	refactoring_id	The refactoring which will be handed to the solution for execution.
){		
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps executed if the expectation has been matched. The test steps can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
} else {		
	warning [String]	A message displayed if the else block has been entered.
	$test\_step_0, ..., test\_step_n$	An ordered list of test steps executed if the expectation has not been matched. The test steps can contain **step**, **assertTrue()**, **assertFalse()**, **expectTrue()**, **expectFalse()**, **contains**, ∼**contains**, **extends**, ∼**extends**, **synchronize** and **compile**.
}		

LHS Variable	Command	RHS Variable	Description
class_id	**contains**	*method_id* *field_id*	Checks if the method or field (RHS) is contained in the class (LHS). The test case fails if the method or field is not contained in the class.

LHS Variable	Command	RHS Variable	Description
class_id	∼**contains**	*method_id* *field_id*	Checks if the method or field (RHS) is not contained in the class (LHS). The test case fails if the method or field is contained in the class.

LHS Variable	Command	RHS Variable	Description
class_id	**extends**	*class_id*	Checks whether the LHS class extends the RHS class. The test case fails if LHS does not extend RHS.

LHS Variable	Command	RHS Variable	Description
class_id	∼**extends**	*class_id*	Checks whether the LHS class does not extend the RHS class. The test case fails if LHS extends RHS.

Command	Description
synchronize	Triggers the propagation of changes made on the program graph to the Java source code.
compile	Triggers the compilation of the Java source code.

Solving the TTC Java Refactoring Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

This paper describes the FunnyQT solution to the TTC 2015 Java Refactoring transformation case. The solution solves all core tasks and also the extension tasks 1 and 2, and it has been elected as overall winner of this case.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Java Refactoring Case [3]. It solves all core and exception tasks with the exception of *Extension 3: Detecting Refactoring Conflicts* and has been elected as overall winner of the case. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3].

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are Clojure programs using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities[5] that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSLs) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF models and JGraLab TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each one providing constructs supporting a concrete use-cases, e.g., model management, visualization, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and co-evolution transformations. For solving this case, FunnyQT's out-place and in-place transformation DSLs have been used.

2 Solution Description

The solution consists of three steps. (1) Converting the Java code to a program graph, (2) refactoring the program graph, and (3) propagating changes in the program graph back to the Java code. These steps are discussed in the following sections.

2.1 Step 1: Java Code to Program Graph

The first step in the transformation chain is to create an instance model conforming to the program graph metamodel predefined in the case description from the Java source code that should be subject to refactoring. The FunnyQT solution does that in two substeps.

[1] `http://funnyqt.org`
[2] `https://github.com/tsdh/ttc15-java-refactoring-funnyqt`
[3] The SHARE image name is `ArchLinux64_TTC15-FunnyQT_2`
[4] `http://clojure.org`
[5] The abstract syntax of a program can be accessed as data and manipulated at compile-time.

(a) Parse the Java source code into a model conforming to the EMFText JaMoPP[6] metamodel.
(b) Transform the JaMoPP model to a program graph using a FunnyQT out-place transformation.

Step (a) is implemented in the solution namespace *ttc15-java-refactoring-funnyqt.jamopp*. It simply sets up JaMoPP and defines two functions, one for parsing a source tree to a JaMoPP model, and a second one to synchronize the changes in a JaMoPP model back to the source tree. Both just access JaMoPP built-in functionality. Being able to seamlessly interoperate with Java is a feature FunnyQT gets for free from its host language Clojure.

Step (b) is implemented as a FunnyQT out-place transformation which creates a program graph from the parsed JaMoPP model.

The transformation also tries to keep the target program graph minimal. The source JaMoPP model contains the complete syntax graph of the parsed Java sources including all their dependencies. In contrast, the program graph created by the transformation only contains TClass elements for the Java classes parsed from source code and direct dependencies used as field type or method parameter or method return type. TMember elements are only created for the methods of directly parsed Java classes, and then only for those members that are not static because the case description explicitly excludes those. As a result, the program graph contains only the information relevant to the refactorings and is reasonably small so that it can be visualized by FunnyQT which is helpful for debugging purposes.

The FunnyQT out-place transformation API used for implementing this task is quite similar to ATL or QVT Operational Mappings. There are mapping rules which receive one or many JaMoPP source elements and create one or many target program graph elements.

A cutout of the transformation showing the rules responsible for transforming fields is given below. The transformation receives one single source model `jamopp` and one single target model `pg`.

```
(deftransformation jamopp2pg [[jamopp] [pg]]
  ...
  (field2tfielddef
   :from [f 'Field]
   :when (not (static? f))
   :to   [tfd 'TFieldDefinition {:signature (get-tfieldsig f)}])
  (get-tfieldsig
   :from [f 'Field]
   :id   [sig (str (type-name (get-type f)) " " (j/name f))]
   :to   [tfs 'TFieldSignature {:field (get-tfield f)
                                :type  (type2tclass (get-type f))}])
  (get-tfield
   :from [f 'Field]
   :id   [n (j/name f)]
   :to   [tf 'TField {:tName n}]
   (pg/->add-fields! *tg* tf))
  (type2tclass
   :from [t 'Type]
   :disjuncts [class2tclass primitive2tclass])
  ...)
```

For each non-static field in the JaMoPP model, the `field2tfielddef` rule creates one TFieldDefinition element in the program graph. The signature of this TFieldDefinition is set to the result of calling the `get-tfieldsig` rule.

This rule uses the `:id` feature to implement a n:1 semantics. Only for each unique string `sig` created by concatenating the field's type and name, a new TFieldSignature is created. If the rule is called thereafter for some other field with the same type and name, the existing field signature created at the first call is returned. The field signature's field and type references pointing to a TField and a TClass respectively are set by calling the two other rules `get-tfield` and `type2tclass`. This latter rule is a disjunctive rule which delegates to either the `class2tclass` or the `primitive2tclass` rule[7].

[6] `http://www.jamopp.org/index.php/JaMoPP`
[7] Rule disjunction is a feature borrowed from QVTo

In total, the transformation consists of 10 rules summing up to 71 lines of code. In addition, there are five simple helper functions like `static?`, `get-type`, and `type-name` that have been used in the above rules already.

A FunnyQT out-place transformation like the one briefly discussed above returns a map of traceability information. This traceability map is used in step 3 of the overall procedure, i.e., the back-propagation of changes in the program graph to the Java source code.

2.2 Step 2: Refactoring of the Program Graph

The refactorings are implemented in the solution namespace *ttc15-java-refactoring-funnyqt.refactor* using FunnyQT in-place transformation rules which combine patterns to be matched in the model with actions to be applied to the matched elements.

All rules defined in the following have a parameter `pg2jamopp-map-atom` which is essentially the inverse of the traceability map created by the JaMoPP to program graph transformation from step 1, i.e., it allows to translate program graph TClass and TMember elements to the corresponding JaMoPP Class and Member elements.

Pull Up Member. The case description requests *pull-up method* as the first refactoring core task. However, with respect to the program graph metamodel, there is actually no difference in pulling up a method (TMethodDefinition) or a field (TFieldDefinition), i.e., it is possible to define the refactoring more generally as *pull-up member* (TMember) and have it work for both fields and methods. This is what the FunnyQT solution does.

The corresponding `pull-up-member` rule is shown in the next listing. The rule is overloaded on arity. There is the version (1) of arity three which receives the program graph `pg`, the inverse lookup map `pg2jamopp-map-atom`, and the JaMoPP resource set `jamopp`, and there is the version (2) of arity four which receives the program graph `pg`, the inverse lookup map atom `pg2jamopp-map-atom`, a TClass `super`, and a TSignature `sig`.

```
(defrule pull-up-member
  ([pg pg2jamopp-map-atom jamopp]                                          ;; (1)
   [:extends [(pull-up-member 1)]]                                         ;; pattern
   ((do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)  ;; action
    jamopp))
  ([pg pg2jamopp-map-atom super sig]                                       ;; (2)
   [super<TClass> -<:childClasses>-> sub -<:signature>-> sig               ;; pattern
    sub -<:defines>-> member<TMember> -<:signature>-> sig
    :nested [others [super -<:childClasses>-> osub
                     :when (not= sub osub)
                     osub -<:signature>-> sig
                     osub -<:defines>-> omember<TMember> -<:signature>-> sig]]
    :when (seq others)                                                     ;; (a)
    super -!<:signature>-> sig                                             ;; (b)
    :when (= (count (pg/->childClasses super)) (inc (count others)))       ;; (c)
    :when (forall? (partial accessible-from? super)                        ;; (d)
                   (mapcat pg/->access (conj (map :omember others) member)))]
   (do-pull-up-member! pg pg2jamopp-map-atom super sub member sig others)))  ;; action
```

The version (2) is the one which is called by the ARTE test framework whereas the first version is called when performing the interactive refactoring extension.

The pattern of the version (2) matches a subclass `sub` of class `super` where `sub` defines a `member` of the given signature `sig`. A nested pattern is used to match all other subclasses of `super` which also define a member with that signature. The constraint (a) ensures that there are in fact other subclasses declaring a member with signature `sig`. Then the negative application condition (b) defines that the superclass `super` must not define a member of the given `sig` already. The constraint (c) ensures that all subclasses define a member of the given `sig`, i.e., not only a subset of all subclasses do so. Lastly, the constraint

(d) makes sure that all field and method definitions accessed by the member to be pulled up are already accessible from the superclass[8].

The pattern of the arity three variant (1) of the `pull-up-member` rule contains just an `:extends` clause specifying that its pattern equals the pattern defined for the arity four variant. As said, this variant is used by the extension task 2 where possible refactorings are to be proposed to the user. The difference between the overloaded versions of the `pull-up-member` rule is that version (1) matches `super` and `sig` itself whereas these two elements are parameters provided by the caller (i.e., ARTE) in version (2).

When a match is found, both versions of the rule call the function `do-pull-up-member!` which is defined as follows.

```
39 (defn do-pull-up-member! [pg pg2jamopp-map-atom super sub member sig others]
40   (doseq [o others]                                              ;; PG modification
41     (doseq [acc (find-accessors pg (:omember o))]
42       (pg/->remove-access! acc (:omember o))
43       (pg/->add-access! acc member))
44     (edelete! (:omember o))
45     (pg/->remove-signature! (:osub o) sig))
46   (pg/->remove-signature! sub sig)
47   (pg/->add-defines! super member)
48   (pg/->add-signature! super sig)
49   (fn [_]                                                        ;; JaMoPP modification
50     (doseq [o others]
51       (edelete! (@pg2jamopp-map-atom (:omember o)))
52       (swap! pg2jamopp-map-atom dissoc (:omember o)))
53     (j/->add-members! (@pg2jamopp-map-atom super) (@pg2jamopp-map-atom member))))

54 (defn find-accessors [pg tmember]
55   (filter #(member? tmember (pg/->access %))
56           (pg/all-TMembers pg)))
```

It first applies the changes to the program graph by deleting all duplicate member definitions from all other subclasses of `super` and pulling up the selected member into `super`. It also updates all accessors of the old members in order to have them access the single pulled up member. Lastly, it returns a closure which performs the equivalent changes in the JaMoPP model and updates the reference to the inverse lookup map when being called.

A function encapsulating the changes is returned here instead of simply applying the changes also to the JaMoPP model because the ARTE TestInterface defines that the back-propagation of changes happens at a different point in time than the refactoring of the program graph. Thus, the solution's TestInterface implementation simply collects the closures returned by appling the rules in a collection and invokes them in its `synchronizeChanges()` implementation.

Note that the rule's variant (1) immediately invokes the function returned by `do-pull-up-member!`. This is because this variant is not called by ARTE but is intended for extension task 2, and with that there is no need to defer back-propagation.

The rule `create-superclass` implementing the other core task is defined analogously, and the extension task 1 rule `extract-superclass` simply combines `create-superclass` with `pull-up-member`.

FunnyQT provides built-in functionality to let users steer rule application, i.e., choose an applicable rule and one of its matches and then apply the rule to that match. This feature is used for solving the second extension task of proposing refactorings to the user.

2.3 Step 3: Program Graph to Java Code

The core `pull-up-member` and `create-superclass` rules return closures which perform the refactoring's actions in the JaMoPP model when ARTE calls the `TestInterface`'s `synchronizeChanges()` method.

[8] The `accessible-from?` predicate has been skipped for brevity.

Then, the JaMoPP model needs to be saved to reflect those changes also in the Java source code files. This is done by the `synchronizeChanges()` method of the solution's `TestInterface` implementation.

```
public boolean synchronizeChanges() {
    try {
        for (IFn synchronizer : synchronizeFns) { synchronizer.invoke(jamoppRS); }
        SAVE_JAVA_RESOURCE_SET.invoke(jamoppRS);
        return true;
    } catch (Exception e) { return false; }
      finally { synchronizeFns.clear(); }
}
```

`synchronizedFns` is the list of closures returned by the rules which simply get invoked and perform the same changes to the JaMoPP model which have previously been applied to the program graph. Thereafter, the JaMoPP resource set is saved which means that the source code files are updated accordingly.

3 Evaluation & Conclusion

In this section, the FunnyQT solution is evaluated according to the criteria suggested in the case description which was also used as the basis for the open peer review.

The FunnyQT solution is *correct*, i.e., all tests performed by ARTE pass, and it implements all core tasks. Thus, it is also *complete* and received a full score for the correctness and completeness criterium.

According to ARTE, the FunnyQT solution runs in less than a tenth of a second for all test cases on an off-the-shelf laptop so the *performance* seems to be good. Nevertheless, the benchmarking performed by the case authors suggested that all other solutions except for NMF perform even better. However, all the ARTE test cases are actually too small to provide meaningful numbers. And in any case, the execution time of the actual refactorings on the program graph and the back-propagation into the JaMoPP model are completely negligible when being compared to the time JaMoPP needs to parse the Java sources, resolve references in the created model, and serialize the model back to Java again.

Another strong point of the solution is its *conciseness*. It consists of only 271 NCLOC of FunnyQT code for all core and the two solved extension tasks and 145 NCLOC of Java code for the `TestInterface` implementation class required by ARTE.

The FunnyQT solution also received a high *extension score* because it provides runnable implementations for the extensions 1 (*extract superclass*) and 2 (*propose refactoring*).

A *main critique* of the solution and FunnyQT in general is that many developers used to languages with C-like syntax such as Java dislike FunnyQT's Lisp-syntax. Additionally, its functional emphasis where transformations and rules are essentially functions which might get composed and passed to higher-order functions requires a shift from the object-oriented to the functional paradigm. Although this provides several benefits it also requires more learning effort and might hinder the adoption of FunnyQT.

Nevertheless, the FunnyQT solution received a reasonably good reviewer score which paired with its correctness and completeness resulted in letting it carry off the overall winner award for this case.

References

[1] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[2] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015.*

[3] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation*. In: *Transformation Tool Contest 2015.*

TTC'2015 Case: Refactoring Java Programs using Spoon

Gérard Paligot
gerard.paligot@inria.fr
Inria

Nicolas Petitprez
nicolas.petitprez@inria.fr
Inria

Martin Monperrus
martin.monperrus@univ-lille1.fr
University of Lille

Abstract

TTC'2015 is the 8th Transformation Tool Contest for users and developers of transformation tools. In this paper, we present the use of Spoon, an open-source library to transform and analyze Java source code for the code refactoring track of TTC'2015. We use Spoon to implement *pull-up-method* and *create super-class* refactorings. The implementation uses an unmodified revision of Spoon and is done in 125 lines.

1 Introduction

Spoon[7] is an open-source library that enables you to transform and analyze Java source code. Spoon provides a complete and fine-grained Java metamodel where any program element (classes, methods, fields, statements, expressions...) can be accessed both for reading and modification. Spoon takes as input source code and produces transformed source code ready to be compiled.

For now, Spoon has been used in many different contexts: program analysis and transformation in Java[6], automatic repair of buggy if conditions[4] or fault injection [3] but nobody has ever studied the use of Spoon for refactoring Java programs. To explore this new usage of Spoon, we have implemented the two types of refactoring asked by Transformation Tool Contest 2015 [2].

> Our solution is publicly available on Github:
> `https://github.com/GerardPaligot/ttc-competition/`

The paper reads as follows. Section 2 gives a description of the chosen case study. Section 3 present the chosen solution. Section 4 explains how we validate our solution. Section 5 give some discussions of design decisions and perspectives for our work. Section 6 concludes this paper.

2 Background

2.1 Case study

The chosen case study is "Object-oriented Refactoring of Java Programs using Graph Transformation" [5], it proposes two object-oriented program refactorings. It consists of implementing two refactorings, namely *pull-up-method* and *create super-class*.

First, we explain how *pull-up-method* works. Before the refactoring, the Java code must have methods with identical signatures (name and parameters) and equivalent behaviors. These methods are then moved to the superclass. After the refactoring, the method is a member of the superclass and deleted from the subclasses. This operation is depicted in Figure 1.

We consider the following conditions to apply the *pull-up-method* refactoring:

1. Each child class of class `ParentClass` has at least one common method signature with the corresponding method definitions having equivalent functionality [5].

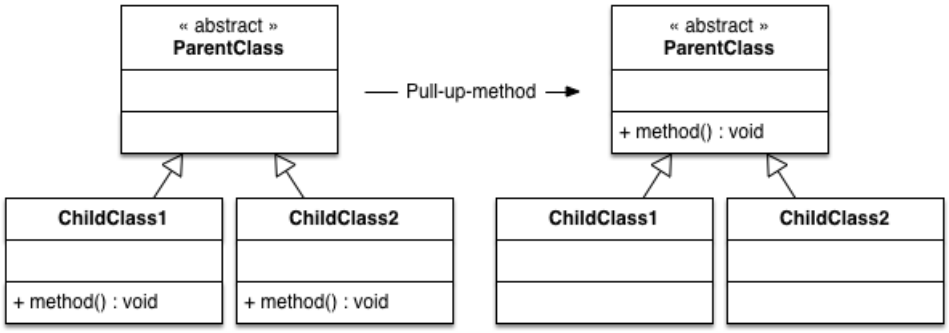


Figure 1: Illustration of a Pull-up-method

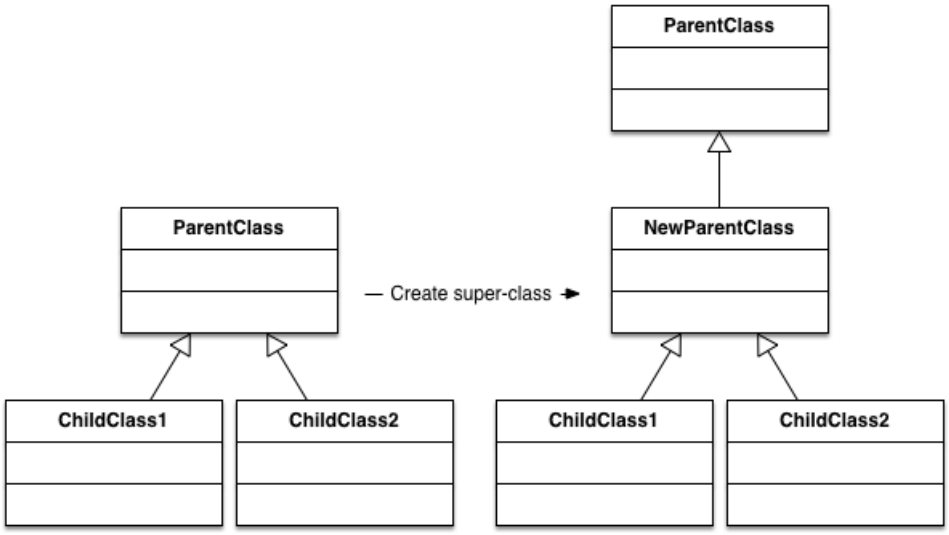


Figure 2: Illustration of a Create a super class

2. Each method in the child classes only accesses methods and fields accessible from `ParentClass`.

3. The `ParentClass` does not belong to a library and is editable. [5]

Second, we explains how the *create super-class* refactoring works. This kind of refactoring is useful when we have a set of classes with similar features. As a first step towards an improved program structure, a new common superclass of these classes is created. When we have subclasses with a parent class, it creates a new parent class and this new class extend the old one (the previous parent). This operation is depicted in the Figure 2.

In this case, we have one precondition: the classes are extending the same superclass. The precondition is always met is the default case since classes with no explicit inheritance in Java are all implementing `java.lang.Object`. [5]

This refactoring has the following post-conditions:

1. Each class has an inheritance to the new super class [5].

2. When the classes had an explicit inheritance relation to a superclass before the refactoring, their new superclass has an inheritance reference to the old super class [5].

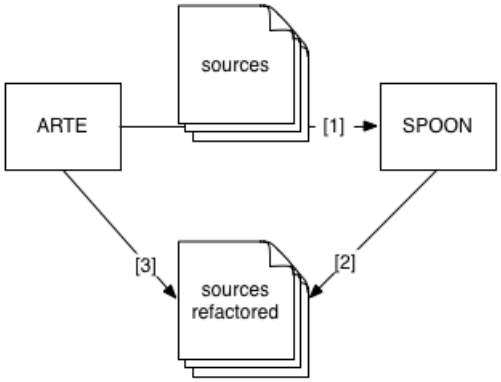


Figure 3: Transformation chain of the study case

2.2 Spoon

Spoon provides a Java abstract syntax tree (AST) designed to be understandable by developers. With this AST, developers can analyze or transform the source code. In our solution, we use two concepts of Spoon: `Factory` and `Query`.

`Factory` creates new elements or retrieves some specific elements.For instance, the factory is used to create the new super class for the *create super-class* refactoring.

`Query` makes complex queries on a AST. If you would like to retrieve all methods, fields, class or any elements of the meta-model, you execute a query to get them. This concept is used to retrieve all methods concerned by the *pull-up-method* refactoring.

2.3 Test infrastructure

ARTE is a Java program which executes test cases specified in a Domain Specific Language (DSL) for validating the solutions of the OO Refactoring Case Study of the Transformation Tool Contest 2015. A test case comprises a sequence of refactoring operations on a Java program as well as the expected results. The tests aim at checking the correct analysis of pre- and postconditions for refactorings and the execution of these refactorings [5].

This test framework defines specific command line arguments. When you execute the jar file, you launch a custom terminal where you execute these commands. From this terminal, you load your solution and execute the ARTE tests. If you execute all tests, it executes public tests and hidden tests. Input source code and assertions are public for public tests but only input sources are public for the hidden tests.

We give an overview of the transformation chain in Figure 3. First, we see than ARTE loads sources and gives them to our solution based on Spoon. Second, our implementation refactors the Fava source code given to print sources refactored. Third, ARTE uses sources refactored by our solution to check assertions on our results.

3 Presentation of the Solution

3.1 Pull-up-method

Algorithm 1 shows the pseudo-code of our implementation. In input, we have the method to be refactored and the superclass element where we should put the refactor method. These objects are given as parameter of the refactoring method. As output, a boolean tells whether the refactoring is done or not. Let's explain this algorithm step by step:

1. We check that the superclass given as parameter exists. If it doesn't exist, we are not allowed to pull up the method. The refactoring fails.

2. We retrieve all methods candidates for the refactoring with the same name and type parameters and we store them in a list named `candidates`.

3. For each candidate, we check that the superclass of the declaring class of the current candidate method exists. If this superclass doesn't exist, the refactoring fail.

4. For each candidate, we check that the body of the current candidate method does not try to access fields of the declaring class. If it tries, the refactoring fails.

5. For each candidate, we check that the body of the current candidate method does not try to access methods of the declaring class. If it tries, the refactoring fails.

6. We retrieve all subclasses of the superclass and for each subclass, we check that the method exists in it. If not, the refactoring fails.

7. When all previous conditions are passed, we apply the refactoring. For each candidate, we remove it from the declaring class and we set the method asked in the superclass.

Data: *superclass* element and *method* element
Result: true if the refactoring is done
if *superclass doesn't exist* **then**
  fail
end
candidates ← all methods candidates for refactoring;
foreach *candidate in candidates* **do**
  if *superclass of candidate doesn't exist* **then**
    fail
  end
  if *body of candidate try to access fields of declaring class* **then**
    fail
  end
  if *body of candidate try to access methods of declaring class* **then**
    fail
  end
end
foreach *subclass in subClasses of superclass* **do**
  if *method to refactor isn't present in subclass* **then**
    fail
  end
end
foreach *candidate in candidates* **do**
  Removes *candidate* method from declaring class;
end
Adds method in *superclass*;

Algorithm 1: Pull up methods in their superclass

3.2 Create super-class

Algorithm 2 shows the pseudo-code of our implementation of *create super-class*. As input, we have a set of children and a superclass element. These objects are given as parameter of the refactoring method. As output, a boolean tells whether the refactoring is done or not. Let's explain this algorithm step by step:

1. We check that the superclass does not already exist. If yes, the refactoring fails because we are not allowed to create a superclass on an existing class.

2. We create the new superclass from the superclass.

pub pum2 1	0,063 seconds
pub pum1 1 paper1	0,018 seconds
pub pum1 2	0,002 seconds
pub csc1 1	0,136 seconds
pub csc1 2	0,002 seconds
pub pum3 1	0,005 seconds
hidden csc1 1	0,003 seconds
hidden csc1 2	0,002 seconds
hidden pum1 1	0,003 seconds
hidden pum1 2	0,003 seconds
hidden csc2 1	0,003 seconds
hidden pum2 1	0,005 seconds
hidden pum2 2	0,003 seconds
hidden csc3 1a	0,009 seconds
hidden csc3 1	0,004 seconds

Table 1: Execution time measurements

3. We collect all super-classes of children and we check that there are all the same superclass. If yes, new superclass extends this superclass. Otherwise, the refactoring fails.

4. For each child in set of children, we set its superclass with the new superclass.

Data: set of *children* and *superclass* element
Result: true if the refactoring is done
if *superclass already exists* **then**
  fail
end
Create *newsuperclass* from *superclass*;
Set superclass of *newsuperclass* from superclasses of *children*;
foreach *child in children* **do**
  Set superclass of *child* with *newsuperclass*;
end

Algorithm 2: Creates and sets the new superclass for all children

---

3.3 Execution time measurements

When we execute a test in ARTE, we see in output the name of the test case, the executed refactoring, results of assertions and the execution time. Table 1 shows execution time measured by ARTE when we execute all tests (execution time measurements are different when we execute one by one).

We see that the performances are good. The worst execution time is the test case pub csc1 1. This test case applies the refactoring *create super-class* on an example with two child class and a super class for these subclasses.

3.4 Architecture

Our solution is available on Github [1]. It is a Maven project in Java 8 with only one "compile" dependency: spoon. The solution has 2 "provided" dependencies: TTCTestInterface and EMF. TTCTestInterface is a Jar file given by the case study and versioned in the project. We have created a local maven repository in the project to save all versions of TTCTestInterface jar file updated by organizers. TTCTestInterface contains an interface which returns objects with EMF objects, like `EList`. To manipulate objects like `EList`, we need the EMF dependency. Finally, there are 2 "test" dependencies: junit and mockito which are used to test our solution.

We generate the solution in a jar file with the maven command:

```
$ mvn clean assembly:assembly
```

This command compiles the project, launches all Junit test cases and builds the final jar file with dependencies in the target directory of the project. After that, this jar file is used on ARTE to launch all tests of this last tool. According to our experience, it isn't possible to integrate ARTE in the maven process because ARTE must be launched as command line.

All Junit tests are available in the folder `src/test/java` and correspond to public and hidden test cases given by organizers executed in ARTE. All Java source code used by ARTE has been copied in `src/test/java/resources` and used by test cases in `src/test/java/fr/inria`. So, when we compile the project with the command line given in the next section, we execute the same tests than the tests executed in ARTE.

The implementation asked of TTCTestInterface is SpoonTtc. This class retrieves the Java source code in the method `createProgramGraph` and builds the Spoon AST. This AST is used on methods to apply refactorings with 2 stages: First, refactoring methods check whether we must apply the refactoring. Second, refactoring methods apply the refactoring on the Spoon AST. The Java source code refactored is printed in the method `synchronizeChanges` in the source directory of the original Java program.

4 Validation

For *pull-up-method*, it has 9 tests corresponding to the ARTE public. There is on parameterized test class to launch 6 tests on all examples available in resources. For *create super-class*, it has 6 tests corresponding to the ARTE public and hidden tests and has a test class parameterized to launch 6 tests on examples.

For each test case, we make some assertions on the Spoon AST and the boolean result of the refactoring method. We make pre-conditions to know if the Spoon AST is in a correct state. We check that the refactoring method returns the expected boolean value. Finally, we make post-condition on the Spoon AST to know if the refactoring is applied or not, according to the boolean result of the refactoring method.

When we call refactoring methods, its parameters has a dependency to EMF. So, we add the mockito dependency to simulate these objects and test our implementation in a controlled environment. We build mocked objects and we add them in parameter of refactoring methods. For example, Listing 1 shows a mocked object given at the method `applyPullUpMethod`.

Listing 1 Test case for the pull-up-method

```
@Test
public void testPullUpMethod11() throws Exception {
  spoonTtc.createProgramGraph("./src/test/resources/paper-example01/");
  // Pre-asserts on the Spoon AST.
  assertTrue(spoonTtc.applyPullUpMethod(
    getPullUpRefactoringMocked(
      "example01.ParentClass", "foo", "java.lang.String", "int")));
  // Post-asserts on the Spoon AST.
}
```

This example calls the method `applyPullUpMethod` to apply the refactoring of the same name. As parameter, we give the mocked object. To build this last object, we give the super class where the method will be pulled up and the method concerned by the refactoring with type of its parameters. In this case, the refactoring is possible so we check than the result is `true`.

5 Discussions

To our opinion, Spoon was well suited for this case study. Its understandable AST and its capability to transform Java programs were appropriate. We realized the refactorings quickly and within a few lines. Implementing the refactoring case study with Spoon took 80 lines for *pull-up-method* and 21 lines for *create super-class*.

Spoon has no module to refactor Java source code. This case study was a great opportunity for us to start such a module. All contributions here will be integrated in Spoon in a next release and will be improved with some new refactorings in the future.

6 Conclusion

We have presented a solution for the for this edition of Transformation Tool Contest based on Spoon. It has validated the idea of using Spoon for implementing refactorings of Java source code.

References

[1] GitHub repository of our submission. `https://github.com/GerardPaligot/ttc-competition/tree/master/lib/fr/inria/TTCTestInterface/`.

[2] Transformation Tool Contest 2015. `http://www.transformation-tool-contest.eu/`.

[3] Benoit Cornu, Lionel Seinturier, and Martin Monperrus. Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions. *Information and Software Technology*, 57(0):66 – 76, 2015.

[4] Favio Demarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *CSTVA'2014*, Hyderabad, India, 2014.

[5] Sven Peldszus Géza Kulcsàr and Malte Lochau. Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation. 2015.

[6] Renaud Pawlak, Carlos Noguera, and Nicolas Petitprez. Spoon: Program Analysis and Transformation in Java. Research Report RR-5901, Inria, 2006.

[7] Spoon. Spoon project on GitHub. `https://github.com/INRIA/spoon`.

An NMF solution to the Java Refactoring Case at the TTC 2015

Georg Hinkel
Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14
Karlsruhe, Germany
hinkel@fzi.de

Lehman's laws state that dedicated efforts must be spent for any software artifact to prevent a loss of quality. For code, such efforts are called refactoring operations and are an important aspect of many software engineers day-to-day business. Many of these refactoring operations are specified on a much higher abstraction level than the actual source code of a given language like Java. To be able to specify these refactoring operations on a higher abstraction level as proposed in the Java Refactoring Case at the Transformation Tool Contest (TTC) 2015, we propose a solution using an incremental synchronization with NMF Synchronizations of the source code regarded as a model on the one side and a simplified program graph model on the other side.

1 Introduction

This paper proposes a solution for the Java Refactoring Case[1] at the Transformation Tool Contest (TTC) 2015. Our solution is publicly available on CodePlex[2] and SHARE[3] and built upon the .NET Modeling Framework[4] (NMF), especially on NMF Synchronizations [1].

All of the technologies used in this solution are implemented as internal languages hosted by C#. The reason for this is that we try to let developers stay with the language that they are most confident with as much as possible as recent research suggests that they will hardly change them voluntarily [2]. One of the reasons for this especially in an MDE context is that many transformation languages lack the tool support offered by mainstream languages such as Java or C# [3], [4].

As our solution is based on internal languages, we do not have this problem and thus, our solution is entirely specified in C# (besides JaMoPP). However, we were facing issues converting the JaMoPP models back to Java source files, which we were unable to resolve. As a consequence, the solution only creates the refactored JaMoPP models, but not the Java files. The solution is presented in detail in the next section.

2 Solution with NMF Synchronizations

We use JaMoPP [5] to translate Java files into a model representation. Since the NMF meta-metamodel NMeta is compatible with Ecore, we can easily transform the JaMoPP metamodel to an NMeta metamodel and consume the JaMoPP generated XMI representations of the input files directly. In between, we use the model synchronization language of NMF to refactor the Java files.

[1] https://github.com/Echtzeitsysteme/java-refactoring-ttc/raw/master/Case_Description-final.pdf
[2] http://ttc2015javarefactoringnmf.codeplex.com/
[3] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64-TTC15_NMF.vdi
[4] http://nmf.codeplex.com

NMF Synchronizations is a bridge between the model transformation language NMF Transformations [6], [7] and NMF Expressions[5], responsible for the incremental evaluation of arbitrary expressions. NMF Synchronizations uses NMF Expressions to make model transformations bidirectional and incremental, i.e. any changes of either left hand side (LHS) or right hand side (RHS) of the model can be propagated to the other. This change propagation is optional and can be chosen by the developer when running the transformation. In total, we support 18 different modes of operation, namely six directions and three different modes of change propagation (none, one way or two way). Details can be found in prior work [1].

Although NMF Synchronizations offers support for bidirectional model transformations, we do not use this feature. The reason is that the model transformation task of the proposed case conflicts with our definition of a model transformation being basically a function from one metamodel to another. Therefore, we classify the task of the present case study rather as a model synchronization task. We transform the Java model (we use JaMoPP) to a Program Graph model, modify this Program Graph model and propagate these changes back to the original JaMoPP model.

Currently, NMF Synchronizations only supports online synchronization. This means, any changes in the Program Graph model are immediately reflected in the JaMoPP model. In particular, the model synchronization adds hooks into the Program Graph model and reacts on changes in that it applies these changes to the JaMoPP model. As a consequence, the backward transformation from the case description and the refactoring operation on the PG get merged.

However, we do not support a one-way change propagation mode in the opposite direction of the transformation, and so we have selected the two way change propagation mode. That is, any changes in either of the JaMoPP model or the Program Graph model will be reflected in the other model.

We are aware that this causes some overhead when the Program Graph model is used only for a one-time refactoring, and we will add a change propagation mode one way to source in the future. Originally, when NMF Synchronizations was designed, we could not think of a useful application for this change propagation mode, but the present case study offers a good one.

2.1 Synchronization of JaMoPP and PG

Model synchronizations in NMF Synchronizations are classes and the synchronization rules are represented by public non-abstract nested classes. Listing 1 shows an excerpt of the model synchronization that synchronizes classes.

```
1  class JavaPGSynchronization : ReflectiveSynchronization {
2     public class Class2Class : SynchronizationRule<IClass, ITClass> {
3        public override void DeclareSynchronization() {
4           Synchronize(cl => cl.Name, cl => cl.TName);
5           SynchronizeMany(SyncRule<Member2Member>(),
6              cl => cl.Members.Where(m => m is ClassMethod || m is Field),
7              cl => cl.Defines);
8           Synchronize(this,
9              cl => cl.Extends as IClassifierReference != null
10                ? (cl.Extends as IClassifierReference).Target as IClass
11                : null, RegisterNewBaseClass,
12             cl => cl.ParentClass);
13       }
14    }
15 }
```

Listing 1: Synchronization of classes in JaMoPP and the PG metamodel

[5] http://nmfexpressions.codeplex.com

In Line 2, we declare that `Class2Class` is a synchronization rule synchronizing JaMoPP classes with PG classes. Line 4 specifies that whenever we find such two classes that correspond (decided by another method called `ShouldCorrespond`), their names should be synchronized. Line 5-7 specify that each member of a JaMoPP class should correspond to a definition in the PG. The details for this correspondence are left to the `Member2Member` rule.

Lines 8-12 specify that the base classes should be synchronized. The current rule (`Class2Class` should be used to identify corresponding base classes as well, explaining the `this` parameter in Line 8. However, whereas the base class of a Java class in the PG metamodel is available directly as a reference, the base class in JaMoPP is encoded in a classifier reference, making the expression to obtain the base class slightly more complex. As a consequence, NMF Synchronizations is not able to infer how to revert the expression and we have to specify this (i.e. how a JaMoPP class is assigned another class as a base class) through another method, `RegisterNewBaseClass`. With this method, the behavior how to assign a JaMoPP class a new base class is implemented in regular imperative code.

The implementation of `Member2Member` for the case of methods is presented in Listing 2.

```
1  public class Method2MethodDefinition : SynchronizationRule<IMethod, ITMethodDefinition> {
2     public override bool ShouldCorrespond(IMethod left, ITMethodDefinition right, ISynchronizationContext context) {
3        var sig = right.Signature;
4        if (sig == null) return false;
5        var meth = sig.Method;
6        if (meth == null) return false;
7        return left.Name == meth.TName;
8     }
9     public override void DeclareSynchronization() {
10       MarkInstantiatingFor(SyncRule<Member2MemberDefinition>());
11       Synchronize(meth => meth.Name, meth => meth.Signature.Method.TName);
12       LeftToRight.Require(Rule<Method2MethodSignature>(), meth => meth.Name,
13          meth => meth.Parameters.Select(p => GetBaseClass(p.TypeReference)).AsItemEqual(),
14          (meth, signature) => meth.Signature = signature);
15    }
16 }
```

Listing 2: The synchronization rule for method definitions

In this listing, again Line 1 declares `Method2MethodDefinition` as a synchronization rule from JaMoPP methods to PG method definitions. A JaMoPP method should correspond to a PG method definition in a given scope if the methods have the same name here. We specify the exact behavior in lines 2-8. Since the structure of the PG metamodel is very different to JaMoPP in this regard, the method is a few lines long.

Line 11 marks the synchronization rule instantiating for the `Member2Member`-rule. That is, if a member is a method, then the rule `Method2MethodDefinition` should be used to synchronize members, regardless of the transformation direction. Another rule `Field2FieldDefinition` is used for synchronizing fields.

Line 12 denotes that the name of a method in JaMoPP should be kept consistent with the name of the method in the PG model. If we changed the name of a method in JaMoPP, the change is propagated to the PG `TMethod` element. However, this change is propagated back to the JaMoPP model causing all methods that are connected to this PG method to change their name accordingly, regardless of their declaration scope or signature. So we have specified a very powerful rename refactoring in just a single line of code.

NMF Synchronizations under the hood uses the transformation engine of NMF Transformations. In particular, every synchronization rule is mapped to a pair of transformation rules, one for either direction of the synchronizations. Since these transformation rules are still accessible, we can add a dependency to the `Method2MethodSignature` that creates a `TMethodSignature` element for the given name and parameter list. For a given name and parameter list, the transformation engine ensures that only one method

signature element is created. This transformation rule calls another rule `Method2Method` that creates a method element for each string that appears as a method name in the JaMoPP model.

These transformation rules `Method2Method` and `Method2MethodSignature` are called any time the *LeftToRight* rule of the synchronization rule `Method2MethodDefinition` are called. This is done either initially for each method in the JaMoPP model (restricted to at most once per input names and parameter lists) as well as for any new JaMoPP method that is added to the JaMoPP model afterwards.

2.2 Refactoring of the PG Graph

The refactoring part of our solution uses straightforward imperative code to achieve the refactoring operations. As the *Create Superclass* is very simple to implement in classic C# code, we omit a description. The implementation of the *Pull Up Method* refactoring is shown in Listing 3.

```
1  public bool PullUpMethod(TypeGraph typeGraph) {
2    foreach (var method in typeGraph.Methods) {
3      foreach (var signature in method.Signatures) {
4        var methodsGroupsToPull = from def in signature.Definitions
5                                  where def.Overriding == null
6                                  group def by (def.Parent as TClass).ParentClass into methods
7                                  select methods;
8        foreach (var methodGroup in methodsGroupsToPull.Where(group => group.Count() >= 2)) {
9          if (methodGroup.Key != null) {
10           var first = methodGroup.First();
11           var firstParent = first.Parent as ITClass;
12           methodGroup.Key.Defines.Add(first);
13           firstParent.Defines.Remove(first);
14           foreach (var m in methodGroup.Skip(1)) {
15             (m.Parent as ITClass).Defines.Remove(m);
16           }}}}}}
```

Listing 3: The implementation of *Pull Up Method*

The solution utilizes the Language Integrated Query (LINQ) that is around for almost ten years now and used by thousands of developers. Given the conciseness of this specification based on the `TypeGraph` metamodel, we see no reason to use a specialized language for the refactoring. However, due to the online synchronization, we have to be careful to always keep the model in a consistent state, we must not discard the method that should stay as otherwise the connected implementation in the JaMoPP model would be lost. In particular, at least one method element of the PG model must be reused in the refactoring as otherwise no method body is attached to a newly created method element.

3 Evaluation and Discussion

We did not manage to integrate our solution into the ARTE framework suggested by the case authors in order to get a reliable performance comparison, nor did the serialization of the JaMoPP model back to Java source code work. Therefore, we only validated the correctness of our solution manually. Hence, our solution only is a proof-of-concept.

The main insight from the Java Refactoring case for us is that the bidirectional model synchronization of structurally different models is a powerful yet dangerous tool. Powerful because it allows to specify some refactoring operations like renaming in a very concise way. It is dangerous because it is opaque to the developer that the code model is synchronized with a refactoring model, especially because it currently is impossible to break up the synchronization. This synchronization yields that when someone changes the name of a method in the code model, automatically all methods with the same name are

renamed as well. On the other hand, if a method's name is changed into one that already exists, then the method elements in the program graph model are not merged, leading to an inconsistent behavior. In particular, as soon as this operation is performed and one changes the name of such a method in the JaMoPP model, some methods are renamed but others are not as they are synchronized with a different method element in the program graph model.

This is of course a more general problem of unclear semantics synchronizing structurally heterogeneous models with overlapping semantics. It not only related to our solution. A solution for this dilemma would be to disable two-way synchronization but restrict to one-way synchronization against the transformation direction, i.e. that changes in the target model are propagated back to the source.

4 Conclusion

In this paper, we have presented our solution to the Java Refactoring case using NMF Synchronizations. In this solution, we refactor Java code by first loading it into memory as a JaMoPP model, synchronizing this model with a new Program Graph model and refactoring the resulting Program Graph model. As a consequence of the bidirectional synchronization, the original refactoring is automatically applied to the source code model. The advantage here is that the high-level program graph model can be used for a multitude of refactoring operations.

We identified this case as a premier use case for change propagation in the opposite of the primary transformation direction, a feature where we did not see application scenarios yet. This feature will be included in NMF Synchronizations in order to make it more flexible. The bidirectional synchronization that we have applied in the meantime offers powerful refactorings with a very concise implementation but yields consequences hard to foresee.

Our solution is not integrated into the ARTE framework and we therefore do not have any performance results comparing the solution alternative solutions.

References

[1] G. Hinkel, "Change propagation in an internal model transformation language," in *Theory and Practice of Model Transformations*, Springer, 2015.

[2] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[3] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[4] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[5] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende, *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

[6] G. Hinkel, "An approach to maintainable model transformations using an internal DSL," Master's thesis, Karlsruhe Institute of Technology, 2013.

[7] G. Hinkel and L. Happe, "Using component frameworks for model transformations by an internal DSL," in *1st International Workshop on Model-Driven Engineering for Component-Based Systems (ModComp 2014)*, ser. CEUR, 2014.

Java Refactoring Case: a VIATRA Solution*

Dániel Stein Gábor Szárnyas István Ráth

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary

daniel.stein@inf.mit.bme.hu, {szarnyas, rath}@mit.bme.hu

This paper presents a solution for the Java Refactoring Case of the 2015 Transformation Tool Contest. The solution utilises Eclipse JDT for parsing the source code, and uses a visitor to build the program graph. EMF-INCQUERY, VIATRA and the Xtend programming language are used for defining and performing the model transformations.

1 Introduction

This paper describes a solution for the extended version of the TTC 2015 Java Refactoring Case. The source code of the solution is available as an open-source project.[1] There is also a SHARE image available.[2]

The use of automated model transformations is a key factor in modern model-driven system engineering. Model transformations allow the users to query, derive and manipulate large industrial models, including models based on existing systems, e.g. source code models created with reverse engineering techniques. Since such transformations are frequently integrated to modeling environments, they need to feature both high performance and a concise programming interface to support software engineers. EMF-INCQUERY and VIATRA aim to provide an expressive query language and a carefully designed API for defining model queries and transformations.

2 Case Description

Refactoring operations are often used in software engineering to improve the readability and maintainability of existing source code without altering the behaviour of the software. The goal of the Java Refactoring Case [10] is to use model transformation tools to refactor Java source code. We decided to solve the extended version of the case. To achieve this, the solution has to tackle the following challenges:

1. Transforming the *Java source code* to a *program graph* (PG).
2. Performing the refactoring transformation on the program graph.
3. Synchronising the source code and the program graph.

The source code is defined in a restricted sub-language of Java 1.4. The EMF metamodel of the PG is provided in the case description. The case considers two basic refactoring operations: Pull Up Method and Create Superclass. The solution is tested in an automated test framework, ARTE (Automated Refactoring Test Environment).

*This work was partially supported by the MONDO (EU ICT-611125) project.

[1] https://github.com/FTSRG/java-refactoring-ttc-viatra

[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_java-refactoring-viatra.vdi

3 Technologies

Solving the case requires the integration of a model transformation tool and a Java source code parser. In this section, we introduce the technologies used in our solution.

EMF-INCQUERY. The objective of the EMF-INCQUERY [4, 6] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA2 with new features (including transitive closure, role navigation, match count) and tailored it to EMF models, resulting in the INCQUERY Pattern Language [5]. While EMF-INCQUERY is developed with a focus on *incremental query evaluation*, the latest version also provides a *local search-based query evaluation* algorithm.

VIATRA. The VIATRA framework supports the development of model transformations with a particular emphasis on event-driven, reactive transformations [8]. Building upon the incremental query support provided by EMF-INCQUERY, VIATRA offers a language to define transformations and a reactive transformation engine to execute certain transformations upon changes in the underlying model. The current VIATRA project is a full rewrite of the previous VIATRA2 framework, now with full compatibility and support for EMF models.

Java Development Tools. The solution requires a technology to parse the Java code into a program graph model and serialize the modified graph model back to source code. While the case description mentions the JaMoPP [1] and MoDisco [2] technologies, our solution builds on top of the Eclipse Java Development Tools (JDT) [7] used in the Eclipse Java IDE as we were already using JDT in other projects. Compared to the MoDisco framework (which uses JDT internally), we found JDT to be simpler to deploy outside the Eclipse environment, i.e. without defining an Eclipse workspace. Meanwhile, the JaMoPP project has almost completely been abandoned and therefore it is only capable of parsing Java 1.5 source files. While this would not pose a problem for this case, we think it is best to use an actively developed technology such as JDT which supports the latest (1.8) version of the Java language. As JDT is frequently used to parse large source code repositories, it is carefully optimised and supports lazy loading. Unlike JaMoPP and MoDisco, JDT does not produce an EMF model.

4 Implementation

The solution was developed partly in IntelliJ IDEA and partly in the Eclipse IDE. The projects are not tied to any development environment and can be compiled with the Apache Maven [3] build automation tool. This offers a number of benefits, including easy portability and the possibility of continuous integration.

The code is written in Java 8 and Xtend [9]. The queries and transformations were defined in EMF-INCQUERY and VIATRA, respectively. For developing the Xtend code and editing the graph patterns, it is required to use the Eclipse IDE. For setting up the development environment, please refer to the readme file.

4.1 Workflow of the Transformation

Figure 1 shows the high-level workflow of the transformation. This consists of five steps: the source code is parsed into an ASG ①; a PG is produced ②; based on the PG, the possible transformations are

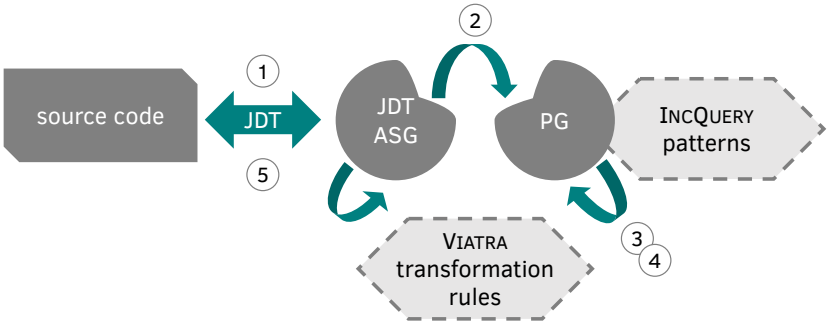


Figure 1: Workflow of the transformation.

calculated ③; if possible, these transformations are executed ④; finally, the results of the transformations are serialized ⑤. In the following, we discuss these steps in detail.

4.2 Parsing the Source Code ①

The solution receives the path to the directory containing the source files. JDT parses these files and returns each file parsed as an AST. These ASTs are interconnected, which means that using JDT's binding resolution mechanism, the developer can navigate from one AST to another one.

4.3 Producing the Program Graph ②

Since JDT does not produce EMF models, the generated ASTs do not support complex queries and traversal operations as the ones provided by EMF and EMF-based query languages (e.g. Eclipse OCL or EMF-INCQUERY). To extract information and to build the PG, our solution applies a visitor resulting a two-pass traversal on the ASG.

1. For each object, the visitor method creates the corresponding object(s) in the PG. Since the order of these visits is non-deterministic, the visitor maintains maps to store the mapping from the objects in the ASG to the objects in the PG. These maps provide trace information between the JDT model and the partially built PG. The visitor also collects the relations between JDT nodes and caches the unique identifiers of each connected node for every relation type.
2. After every compilation unit has been parsed, the previously populated caches are used to create the cross-references between the objects in the PG (e.g. TMember.access).

4.4 Extending the PG with the Trace Model ②③④

The main patterns for both refactoring operations contain a condition that EMF-INCQUERY does not support out of the box, e.g. checking "every child" (a collection of classes) of a certain class. Passing collections as pattern parameter is only possible with a workaround. Also, the INCQUERY Pattern Language does not support universal quantifiers. To overcome these limitations, we extended the program graph metamodel with a *trace model* shown in Figure 2. The trace model defines traces for method signatures and class lists:

- MethodSignatureTrace. Java methods are uniquely identifiable by their signature. The basic PG metamodel contains a TMethodSignature class, which only identifies itself with the name of the method (using a relation to the TMethod object) and the list of its parameter types.

To support querying TMethodSignature objects with EMF-INCQUERY, we created a trace reference for each of them identified by their partial[3] method signature. For example, a method method() expecting a String and an Integer will have the `.method(Ljava/lang/String;I)` trace signature.

- ClassListTrace. To express the collection of classes, a ClassListTrace object will identify them with their signatures joined by the # character. For example, a list of the ChildClass1 and ChildClass2 classes in the example04 package has the `Lexample04/ChildClass1;#Lexample04/ChildClass2;` trace signature.

After the PG is produced, it is extended with the trace model. The traces are based on EMF-INCQUERY patterns (Listing 1) and generated with a VIATRA transformation (Listing 4).

The *universal quantifier* is implemented as a double negation of the existential quantifier using the well-known identity $(\forall x)P(x) \Leftrightarrow \neg(\exists x)\neg P(x)$.

4.5 Refactoring ③④

The refactoring operations are implemented as model transformations on the JDT ASG and the PG. Each model transformation is defined in VIATRA: the LHS is defined with an EMF-INCQUERY pattern and the RHS is defined with imperative Xtend code. As VIATRA does not support bidirectional transformations, for each transformation on the PG, we also execute the corresponding actions on the ASG to keep the two graphs in sync.

4.5.1 Pull Up Method

After creating the method signature traces, the following preconditions must be satisfied before pulling up a method:

- every child class has a method with the given signature,
- the parent class does not have a method with this signature,
- the transformation will not create an unsatisfiable method or field access.

To decide whether the refactoring can be executed, every $\langle$parent class, method signature$\rangle$ pair satisfying the preconditions is collected by the main pattern (possiblePUM). The LHS is defined with six patterns in total. The execution is controlled by parameterising the main pattern listed in Listing 2. The RHS is defined in Listing 5 using one utility pattern.

4.5.2 Create Superclass

To create a new superclass, the parent class and the list of selected classes (connected to a class list trace) have to be passed to the pattern. The transformation can be executed if the following preconditions are satisfied:

- the target parent class does not exist,
- every selected child class has the same parent.

The LHS is defined in Listing 3 with the possibleCSC pattern using five other patterns. The RHS is defined in Listing 6, also using a utility pattern.

[3]The complete signature would also contain the defining type (class or interface) signature and the return type signature.

4.6 Transforming the ASG to Source Code ⑤

The changes in the ASG made by the transformations are propagated to the source code. JDT is capable of incrementally maintaining each source code file (compilation unit) based on the changes in its AST.

5 Evaluation

We executed the tests and used the log files to determine the execution times. The execution times of the test cases are listed in Table 1. The results show that all public and hidden test cases have been executed successfully. Hence, we consider the solution *complete* and *correct*. As the test cases only contained small examples, we cannot draw conclusions on the performance of the solution. Still, it is worth noting that all test cases executed in less than half a second.

The implementation of the solution required quite a lot of code. The patterns were formulated in about 150 lines of INCQUERY Pattern Language code. The transformations required 400 lines of Xtend code, while implementing the interface required by ARTE and the visitor for the transformation required more than 800 lines of Java code. However, the source code is well-structured and is easy to comprehend.

6 Summary

This paper presented a solution for the Java Refactoring case of the 2015 Transformation Tool Contest. The solution addresses both challenges (bidirectional synchronisation and program transformation) and both refactoring operations (Pull Up Method, Create Superclass) defined in the case. The framework is flexible enough to allow the user to define new refactoring operations, e.g. Extract Class or Pull Up Field.

Acknowledgements. The authors would like to thank Ábel Hegedüs, Oszkár Semeráth and Zoltán Ujhelyi for providing valuable insights into EMF-INCQUERY and VIATRA.

References

[1] *JaMoPP*. http://www.jamopp.org/index.php/JaMoPP.

[2] *MoDisco*. https://eclipse.org/MoDisco/.

[3] Apache.org: *Maven*. http://maven.apache.org.

[4] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS2010*, Springer, Springer, doi:http://dx.doi.org/10.1007/978-3-642-16145-2_6. Acceptance rate: 21%.

[5] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF models*. In: *Theory and Practice of Model Transformations, Fourth Intl. Conf., LNCS* 6707, Springer.

[6] Eclipse.org: *EMF-IncQuery*. http://eclipse.org/incquery/.

[7] Eclipse.org: *Java Development Tools (JDT)*. https://eclipse.org/jdt/.

[8] Eclipse.org: *VIATRA Project*. https://www.eclipse.org/viatra/.

[9] Eclipse.org: *Xtend – Modernized Java*. https://www.eclipse.org/xtend/.

[10] Géza Kulcsár, Sven Peldszus & Malte Lochau (2015): *The Java Refactoring Case*. In: *8th Transformation Tool Contest (TTC 2015)*.

A Appendix

A.1 Patterns

```
package hu.bme.mit.ttc.refactoring.patterns

import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"

pattern methodSignature(methodSignature) {
  TMethodSignature(methodSignature);
}

pattern tClassName(tClass, className) {
  TClass(tClass);
  TClass.tName(tClass, className);
}
```

Listing 1: Patterns for generating the trace model.

```
package hu.bme.mit.ttc.refactoring.patterns

import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"

/*
 * Main decision pattern. If the preconditions are statisfied (parentClass
 * and methodSignatureTrace can be bound as parameters), the pattern returns
 * its parameters, if:
 *  - every child class has a method with the given signature (N = M)
 *  - the parent class does not have it already
 *  - the transformation will not create unavailable access
 */
pattern possiblePUM(parentClass : TClass, methodSignatureTrace : MethodSignatureTrace) {
  MethodSignatureTrace.tMethodSignature(methodSignatureTrace, methodSignature);

  // every child class has the method signature
  N == count find childClassesWithSignature(parentClass, _, methodSignature);
  M == count find childClasses(parentClass, _);
  check(N == M && N != 0);

  // parent does not already have this method
  neg find classWithSignature(parentClass, methodSignature);

  // the fields and methods will still be accessible after PUM
  neg find childrenClassMethodDefinitionsAccessingSiblingMembers(childClass, methodSignature);
}

pattern childClasses(parentClass : TClass, childClass : TClass) {
  TClass.childClasses(parentClass, childClass);
}

pattern childClassesWithSignature(parentClass : TClass, clazz : TClass, methodSignature : TMethodSignature)
    {
  TClass(parentClass);
  TClass.childClasses(parentClass, clazz);

  find classWithSignature(clazz, methodSignature);
}

pattern classWithSignature(clazz : TClass, methodSignature : TMethodSignature) {
  TClass(clazz);
  TMethodSignature(methodSignature);
  TMethodSignature.definitions(methodSignature, methodDefinition);
  TClass.defines(clazz, methodDefinition);
```

```
45 }
46
47 pattern methodsAccessingSiblingMembers(methodDefinition : TMethodDefinition) {
48   TMember.access(methodDefinition, accessedMember);
49   TClass.defines(tClass, methodDefinition);
50   TClass.defines(tClass, accessedMember);
51 } or {
52   TClass.defines(tClass, methodDefinition);
53   TMember.access(methodDefinition, accessedMember);
54   TClass.defines(otherClass, accessedMember);
55   TClass.parentClass.childClasses(tClass, otherClass);
56 }
57
58 pattern childrenClassMethodDefinitionsAccessingSiblingMembers(parentClass : TClass, methodSignature :
       TMethodSignature) {
59   TClass.childClasses(parentClass, childClass);
60   TClass.defines(childClass, methodDefinition);
61   TMethodSignature.definitions(methodSignature, methodDefinition);
62   find methodsAccessingSiblingMembers(methodDefinition);
63 }
64
65 // fire precondition pattern
66 pattern classWithName(tClass : TClass, className) {
67   TClass.tName(tClass, className);
68 }
69
70 // fire precondition pattern
71 pattern methodWithSignature(trace : MethodSignatureTrace, signature) {
72   MethodSignatureTrace.signatureString(trace, signature);
73 }
74
75 // pattern for PG refactor
76 pattern methodDefinitionInClassList(parentClass : TClass, methodSignature : TMethodSignature, clazz :
       TClass, methodDefinition : TMethodDefinition) {
77   TClass.childClasses(parentClass, clazz);
78   TMethodSignature.definitions(methodSignature, methodDefinition);
79   TClass.defines(clazz, methodDefinition);
80 }
```

Listing 2: Patterns for the Pull Up Method refactoring.

```
1  package hu.bme.mit.ttc.refactoring.patterns
2
3  import "platform:/plugin/TypeGraphBasic/model/TypeGraphBasic.ecore"
4  import "platform:/plugin/TypeGraphBasic/model/TypeGraphTrace.ecore"
5
6  /*
7   * Main decision pattern. If the preconditions are statisfied (the
8   * targetClass should not exist), the pattern returns its parameters, if:
9   *  - every child class has the same parent
10  */
11 pattern possibleCSC(concatSignature, methodSignature : TMethodSignature) {
12   ClassListTrace.concatSignature(classListTrace, concatSignature);
13   ClassListTrace.tClasses.signature(classListTrace, methodSignature);
14
15   neg find childClassesWithDifferentParents(classListTrace, _, _);
16 }
17
18 pattern childClassesWithDifferentParents(classListTrace : ClassListTrace, classOne : TClass, classTwo :
       TClass) {
19   ClassListTrace.tClasses(classListTrace, classOne);
20   ClassListTrace.tClasses(classListTrace, classTwo);
21   find differentParents(classOne, classTwo);
22 }
23
```

```
24 pattern differentParents(classOne : TClass, classTwo : TClass) {
25   TClass.parentClass(classOne, parentClassOne);
26   TClass.parentClass(classTwo, parentClassTwo);
27   parentClassOne != parentClassTwo;
28 } or {
29   TClass(classTwo);
30   find hasParent(classOne);
31   neg find hasParent(classTwo);
32 } or {
33   TClass(classOne);
34   find hasParent(classTwo);
35   neg find hasParent(classOne);
36 }
37
38 pattern hasParent(tClass : TClass) {
39   TClass.parentClass(tClass, _);
40 }
41
42 pattern classesOfClassListTrace(concatSignature, tClass : TClass) {
43   ClassListTrace.concatSignature(classListTrace, concatSignature);
44   ClassListTrace.tClasses(classListTrace, tClass);
45 }
46
47 pattern methodSignatureAndTrace(trace : MethodSignatureTrace, methodSignature : TMethodSignature) {
48   MethodSignatureTrace.tMethodSignature(trace, methodSignature);
49 }
50
51 // pattern for PG refactor
52 pattern packageWithName(tPackage : TPackage, packageName) {
53   TPackage.tName(tPackage, packageName);
54 }
55
56 // pattern for PG refactor
57 pattern typeGraphs(typeGraph : TypeGraph) {
58   TypeGraph(typeGraph);
59 }
60
61 // fire precondition pattern
62 pattern classWithName(tClass : TClass, className) {
63   TClass.tName(tClass, className);
64 }
```

Listing 3: Patterns for the Create Superclass refactoring.

A.2 Transformations

```
1  package hu.bme.mit.ttc.refactoring.transformations
2
3  import TypeGraphBasic.TClass
4  import TypeGraphTrace.Trace
5  import TypeGraphTrace.TypeGraphTracePackage
6  import hu.bme.mit.ttc.refactoring.patterns.TraceQueries
7  import java.util.ArrayList
8  import java.util.List
9  import org.apache.log4j.Level
10 import org.eclipse.emf.ecore.resource.Resource
11 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
12 import org.eclipse.incquery.runtime.evm.api.RuleEngine
13 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
14 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
15 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
16 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
17 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
```

```
18 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
19 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
20
21 class TraceTransformation {
22
23   extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
24   extension BatchTransformation transformation
25   extension BatchTransformationStatements statements
26   extension IModelManipulations manipulation
27
28   extension TypeGraphTracePackage tgtPackage = TypeGraphTracePackage::eINSTANCE
29   extension TraceQueries queries = TraceQueries::instance
30   val AdvancedIncQueryEngine engine
31   Resource resource
32   val Trace trace
33
34   new(AdvancedIncQueryEngine engine, Resource resource) {
35     this(RuleEngines.createIncQueryRuleEngine(engine), resource)
36   }
37
38   new(RuleEngine ruleEngine, Resource resource) {
39     engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
40     transformation = BatchTransformation.forEngine(engine)
41     statements = new BatchTransformationStatements(transformation)
42     manipulation = new SimpleModelManipulations(iqEngine)
43     transformation.ruleEngine.logger.level = Level::OFF
44     this.resource = resource
45     this.trace = resource.contents.get(0) as Trace
46   }
47
48   val methodSignatureTraceRule = createRule.precondition(methodSignature).action [
49     val methodSignatureTrace = typeGraphTraceFactory.createMethodSignatureTrace
50     trace.methodSignatures += methodSignatureTrace
51
52     val sb = new StringBuilder(".")
53     sb.append(methodSignature.method.TName)
54     sb.append("(")
55     methodSignature.paramList.forEach[sb.append(it.TName)]
56     sb.append(")")
57
58     methodSignatureTrace.signatureString = sb.toString
59     methodSignatureTrace.TMethodSignature = methodSignature
60   ].build
61
62
63   def run() {
64     fireAllCurrent(methodSignatureTraceRule)
65   }
66
67   def addNewClassListTrace(List<String> classSignatures) {
68     val List<TClass> tClasses = new ArrayList
69     for (signature : classSignatures) {
70       tClasses += engine.getMatcher(TClassName).getAllValuesOftClass(signature)
71     }
72
73     val classListTrace = typeGraphTraceFactory.createClassListTrace
74     classListTrace.concatSignature = classSignatures.join("#")
75     classListTrace.TClasses += tClasses
76
77     trace.classLists += classListTrace
78     return classListTrace
79   }
80 }
```

Listing 4: Transformation for generating the trace model.

```
1  package hu.bme.mit.ttc.refactoring.transformations
2
3  import TypeGraphBasic.TClass
4  import TypeGraphBasic.TypeGraphBasicPackage
5  import TypeGraphTrace.MethodSignatureTrace
6  import com.google.common.collect.BiMap
7  import hu.bme.mit.ttc.refactoring.patterns.PUMQueries
8  import java.io.File
9  import java.util.ArrayList
10 import java.util.List
11 import java.util.Scanner
12 import java.util.Set
13 import org.apache.log4j.Level
14 import org.eclipse.emf.ecore.util.EcoreUtil
15 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
16 import org.eclipse.incquery.runtime.evm.api.RuleEngine
17 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
18 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
19 import org.eclipse.jdt.core.dom.ASTNode
20 import org.eclipse.jdt.core.dom.CompilationUnit
21 import org.eclipse.jdt.core.dom.MethodDeclaration
22 import org.eclipse.jdt.core.dom.TypeDeclaration
23 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
24 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
25 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
26 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
27 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
28
29 class PUMTransformation {
30   extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
31   extension BatchTransformation transformation
32   extension BatchTransformationStatements statements
33   extension IModelManipulations manipulation
34
35   extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
36   extension PUMQueries queries = PUMQueries::instance
37
38   val AdvancedIncQueryEngine engine
39   val String parentSignature
40   val String methodSignature
41   val BiMap<String, CompilationUnit> compilationUnits
42
43   new(AdvancedIncQueryEngine engine, String parentSignature, String methodSignature, BiMap<String,
        CompilationUnit> compilationUnis) {
44     this(RuleEngines.createIncQueryRuleEngine(engine), parentSignature, methodSignature, compilationUnis)
45   }
46
47   new(RuleEngine ruleEngine, String parentSignature, String methodSignature, BiMap<String, CompilationUnit>
       compilationUnits) {
48     engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
49     transformation = BatchTransformation.forEngine(engine)
50     statements = new BatchTransformationStatements(transformation)
51     manipulation = new SimpleModelManipulations(iqEngine)
52     transformation.ruleEngine.logger.level = Level::OFF
53
54     this.parentSignature = parentSignature
55     this.methodSignature = methodSignature
56     this.compilationUnits = compilationUnits
57
58     compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
59   }
60
61   val PUMRule = createRule.precondition(possiblePUM).action [
62     val parentClassKey = parentClass.TName
```

```
63      val childClasses = engine.getMatcher(childClasses).getAllValuesOfchildClass(parentClass)
64
65      var TypeDeclaration astParentClass
66      var List<TypeDeclaration> astChildClasses = new ArrayList
67      var List<MethodDeclaration> astMethodDeclarations
68
69      astParentClass = findCompilationUnits(parentClassKey, childClasses, astChildClasses)
70      astMethodDeclarations = findMethodDeclarations(astChildClasses, methodSignatureTrace)
71
72      updateASTAndSerialize(astParentClass, astChildClasses, astMethodDeclarations)
73
74
75      // --------------- /\ JDT transformation ------------- PG transformation \/ ---------------
76
77
78      val methodDefinitionsToDelete = engine.getMatcher(methodDefinitionInClassList).getAllMatches(
79        parentClass, methodSignatureTrace.TMethodSignature, null, null
80      )
81
82      val firstMethodDefinition = methodDefinitionsToDelete.get(0)
83      val savedSignature = firstMethodDefinition.methodSignature
84      val savedReturnType = firstMethodDefinition.methodDefinition.returnType
85      val savedAccess = firstMethodDefinition.methodDefinition.access
86
87      methodDefinitionsToDelete.forEach[
88        it.clazz.signature.remove(it.methodDefinition.signature); // remove signature from class
89        EcoreUtil.delete(it.methodDefinition, true)  // remove the method definition
90      ]
91
92      val tMethodDefinition = tgPackage.typeGraphBasicFactory.createTMethodDefinition
93      tMethodDefinition.returnType = savedReturnType
94      tMethodDefinition.signature = savedSignature
95      tMethodDefinition.access += savedAccess
96
97      parentClass.defines += tMethodDefinition
98
99      println(tMethodDefinition)
100   ].build
101
102   protected def readFileToString(String path) {
103     new Scanner(new File(path)).useDelimiter("\\A").next
104   }
105
106   protected def TypeDeclaration findCompilationUnits(String parentClassKey, Set<TClass> childClasses, List<
        TypeDeclaration> astChildClasses) {
107     var TypeDeclaration result
108     for (cu : compilationUnits.values) {
109       // the just created CU can not resolve
110       val firstTypeKey = "L"
111             + cu.package.name.fullyQualifiedName.replace('.', '/')
112             + "/"
113             + ((cu.types.get(0) as TypeDeclaration).name.fullyQualifiedName)
114             + ";"
115
116       if (parentClassKey.equals(firstTypeKey)) {
117         result = cu.types.get(0) as TypeDeclaration
118       }
119
120       for (child : childClasses) {
121         if (firstTypeKey.equals(child.TName)) {
122           astChildClasses += cu.types.get(0) as TypeDeclaration
123         }
124       }
125     }
126
```

```
127     return result
128   }
129
130   protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses,
          MethodSignatureTrace methodSignatureTrace) {
131     val List<MethodDeclaration> astMethodDeclarations = new ArrayList
132
133     for (childCU : astChildClasses) {
134       val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
135       val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY)
          as List<TypeDeclaration>
136       for (type : types) {
137         for (method : (type as TypeDeclaration).methods) {
138           if (method.resolveBinding.key.startsWith(methodSignature)) {
139             astMethodDeclarations += method
140           }
141         }
142       }
143     }
144
145     return astMethodDeclarations
146   }
147
148   protected def updateASTAndSerialize(TypeDeclaration astParentClass, List<TypeDeclaration> astChildClasses
          , List<MethodDeclaration> astMethodDeclarations) {
149     if (astMethodDeclarations.size > 0) {
150       astParentClass.bodyDeclarations.add(ASTNode.copySubtree(astParentClass.AST, astMethodDeclarations.get
          (0)) as MethodDeclaration)
151
152       for (methodDeclaration : astMethodDeclarations) {
153         methodDeclaration.delete
154       }
155     }
156   }
157
158     def fire() {
159     fireAllCurrent(
160       PUMRule,
161       "parentClass.tName" -> parentSignature,
162       "MethodSignatureTrace.signatureString" -> methodSignature
163       )
164   }
165
166   def canExecutePUM() {
167     // get the method signature by string, then get one arbitrary match with it bound
168     val parentTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, parentSignature)
169     val trace = engine.getMatcher(methodWithSignature).getOneArbitraryMatch(null, methodSignature)
170
171     return
172     parentTClass != null &&
173     trace != null &&
174     engine.getMatcher(possiblePUM).getOneArbitraryMatch(parentTClass.TClass, trace.trace) != null
175   }
176 }
```

Listing 5: Pull Up Method transformation.

```
1 package hu.bme.mit.ttc.refactoring.transformations
2
3 import TypeGraphBasic.TClass
4 import TypeGraphBasic.TMethodSignature
5 import TypeGraphBasic.TPackage
6 import TypeGraphBasic.TypeGraph
7 import TypeGraphBasic.TypeGraphBasicPackage
8 import com.google.common.collect.BiMap
```

```
 9 import hu.bme.mit.ttc.refactoring.patterns.CSCQueries
10 import java.io.File
11 import java.util.ArrayList
12 import java.util.List
13 import java.util.Scanner
14 import java.util.Set
15 import org.apache.commons.lang3.StringUtils
16 import org.apache.log4j.Level
17 import org.eclipse.incquery.runtime.api.AdvancedIncQueryEngine
18 import org.eclipse.incquery.runtime.evm.api.RuleEngine
19 import org.eclipse.incquery.runtime.evm.specific.RuleEngines
20 import org.eclipse.incquery.runtime.evm.specific.event.IncQueryEventRealm
21 import org.eclipse.jdt.core.dom.ASTNode
22 import org.eclipse.jdt.core.dom.CompilationUnit
23 import org.eclipse.jdt.core.dom.MethodDeclaration
24 import org.eclipse.jdt.core.dom.Modifier.ModifierKeyword
25 import org.eclipse.jdt.core.dom.Name
26 import org.eclipse.jdt.core.dom.Type
27 import org.eclipse.jdt.core.dom.TypeDeclaration
28 import org.eclipse.viatra.emf.runtime.modelmanipulation.IModelManipulations
29 import org.eclipse.viatra.emf.runtime.modelmanipulation.SimpleModelManipulations
30 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationRuleFactory
31 import org.eclipse.viatra.emf.runtime.rules.batch.BatchTransformationStatements
32 import org.eclipse.viatra.emf.runtime.transformation.batch.BatchTransformation
33
34 class CSCTransformation {
35   extension BatchTransformationRuleFactory factory = new BatchTransformationRuleFactory
36   extension BatchTransformation transformation
37   extension BatchTransformationStatements statements
38   extension IModelManipulations manipulation
39
40   extension TypeGraphBasicPackage tgPackage = TypeGraphBasicPackage::eINSTANCE
41   extension CSCQueries queries = CSCQueries::instance
42
43   val AdvancedIncQueryEngine engine
44   val String concatSignature
45   val String targetPackage
46   val String targetName
47   val BiMap<String, CompilationUnit> compilationUnits
48
49   var CompilationUnit targetCU
50
51   new(AdvancedIncQueryEngine engine, List<String> childClassSignatures, String targetPackage, String
       targetName, BiMap<String, CompilationUnit> compilationUnis) {
52     this(RuleEngines.createIncQueryRuleEngine(engine), childClassSignatures, targetPackage, targetName,
       compilationUnis)
53   }
54
55   new(RuleEngine ruleEngine, List<String> childClassSignatures, String targetPackage, String targetName,
       BiMap<String, CompilationUnit> compilationUnits) {
56     engine = (ruleEngine.eventRealm as IncQueryEventRealm).engine as AdvancedIncQueryEngine
57     transformation = BatchTransformation.forEngine(engine)
58     statements = new BatchTransformationStatements(transformation)
59     manipulation = new SimpleModelManipulations(iqEngine)
60     transformation.ruleEngine.logger.level = Level::OFF
61
62     this.concatSignature = childClassSignatures.join("#")
63     this.targetPackage = targetPackage
64     this.targetName = targetName
65     this.compilationUnits = compilationUnits
66
67     compilationUnits.values.forEach[ try { it.recordModifications } catch (Exception e) {}]
68   }
69
70   val CSCRule = createRule.precondition(possibleCSC).action [
```

```
71      val tClasses = engine.getMatcher(classesOfClassListTrace).getAllValuesOftClass(concatSignature)
72
73      val List<TypeDeclaration> astChildClasses = findCompilationUnits(tClasses)
74
75      val firstChild = astChildClasses.get(0)
76
77      if (targetCU == null) {
78        targetCU = createTargetClass(firstChild, firstChild.superclassType)
79      }
80
81      setParentClass(astChildClasses)
82
83      serializeCUs
84
85
86      // --------------- /\ JDT transformation ------------- PG transformation \/ ---------------
87
88      val oldParentTClass = tClasses.get(0).parentClass
89      if (oldParentTClass != null) {
90        oldParentTClass.childClasses -= tClasses
91      }
92
93      val targetSignature = "L" + targetPackage.replace('.', '/') + "/" + targetName + ";";
94      val typeGraph = engine.getMatcher(typeGraphs).oneArbitraryMatch.typeGraph
95
96      val targetTClassMatch = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)
97      var TClass targetTClass
98      if (targetTClassMatch == null) {
99        targetTClass = tgPackage.typeGraphBasicFactory.createTClass
100       targetTClass.TName = targetSignature
101
102       targetTClass.package = createPackagesFor(typeGraph, targetPackage)
103       targetTClass.parentClass = oldParentTClass
104     } else {
105       targetTClass = targetTClassMatch.TClass
106     }
107
108     (tClasses.get(0).eContainer as TypeGraph).classes += targetTClass
109     targetTClass.childClasses += tClasses
110   ].build
111
112   protected def createPackagesFor(TypeGraph typeGraph, String pkg) {
113     val String[] split = pkg.split("\\.");
114
115     var previous = "";
116     var TPackage previousTPackage
117     for (var i = 0; i < split.length; i++) {
118       var String current = previous
119       if (i != 0)  {
120         current += "."
121       }
122       current += split.get(i);
123
124       var currentTPackageMatch = engine.getMatcher(packageWithName).getOneArbitraryMatch(null, current)
125       if (currentTPackageMatch != null) {
126         previousTPackage = currentTPackageMatch.TPackage
127       } else {
128         val TPackage currentTPackage = tgPackage.typeGraphBasicFactory.createTPackage
129         currentTPackage.TName = current
130         if (previousTPackage != null) {
131           currentTPackage.parent = previousTPackage
132         } else {
133           typeGraph.packages += currentTPackage
134         }
135
```

```
136         previousTPackage = currentTPackage
137       }
138     }
139
140     previousTPackage
141   }
142
143   protected def List<TypeDeclaration> findCompilationUnits(Set<TClass> childClasses) {
144     val List<TypeDeclaration> astChildClasses = new ArrayList
145
146     for (cu : compilationUnits.values) {
147       for (child : childClasses) {
148         if (cu.findDeclaringNode(child.TName) != null) {
149           astChildClasses += cu.findDeclaringNode(child.TName) as TypeDeclaration
150         }
151       }
152     }
153
154     return astChildClasses
155   }
156
157   protected def List<MethodDeclaration> findMethodDeclarations(List<TypeDeclaration> astChildClasses,
       TMethodSignature tMethodSignature) {
158     val List<MethodDeclaration> astMethodDeclarations = new ArrayList
159     val methodSignatureTrace = engine.getMatcher(methodSignatureAndTrace).getAllValuesOftrace(
       tMethodSignature).get(0)
160
161     for (childCU : astChildClasses) {
162       val methodSignature = childCU.resolveBinding.key + methodSignatureTrace.signatureString;
163       val types = (childCU.root as CompilationUnit).getStructuralProperty(CompilationUnit.TYPES_PROPERTY)
       as List<TypeDeclaration>
164       for (type : types) {
165         for (method : (type as TypeDeclaration).methods) {
166           // match
167           if (method.resolveBinding.key.startsWith(methodSignature)) {
168             astMethodDeclarations += method
169           }
170         }
171       }
172     }
173
174     return astMethodDeclarations
175   }
176
177   protected def CompilationUnit createTargetClass(TypeDeclaration childClass, Type superClassType) {
178     val ast = childClass.AST
179     val compilationUnit = ast.newCompilationUnit
180
181     if (targetPackage != null) {
182       val packageDeclaration = ast.newPackageDeclaration
183       var Name packageName
184       for (part : targetPackage.split("\\.")) {
185         if (packageName == null) {
186           packageName = ast.newSimpleName(part)
187         } else {
188           packageName = ast.newQualifiedName(packageName, ast.newSimpleName(part))
189         }
190       }
191       packageDeclaration.name = packageName
192       compilationUnit.package = packageDeclaration
193     }
194
195     compilationUnit.imports += ASTNode.copySubtrees(ast, (childClass.root as CompilationUnit).imports)
196
197     val typeDeclaration = ast.newTypeDeclaration
```

```
198     typeDeclaration.modifiers().add(ast.newModifier(ModifierKeyword.PUBLIC_KEYWORD))
199     typeDeclaration.name = ast.newSimpleName(targetName)
200
201     if (superClassType != null) {
202       typeDeclaration.superclassType = ASTNode.copySubtree(ast, superClassType) as Type
203     }
204
205     compilationUnit.types += typeDeclaration
206
207     compilationUnit
208   }
209
210   protected def insertMethodDeclaration(MethodDeclaration declaration) {
211     val typeDeclaration = targetCU.types.get(0) as TypeDeclaration
212     typeDeclaration.bodyDeclarations.add(ASTNode.copySubtree(targetCU.AST, declaration) as
        MethodDeclaration)
213   }
214
215   protected def setParentClass(List<TypeDeclaration> typeDeclarations) {
216     val ast = targetCU.AST
217
218     var Type fqn
219     if (targetPackage != null) {
220       for (part : targetPackage.split("\\.")) {
221         if (fqn == null) {
222           fqn = ast.newSimpleType(ast.newSimpleName(part))
223         } else {
224           fqn = ast.newQualifiedType(fqn, ast.newSimpleName(part))
225         }
226       }
227
228       fqn = ast.newQualifiedType(fqn, ast.newSimpleName(targetName))
229     } else {
230       fqn = ast.newSimpleType(ast.newSimpleName(targetName))
231     }
232
233     for (declaration : typeDeclarations) {
234       declaration.superclassType = ASTNode.copySubtree(declaration.AST, fqn) as Type
235     }
236   }
237
238   protected def removeChildMethodDeclarations(List<MethodDeclaration> methodDeclarations) {
239     for (declaration : methodDeclarations) {
240       declaration.delete
241     }
242   }
243
244   def serializeCUs() {
245     val targetDir = StringUtils.substringBefore(
246                 compilationUnits.keySet.get(0),
247                 "/src/"
248               ) + "/src/" + targetPackage.replace('.', '/')
249     val targetPath = targetDir + "/" + targetName + ".java"
250
251     val targetFile = new File(targetPath)
252     targetFile.parentFile.mkdirs
253
254     compilationUnits.put(targetPath, targetCU)
255   }
256
257   protected def readFileToString(String path) {
258     new Scanner(new File(path)).useDelimiter("\\A").next
259   }
260
261
```

```
262   def fire() {
263     fireAllCurrent(
264       CSCRule,
265       "concatSignature" -> concatSignature
266       )
267   }
268
269   def canExecuteCSC() {
270     val targetSignature = "L" + targetPackage.replace('.', '/') + "/" +  targetName + ";"
271     val targetTClass = engine.getMatcher(classWithName).getOneArbitraryMatch(null, targetSignature)
272
273     if (targetTClass != null) {
274       return false
275     }
276
277     engine.getMatcher(possibleCSC).countMatches > 0
278   }
279
280 }
```

Listing 6: Create Superclass transformation.

A.3 Metamodel of the Trace Model

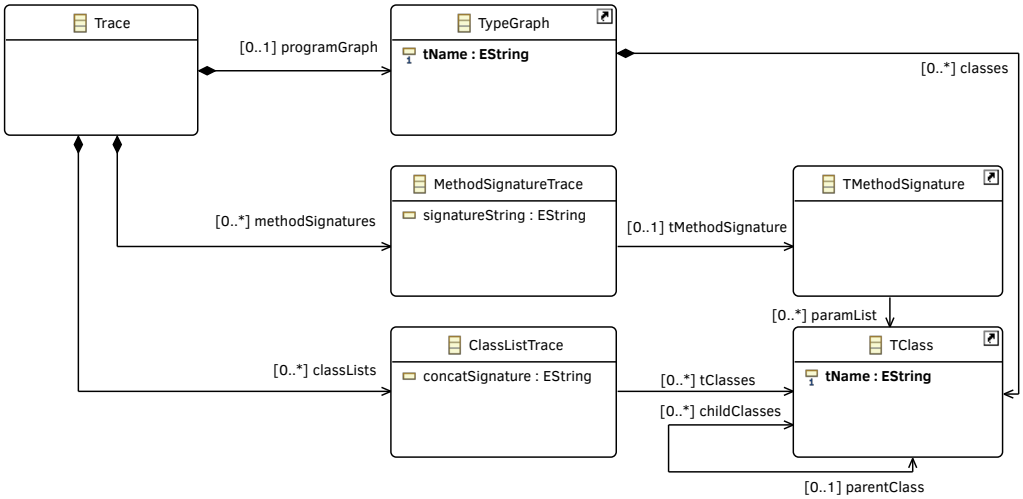


Figure 2: Metamodel of the trace model.

A.4 Benchmark Results

The benchmarks were conducted on a 64-bit Arch Linux virtual machine running in SHARE. The machine utilized a single core of a 2.00 GHz Xeon E5-2650 CPU and 1 GB of RAM. We used OpenJDK 8 to run the ARTE framework and the solution.

test case	time [s]
pub_pum1_1_paper1	0.463
pub_pum1_2	0.013
pub_pum2_1	0.333
pub_pum3_1	0.094
pub_csc1_1	0.189
pub_csc1_2	0.093
hidden_pum1_1	0.063
hidden_pum1_2	0.013
hidden_pum2_1	0.114
hidden_pum2_2	0.082
hidden_csc1_1	0.081
hidden_csc1_2	0.007
hidden_csc2_1	0.058
hidden_csc3_1a	0.189
hidden_csc3_1	0.179

Table 1: Execution times for the test cases.

A Solution to the Java Refactoring Case Study using eMoflon

Sven Peldszus
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Géza Kulcsár
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

Malte Lochau
Technische Universität Darmstadt
Real-Time Systems Lab
Merckstr. 25
64283 Darmstadt, Germany

{sven.peldszus@stud|geza.kulcsar@es|malte.lochau@es}.tu-darmstadt.de

Our solution to the Java Refactoring case study of the Transformation Tool Contest (TTC 2015) is implemented using eMoflon, a meta-modeling and model transformation tool developed at the Real-Time Systems Lab at TU Darmstadt. The solution, available as a virtual machine hosted on SHARE [5] and at GitHub [6], includes a bidirectional synchronization between a Java model and an abstract program graph specified using Triple Graph Grammars (TGG) as well as a graph-based implementation for two refactoring operations using Story Driven Modeling (SDM).

1 Introduction

The Java Refactoring case study [3] of the Transformation Tool Contest 2015[1] revolves around a challenging object-oriented refactoring scenario. Two classical refactoring operations, `Create Superclass` and `Pull Up Method`, have to be implemented by solution developers, taking Java source code as input and producing a refactored version of it as output. We use a meta-model specified in the case study, called the Program Graph (PG). The PG is an abstract representation of the input Java program and is used to define and perform the given refactoring operations on this model of the program. One of the main difficulties comes from the bidirectional nature of synchronizing source code and program graph. Our tool eMoflon [4] supports both EMF meta-modeling and bidirectional transformations using *Triple Graph Grammars* (TGGs). TGGs [8] are a rule-based, declarative language, which can be used for specifying transformations, where both directions (forward and backward transformation) can be derived from the same specification.

Another eMoflon feature, *Story Driven Modeling* (SDM), [1] is used in our solution to implement refactorings. SDM is a visual language for describing programmed graph rewritings; an SDM method consists of a set of graph transformation rules with an additional control flow specification to describe their execution order dependencies in an imperative fashion.

In this paper, we investigate to what extent TGGs are able to cope with advanced bidirectional text-to-model scenarios with change propagation by solving the Java Refactoring case study of the TTC 2015. We use the given PG format as the abstract representation for Java programs. In the following, we provide a stepwise, detailed description of the solution including the technical difficulties that arose and evaluate the solution.

[1] http://www.transformation-tool-contest.eu/

2 The Solution using eMoflon, TGGs and SDM

In the following, we give a detailed description of the steps of our solution.

Java to JaMoPP. The Java source code is parsed and converted into an intermediate EMF representation using the JaMoPP framework [2]. To quote the website of JaMoPP: "JaMoPP is a set of Eclipse plug-ins that can be used to parse Java source code into EMF-based models and vice versa."[2]

JaMoPP to PG. While working with the JaMoPP meta-model for Java, we have found out that some parts of it do not comply with the PG meta-model and with some properties of the planned TGG translation. Two preprocessing actions are necessary to make a JaMoPP model instance TGG-conform.

Creating the package structure. JaMoPP encodes the package hierarchy of the program into dot separated string or as array of strings. As it would require extra efforts and the usage of external hand-written code to handle these constructs when specifying our TGG, we decided to implement this transformation as a preprocessing step in order to keep our TGG clean and concise.

Retaining the parameter order of methods. A transformation specified by a set of TGG rules is per definition nondeterministic, i.e., if the source side of a rule has multiple matches in a source model, we cannot be sure in which order they will be processed. To preserve the original order of a parameter list, which is represented by independent child nodes of a method node, we have to turn the set of parameter nodes into a list representation so that the parameter nodes can only be processed in the given order.

TGGs describe a correspondence between instances of a *source* and a *target* meta-model, specified by means of a mediating *correspondence graph* (hence the name Triple Graph Grammars). A TGG specification consists of declarative rules. A transformation using TGGs consists in building up a target model incrementally on the basis of a source model (or vice versa) using the correspondence links between the elements of the models. Applying a TGG rule essentially means that a given structure in the target model is built up which corresponds to a part of the source model which is matched by the source side of the rule definition.

Our TGG specification consists of 20 rules. We have identified 5 main components which have to be considered: initialization of the PG, packages, classes, methods, and fields. In Figure 1, we show a sample rule `MethodNameCreate` to introduce our visual TGG syntax and to give an idea about the rule semantics. For further details of the TGG implementation, please refer to [5, 6].

By convention, the source node part is on the left and the target node side is on the right, with the correspondence graph (hexagonal boxes) in between. Boxes and edges marked with ++ (highlighted in green) are the elements created by the rule application. All other boxes and edges represent the context (elements which have to be present for the rule to be applied). A crossed-out box denotes a negative application condition: the object must not be part of the context. The box with an expression and two outgoing edges (in the middle) is a constraint, which ensures that the name attributes of the referred elements have the same value (here, the built-in `eq` function is used; however, there are various other built-in functions and the developer can also create custom ones). The meaning of this rule is the following: whenever there is a class in the source with a corresponding class in the target, if a method of the source class is not yet translated (thus, processed at application time, hence its green color), and the target PG does not have a method with the same name, then a new method and a corresponding method definition are created in the target.

Refactoring of the PG. The refactoring rules `Pull Up Method` and `Create Superclass` have been implemented using Story Driven Modeling (SDM) [1]. As these operations do not have to be bidirectional, it was a convenient choice to use SDMs which comprise a more flexible way of specifying

[2] http://www.jamopp.org/

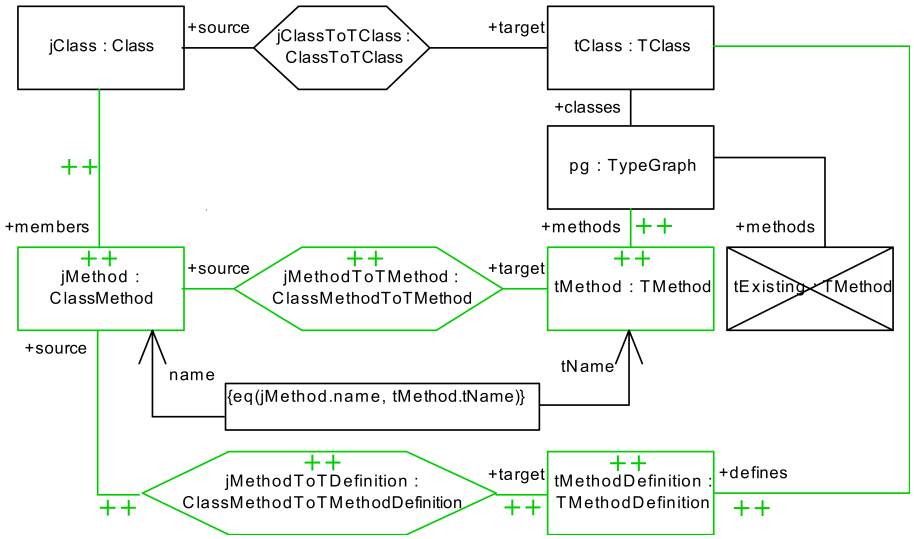


Figure 1: Example TGG rule `MethodNameCreate`

transformations compared to TGGs.

SDMs provide a way to implement methods of classes of a meta-model (similar to object-oriented programming) in a visual manner based on graph transformation, combining declarative graph transformation rules with an imperative control flow. The basic building blocks of an SDM specification are the *story nodes*. Each story node contains a single graph transformation operation, which is applied according to the standard graph transformation principles (i.e., nondeterministically on a matching part of the model) when the story node is activated. The story nodes are activated as determined by the control flow, with the additional possibilities of adding `if-else` conditions and `for each` loops.

There are two methods implemented for both refactoring operations in the corresponding classes of the PG meta-model. The `isApplicable` methods simply check the feasibility of the rule application to prevent the modification of the PG if a refactoring is not even executable. Thereupon, the `Perform` methods perform the actual refactorings if possible.

In this paper, we omit an elaborated presentation of all our SDM methods; instead we show an example method, introduce our visual SDM syntax, give an intuition about how the method works and refer the reader to [5, 6] for further details.

Figure 2 shows an example SDM method `csc_Perform` which implements the actual application of the `Create Superclass` after the preconditions have been checked. The execution starts with the start node (black circle on top left) and follows the arrows. The larger rounded boxes denote story nodes; each story node contains a graph transformation rule which is applied as the containing story node is activated. A rule application consists in finding a match for the depicted graph pattern in the model where the SDM method has been called, deleting the elements marked with `--` (highlighted in red) and creating the ones marked with `++` (highlighted in green). Boxes with a thick edge correspond to bound object variables that are matched to a fixed object in the model. A story node may have two outgoing edges: the execution continues through `Success` if the application was successful and through `Failure` if not. Story nodes can alternatively contain external method calls. Cascaded-style boxes represent `for each` loops, where the rule is applied to each possible match in the model with a loop body executed after each match (`Each Time` edge). After all the matches have been processed, the loop is exited (`End` edge).

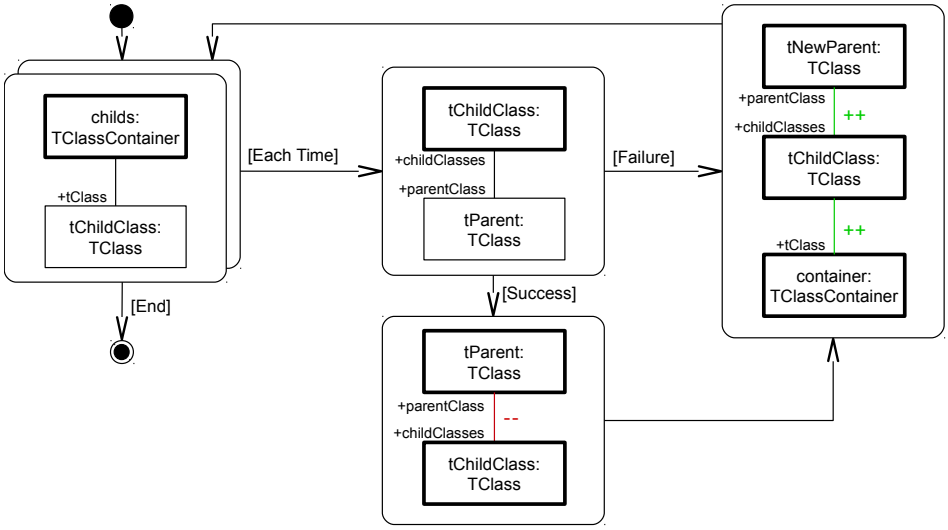


Figure 2: Example SDM method `csc_Perform`

The depicted rule, `csc_Perform`, does the following: after putting the new parent class into the PG by creating the corresponding edge, the old parent of the child classes is identified. Afterwards, in a loop, the parent reference of each child class is newly created to point to the parent created by the refactoring and the old reference is deleted.

PG to JaMoPP. As our TGG describes both a forward and a backward transformation, this step of the transformation requires no extra development efforts. TGGs in eMoflon provide a synchronisation algorithm based on model deltas: whenever one side of a TGG (in our case, the PG instance) is changed, the modification delta is calculated and the TGG mechanism is able to update the other side of the model in correspondence with the change delta. Multiple refactoring operations are performed as a single batch after all the preconditions have been checked by using a bookkeeping mechanism.

JaMoPP to Java. Similar to the first step, the translation of the EMF model to Java code belongs to the central functionality of JaMoPP.

3 Evaluation

Correctness and performance. The case study contains 20 test cases, in which one or more refactorings have to be performed. The feasibility of the given refactoring operations is correctly determined in all test cases. Most of the execution time (60 %) is spent with the Java-to-PG transformation, where JaMoPP consumes almost 30 % of the overall time; although we expected the TGG execution to be the most expensive step, it only takes about 14 % of the whole process (together in both directions). The average execution time for one test case is 0.3367 sec.

Soft aspects. Utilizing TGGs for the synchronization part is responsible for the greatest advantages and disadvantages at once. TGGs provide a powerful declarative language, where the resulting transformations between the source and the target models are consistent regarding the correspondence specified by the TGG. Moreover, by using TGGs, the synchronization part of the challenge requires no extra efforts as a model synchronization algorithm for TGG specifications is already part of eMoflon. The price to pay for those formal and algorithmic properties is the slower execution time compared to task-optimized,

imperative solutions. Extending a TGG might also become problematic as new rules might overlap with old ones, thus, possibly altering the behavior of the core specification.

By using SDMs for specifying refactorings, we have an approach based on graph transformation to handle the PG-based refactoring scenario of the challenge. In addition, the visual specification style facilitates the understanding of the refactoring conditions and operations. Naturally, the resulting generated Java code might fall short in terms of performance if compared to an equivalent hand-written implementation from an experienced Java developer.

4 Conclusion and Future Work

In this paper, we presented our solution for the object-oriented Java refactoring case study of the Transformation Tool Contest 2015. Our solution is implemented using the eMoflon meta-modeling and graph transformation tool, developed at the Real-Time Systems Lab of the TU Darmstadt.

We conclude that both of the transformation languages supported by eMoflon, namely TGGs and SDMs can be utilized for different subtasks of the required transformation chain. TGGs in eMoflon also provide a synchronization algorithm which makes eMoflon a highly adequate tool to deal with bidirectional model synchronization problems similar to the one described in the challenge. With SDMs, we have the possibility to specify the actual refactoring operations in a visual and graph-based manner. (For more information about the difference between TGG and SDM as well as their interplay in the present refactoring scenario, we refer the interested reader to [7].)

Our future work includes the examination of the tool MoDisco[3] (having similar functionality to JaMoPP) in order to potentially reduce the need for pre- and postprocessing and to define a more structured and sophisticated TGG. Moreover, we would like to conduct experiments on real-life Java inputs to evaluate the practical relevance of our approach.

References

[1] T. Fischer, J. Niere, L. Torunski & A. Zündorf (2000): *Story Diagrams: A New Graph Grammar Language Based on the Unified Modelling Language and Java*. In: *TAGT*, *LNCS* 1764, Springer, pp. 157–167.

[2] F. Heidenreich, J. Johannes, M. Seifert & C. Wende (2010): *Closing the Gap between Modelling and Java*. In: *Software Language Engineering*, *LNCS* 5969, Springer, pp. 374–383.

[3] G. Kulcsár, S. Peldszus & M. Lochau: *Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation*. In: *Transformation Tool Contest 2015*. Available at `https://github.com/Echtzeitsysteme/java-refactoring-ttc/`.

[4] E. Leblebici, A. Anjorin & A. Schürr (2014): *Developing eMoflon with eMoflon*. In: *Theory and Practice of Model Transformations*, *LNCS* 8568, Springer, pp. 138–145.

[5] S. Peldszus, G. Kulcsár & M. Lochau (2015): *SHARE Image*. Available at `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=XP-TUe_TTC15-Refactoring.vdi`.

[6] S. Peldszus, G. Kulcsár & M. Lochau (2015): *Source Code at GitHub*. Available at `https://github.com/SvenPeldszus/GravityTTC`.

[7] S. Peldszus, G. Kulcsár, M. Lochau & S. Schulze (2015): *Incremental Co-Evolution of Java Programs Based on Bidirectional Graph Transformation*. In: *PPPJ'15*, ACM, NY, USA, pp. 138–151.

[8] A. Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In: *20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, *LNCS* 903, Springer, pp. 151–163.

[3] https://eclipse.org/MoDisco/

The SDMLib solution to the Java Refactoring case for TTC2015

Olaf Gunkel, Matthias Schmidt, Albert Zündorf
Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, 34121 Kassel, Germany

olaf.gunkel|matthias.schmidt|zuendorf@cs.uni-kassel.de

The Solution is hosted under https://bitbucket.org/mschmidt987/java-refactoring-case-ttc-2015-solution-fg-se-uni-kassel

This paper describes the SDMLib solution to the Java Refactoring case for TTC2015 [2]. SDMLib provides a mechanism for generating an abstraction model of a provided java program. In addition, SDMLib provides code generation that transforms the whole model or parts of it into java code. Thus, for the Java Refactoring case we just added a *Refactorer* that reads a java project and transforms the program graph according to the intended refactorings. These transformations are collected and applied to the source code by the SDMLib generator afterwards.

1 Introduction

Two of our studentical assistants found this case very interesting, because they plan to realize a related case in their master thesis. Their idea is to find bad smells and other structures that should be replaced by a design pattern implementation. After this detection of such places, the replacement should be applied by an automatic refactoring. The implementation of the TTC 2015 refactoring case gave them the chance to have a look on implementing refactorings and estimate the complexity of such code replacement operations.
Furthermore, our team gives a lecture in Graph Engineering at the University of Kassel in which we teach master grade students about the theoretical definition of graphs and practical approaches of graph matching and transformation operations. In addition, we teach them to implement a graph matching algorithm to perform transformations on the previously implemented generic graph. So we are familiar with several graph transformation techniques and interested in tasks that can be solved with them.
In previous work we already addressed the problem of parsing and generating java source code. To solve this, we added some features to our tool SDMLib. It is able to represent parsed code into a class model, that holds enough information to generate updated code afterwards (without changing the present code where it is not needed). We expected that to be a benefit for us when solving this case.
To address the Java Refactoring case, we used the introduced parser of SDMLib to create the program graph before the refactoring. Then we have built a new component to realise the refactorings in the graph. This component uses property change mechanisms to record the changes of the program graph and refactors the source code by calling the SDMLib generator afterwards.

2 SDMLib support for source code abstraction and generation

Transforming java source code into an abstract model is a complex task that can be accomplished by using a powerful parser. To solve this, SDMLib provides a recursive descent parser that analyzes java source code files and create an abstract graph model. Using the parser is really easy due to the fact that, as shown in Listing 1, the source folder and the package name (of the program that should be abstracted) is required.

```
1  public void updateFromCode(String srcFolder, String packageName){...}
```

Listing 1: Signature of the method that calls the parser for java programs

After parsing the source code, SDMLib provides a model that contains all information required for the refactoring case. The parts of the model, which we use to solve the case, can be seen in Figure 1.

The SDMLib model provides nearly every information that we need for the case. The only missing information, which is still missing in the solution, is the *access*-assoziation of class *TMember* as shown in figure 2 of the case description[1]. Despite the fact that the whole model represents complex program structures, it is comfortable, easy to use and enabled us to fullfill the requirements of the given tasks rapidly.

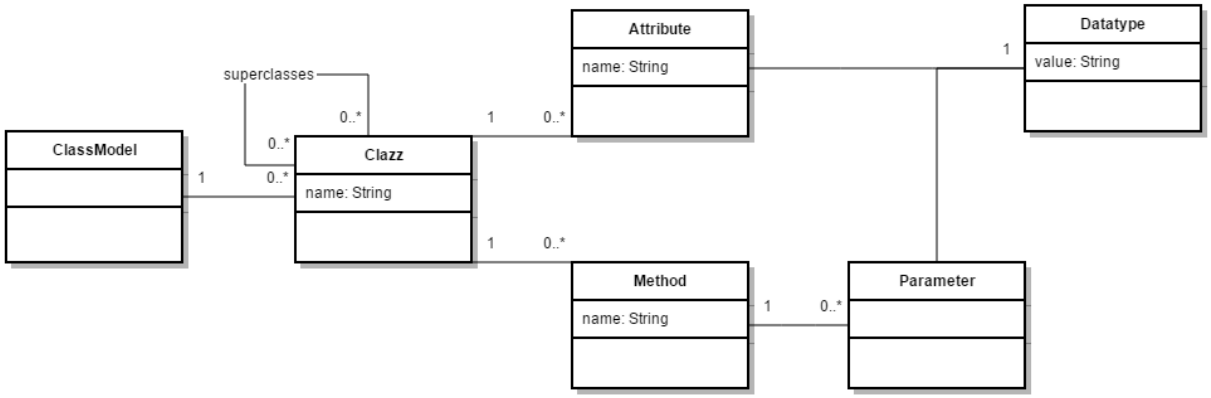


Figure 1: Cut of the source code abstraction model

To push our graph changes into the code, SDMLib supports us with its generator, that updates the parsed code. After creating a *ClassModel* by parsing a java project, every included class has its own parser instance, held by the *ClassModel*. The parsers are holding all relevant information about their class. For example, they have symbol tables in which, for every member, information about its position in the sourcecode are stored. By using this position information, its possible to extract, replace and insert parts of the sourcecode. Because of this relation, we can use the symbol table to delete, move or insert members in the source code. Listing 2 shows how to delete a member from the source file of a class. After replacing entries in the class, we set the boolean field *fileChanged* to *true* and commit the changes to the generating class *CGUtil*. Its *printFile(Parser)* Method writes the changes into the source code files.

```
1   SymTabEntry memberToDelSTE = clazzParser.getSymTabEntry(delMember);
2
3   clazzParser.replace(memberToDelSTE.getStartPos(),
4           memberToDelSTE.getEndPos()+1, "");
5
6   clazzParser.withFileChanged(true);
7
8   CGUtil.printFile(clazzParser);
```

Listing 2: How to push changes to the source code with SDMLib

3 Solving the Java refactoring case with SDMLib

Our solution covers the three major transformation steps (code to program graph, program graph refactoring and program graph to code) with support for create class-, pull up method-, pull up field- and extract superclass refactoring.
SDMLib already contains a mechanism to transform code into a program graph. So this part was quite easy to implement. The method *createModelFromSource* in Listing 3 shows how SDMLib can be used to generate a model out of given java source code. Just the path to the project is necessary.

```
1 public ClassModel createProgrammGraph(String pathToProject)
2 {
3
4    return refactorer.createModelFromSource(pathToProject);
5
6 }
```

Listing 3: Creating a object model from source code in a given package path

The resulted program graph now must be transformed according to the intended refactoring. Our algorithm is split into two parts here. The first part validates that the refactoring can be applied on the given object structure. For example a pull up method refactoring requires, that all child classes contain the method with the right signature. This requirement is checked for a valid match. The second step executes the graph transformation for the refactoring. Figure 2 shows an example situation for the pull up method refactoring. The method of the first child that should be pulled up gets his class relation changed to the parent. Furthermore we remove the matching methods of all other kids from the graph.
To complete the last step, we decided to add property change listeners to all relevant members of the object model. These are the methods, classes and attributes, because the refactorings cause changes to them. Our aim was to trace the changes. After the refactoring, the generator of SDMLib applies the traced transformations to the source code. For example our so called *ClazzSuperClassPropertyFileListener* reacts on changes of the inheritance field of a class. If a new superclass is set, this listener saves an object of the *ClazzSuperClazzPropertyFileChangeStep* Class in a *Queue*. This queue contains all events with their relevant information. To synchronize model and code, we execute all source code transformations according to the previous done model transformations. In this example, the generator changes the *extends* clauses of the affected classes or generates a new superclass.
Overall this case was made for us, because SDMLib already had many features to help us creating a program

graph and updating the appropriate java source code. Especially the parser and the generator of SDMLib helped to complete these tasks. Furthermore the resulting program graph fullfilled all our needs for the refactorings.

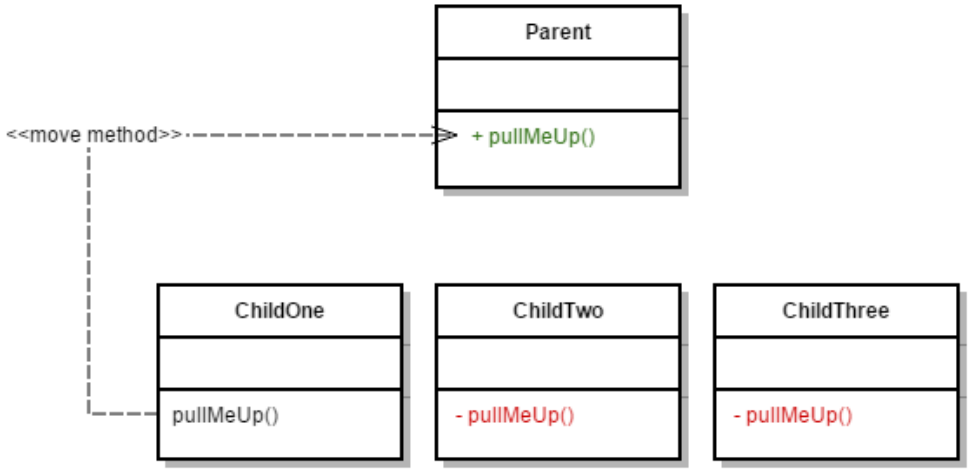


Figure 2: Example Graph Transformation for Pull Up Method Refactoring

4 Accomplished testcases

In Table 1, all execution times und the result of the given cases are presented. Except of one hidden case, our program succeeds in all tests. The one that fails contains a test where a method of two child classes should not be pulled up, because one of them is accessing a field, that the other one do not have. Our program fails here, because our tool does not analyse the semantic of method bodies. So there are no access edges in our program graph.
By writing additional test cases, we make sure to cover many other cases. The pull up refactoring ensures that the parent class is available. Furthermore we detect whether the pull up method or field is already defined in it, that it has childs and that all childs own the method or field with the right set of parameters. The create superclass refactorer also filters out the corner cases. It ensures that the superclass is not already existing. In addition, the refactoring fails with a response if not all chosen classes have the same superclass.

Case	Time(s)	Result
pub_ pum3_ 1	0	SUCCESS
hidden_ csc3_ 1a	0,003	SUCCESS
hidden_ csc1_ 2	0,001	SUCCESS
pub_ pum1_ 1_ paper1	0,005	SUCCESS
hidden_ csc1_ 1	0,007	SUCCESS
pub_ csc1_ 2	0,003	SUCCESS
hidden_ pum1_ 2	0,001	SUCCESS
pub_ csc1_ 1	0,005	SUCCESS
hidden_ pum1_ 1	0,002	FAILURE
hidden_ csc2_ 1	0,002	SUCCESS
pub_ pum1_ 2	0	SUCCESS
hidden_ pum2_ 2	0	SUCCESS
hidden_ pum2_ 1	0,002	SUCCESS
hidden_ csc3_ 1	0,006	SUCCESS
pub_ pum2_ 1	0,001	SUCCESS

Table 1: Execution time of all given test cases

5 Summary

Overall this case was easy for SDMLib as SDMLib already had many features helping us creating a class model graph and updating the appropriate java source code. Especially the parser and the generator of SDMLib helped to complete these tasks. The SDMLib parser and code generator are designed for simplicity. Thus, by default we do NOT use an abstract syntax tree for method bodies. Due to our experience with code generation in the Fujaba project, abstract syntax trees are very large and detailed and it is very tedious to maintain and modify them. For usual class model creation and manipulation, the analysis of method body is not necessary. And code generation for method bodies is much easier done using a template based approach. However without the abstract syntax tree of method bodies, certain refactorings like renaming an attribute or a method cannot be done. For such cases, the parser of SDMLib needs to be extended and the code generation must be able to replace single name tokens. Still we believe that for the manipulation of the program text, a template based approach and the replacement of text fragments is much easier than manipulating an abstract syntax tree.

References

[1] M. L. Géza Kulcsár, Sven Peldszus. Case Study: Object-oriented Refactoring of Java Programs using Graph Transformation.

[2] Object-oriented Refactoring of Java Programs using Graph Transformation (TTC'2015). https://github.com/Echtzeitsysteme/java-refactoring-ttc, 2015.

Part III.
The Train Benchmark Case

The TTC 2015 Train Benchmark Case for Incremental Model Validation*

Gábor Szárnyas Oszkár Semeráth István Ráth Dániel Varró

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
{szarnyas, semerath, rath, varro}@mit.bme.hu

In model-driven development of safety-critical systems (like automotive, avionics or railways), well-formedness of models is repeatedly validated in order to detect design flaws as early as possible. Validation rules are often implemented by a large amount of imperative model traversal code which makes those rule implementations complicated and hard to maintain. Additionally as models are rapidly increasing in size and complexity, efficient execution of these operations is challenging for the currently available toolchains. However, checking well-formedness constraints can be interpreted as evaluation of model queries, and the operations as model transformations, where the validation task can be specified in a concise way, and executed efficiently.

This paper presents a benchmark case and an evaluation framework to systematically assess the scalability of validating and revalidating well-formedness constraints over large models. The benchmark case defines a typical well-formedness validation scenario in the railway domain including the metamodel, an instance model generator, and a set of well-formedness constraints captured by queries and repair operations (imitating the work of systems engineers by model transformations). The benchmark case focuses on the execution time of the query evaluations with a special emphasis on reevaluations, as well as simple repair transformations.

1 Introduction

During the development of safety critical software like automotive, avionics or train control systems, different kind of models are frequently used. The goal of this approach is to develop models to assist the automated generation of various design artifacts (source code, configuration files, etc.) However, design errors of the system model invalidate the correctness of the generated artifacts, thus it is critical to check the well-formedness of such models. Additionally, it is considerably more expensive to fix design flaws in the later stage of the development, thus it is important to detect them as soon as possible by checking the well-formedness constraints repeatedly.

Model validation problems are often addressed by model transformation engines: error cases are defined by model queries, the results of which can be automatically repaired by transformation steps. In practice, this is challenging due to two factors: (i) *instance model sizes* are exhibiting a tremendous growth as the complexity of systems-under-design is increasing, (ii) the *sophistication of validation constraints* in toolchains is increasing. As a consequence, validation of industrial models is challenging or may become completely unfeasible.

To address this challenge, the Train Benchmark is a macro benchmark that aims to measure repetitive query evaluation performance. While there are a number of existing benchmarks for queries over relational databasesand triplestores, modeling tool workloads for well-formedness constraint validation

*This work was partially supported by the MONDO (EU ICT-611125) project and Red Hat Inc.

are significantly different [2]. Specifically, modeling tools use much more complex queries than typical transactional systems, and the real world performance is more affected by response time (i.e. execution time for a specific operation such as validation or transformation) rather than throughput (i.e. the number of parallel transactions). Also, previous TTC cases did not focus on measuring the performance of query reevaluation.

The source code is available at `https://github.com/FTSRG/trainbenchmark-ttc`. This case is strongly based on the Train Benchmark [1], an ongoing benchmark project of our research group.

2 Case Description

A *benchmark case* configuration in the Train Benchmark consists of an *instance model* (Section 2.2), a *query* and a *repair transformation* (Section 3) describing constraint violating elements. As a result of a benchmark case run, the *execution times* of each phase, the *memory usage* and the *number of invalid elements* are measured and recorded. The number of invalid elements are used to check the correctness of the validation, however the collection of element identifiers must also be available for later processing.

2.1 Metamodel

The metamodel of the Train Benchmark is shown in Figure 3. A train route is defined by a sequence of sensors. Sensors are associated with track elements which are either segments (with a specific length) or switches. A route follows certain switch positions which describe the *required* state of a switch belonging to the route. Different route definitions can specify different states for a specific switch. Each route has a semaphore on its entry and exit. Figure 1 shows a typical railway network.

Every railway element is a subtype of the class RailwayElement which has a unique identifier (id). The root of the model is a RailwayContainer which contains the semaphores and the routes of the model. Additionally, the railway container has an invalids reference for storing elements. This is used for serializing EMF models (Section B.1.1).

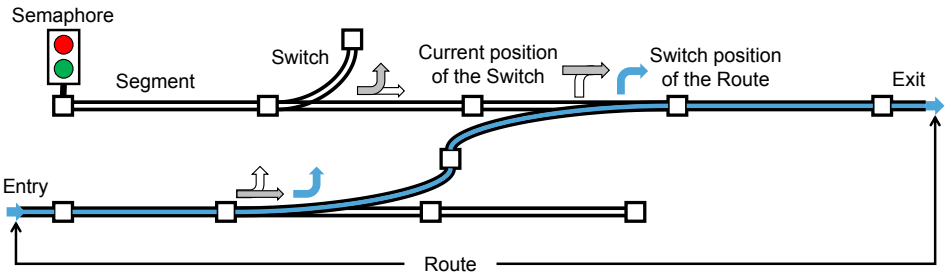


Figure 1: Illustration for the concepts in the Train Benchmark models.

2.2 Instance Models

The instance models are systematically generated for the metamodel: small model fragments are created and connected to each other. Based on the model queries, the generator injects errors to the model by removing edges and changing attribute values with a certain probability. The probability of injecting an error to violate a pattern (Section 3) is shown in Table 1.

This generation method controls the number of matches of all defined model queries. To avoid highly symmetric models, the exact number of elements and cardinalities are randomized. This brings

artificially generated models closer to real world instances and prevents query tools from abusing the artificial regularity of the model. To assess scalability, the benchmark uses instance models of growing sizes, each model containing twice as many model elements as the previous one. The instance models are designated by powers of two (1, 2, 4, 8, . . .), the smallest model containing about 5000 model elements.

2.3 Benchmark Phases

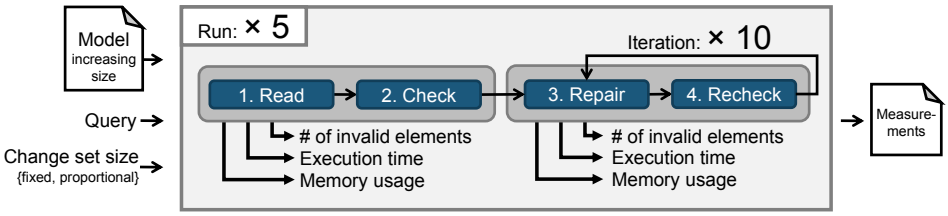


Figure 2: Phases of the benchmark.

To simulate a typical validation workload, four *phases* were defined (Figure 2).

1. During the read phase, the instance model is loaded from hard drive to memory. This includes the parsing of the input as well as initializing data structures (e.g. indexes) of the tool.
2. In the check phase, the instance model is queried to identify invalid elements. The result of this phase is a set of the invalid elements, which will be used in the next phase.
3. In the repair phase, the model is changed to simulate the effects (and measure the performance) of model modifying operations. The transformations are always performed on a subset of the model elements returned by the check phase.
4. The revalidation of the model is carried out in the recheck phase similarly to the check phase. In real-world scenarios, there are often multiple transformations in the system which may interfere with the results of the query. Because of this, we require the tools to reevaluate the query with regards to the current state of the model.

The repair operation intends to fix invalid models elements based on the invalid objects identified during the previous check or recheck phase. We defined two strategies to determine the size of the change set:

fixed 10 of invalid model elements is modified. This tests the efficiency of handling small change sets.

proportional 10% of the dresult set is modified. This tests the efficiency of handling large change sets.

2.4 Queries

The *queries* used in the validation scenario are introduced both informally and as graph patterns. In complexity, the queries range from simple attribute value checks to complex path constraints consisting of several join operations: two simple queries use at most 2 objects (PosLength and SwitchSensor) and three complex queries use 4–8 objects and multiple join operations (RouteSensor, SemaphoreNeighbor, SwitchSet).

2.4.1 Graph Patterns and Transformations

The purpose of the queries is to check well-formedness constraints by matching graph patterns looking for errors in the model. The *graph patterns* are defined by a *name*, a list of symbolic object parameters and the constraints to be satisfied by the parameters. A *pattern match* maps each symbolic parameter

to a model object, where the mapping satisfies the conditions defined by the constraints. The result of the query is the set of all possible matches. The absence of pattern matches means that the model is well-formed, and the matches of the error pattern marks the invalid elements. The *match set* contains all matches for a given pattern.

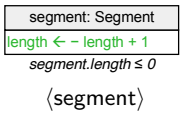
In the *repair* phase, some model elements are *deterministically selected* and repaired. In order to ensure *repeatable results*, (1) the elements for transformation are chosen using a pseudorandom generator, (2) the elements are always selected from the deterministically sorted list (Section 2.4).

3 Tasks

For each task, we present the well-formedness constraints. The *queries* are looking for violations of these constraints. We describe the meaning and the goal of each query and show a graphical notation of the associated graph pattern. We also define the matches as tuples to ensure that the ordering of the matches is consistent between the implementations (Section B.2). The *repair transformations* are represented as graph transformations. For defining the patterns and transformations, we used a graphical syntax similar to GROOVE [3] with a couple of additions:

- Filter conditions are shown in *italic* font.
- Negative application conditions are shown with in a red rectangle with the NEG caption.
- The insertions are with a «new» caption. Attribute updates are also show in green.

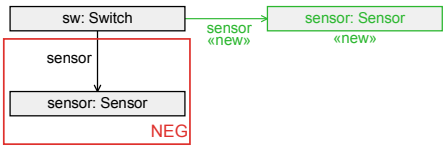
PosLength. Every segment must have a positive length.
Query. The query checks for segments with a length less than or equal to zero.



⟨segment⟩

Repair transformation. The length attribute of the segment in the match is updated to $-\mathsf{length}+1$.
Goal. This query defines an attribute check. This is a common use case in validation scenarios.

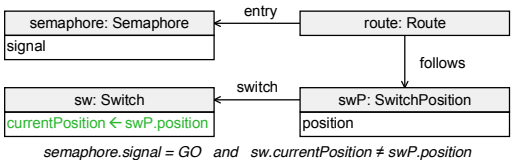
SwitchSensor. Every switch must have at least one sensor connected to it.
Query. The query checks for switches that have no sensors associated with them.
Repair transformation. A sensor is created and connected to the switch.



⟨sw⟩

Goal. This query checks whether an object is connected to a relation. This pattern is common in more complex queries, e.g. it is used in the RouteSensor and the SemaphoreNeighbor queries.

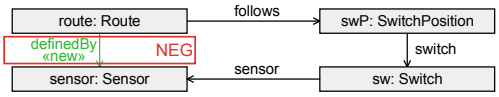
SwitchSet. The entry semaphore of a route may only show GO if all switches along the route are in the position prescribed by the route.
Query. The query checks for routes which have a semaphore that show the GO signal. Additionally, the route follows a switch position (swP) that is connected to a switch (sw), but the switch position (swP.position) defines a different position from the current position of the switch (sw.currentPosition).



⟨semaphore, route, swP, sw⟩

Repair transformation. The currentPosition attribute of the switch is set to the position of swP.
Goal. This pattern tests the efficiency of the join and filtering operations.

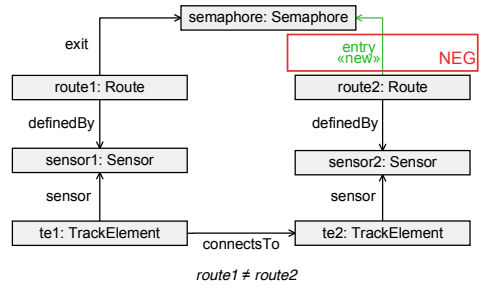
RouteSensor. All sensors that are associated with a switch that belongs to a route must also be associated directly with the same route.
Query. The query looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route.



⟨route, sensor, swP, sw⟩

Repair transformation. The missing definedBy edge is inserted by connecting the route in the match to the sensor.
Goal. This pattern checks for the absence of circles, so the efficiency of the join and the antijoin operations is tested.

SemaphoreNeighbor. Routes that are connected through sensors and track elements must belong to the same semaphore.
Query. The query checks for routes (route1) which have an exit semaphore (semaphore) and a sensor (sensor1) connected to a track element (te1). This track element is connected to another track element (te2) which is connected to another sensor (sensor2) which (partially) defines another, different route (route2), while the semaphore is not on the entry of this route (route2).
Repair transformation. The route2 node is connected to the semaphore node with an entry edge.



⟨semaphore, route1, route2, sensor1, sensor2, te1, te2⟩

Goal. This pattern checks for the absence of circles, so the efficiency of the join operation is tested. One-way navigable references are also present in the constraint, so the efficiency of their evaluation is also measured. Subsumption inference is required, as the two track elements can be switches or segments.

Acknowledgements. The authors would like to thank Benedek Izsó for originally designing and implementing the Train Benchmark, Tassilo Horn for providing valuable comments regarding both the specification of the case and the implementation of the benchmark framework, and Zsolt Kővári for his contributions in the benchmark and visualization scripts.

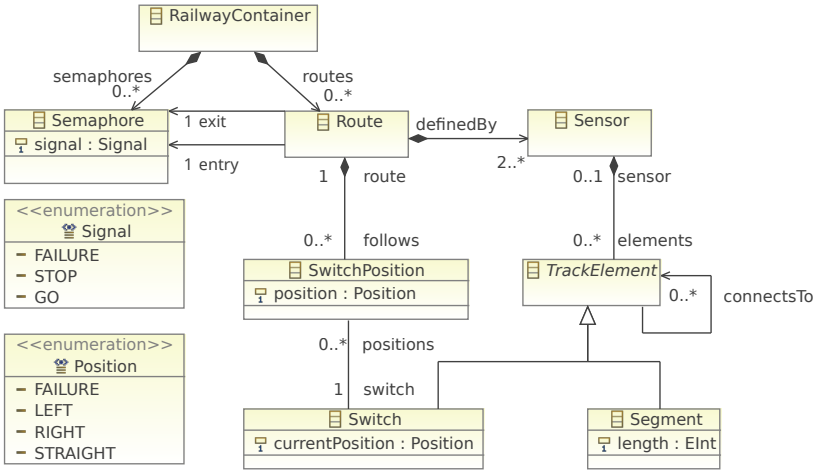
References

[1] Benedek Izsó, Gábor Szárnyas & István Ráth (2014): *Train Benchmark*. Technical Report, Budapest University of Technology and Economics.

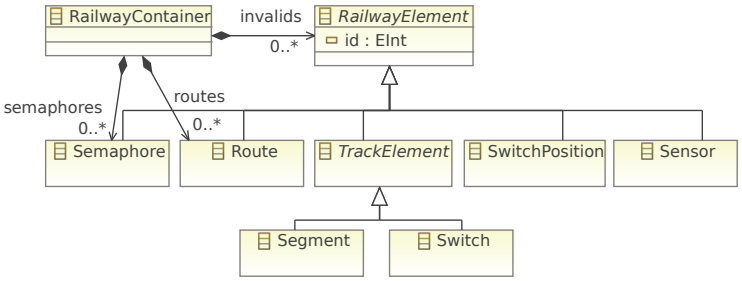
[2] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth & István Ráth (2013): *Towards Precise Metrics for Predicting Graph Query Performance*. In: *ASE 2013*, IEEE, pp. 412–431, doi:10.1109/ASE.2013.6693100.

[3] Arend Rensink (2004): *The GROOVE simulator: A tool for state space generation*. In: *Applications of Graph Transformations with Industrial Relevance*, Springer, pp. 479–485.

A Metamodel



(a) Containment hierarchy and references



(b) Supertype relations

Figure 3: The metamodel of the Train Benchmark.

B Implementation

To aid the development of case solutions, we provide a framework using predefined input and output formats, along with two reference implementations.

B.1 Instance Model Formats

B.1.1 EMF Models

The EMF models are serialized to standard XMI format using the generated EMF code. The injection of errors during the instance model generation (Section 2.2) causes some containment errors. Invalid elements violating the containment hierarchy could not be serialized. As the benchmark requires invalid models, the invalid elements are connected to the root element of the instance model by the invalids reference Figure 3b.

attribute / edge	error percentage
Segment.length	6%
Route.definedBy	10%
Route.exit	15%
Switch.sensor	35%
SwitchPosition.position	30%

Table 1: Error percentages in the generated instance model.

B.1.2 Non-EMF Models

The generator defines a graph-like interface for creating the models. The EMF model generator is an implementation of this interface. To generate non-EMF models, the following approaches are recommended: (1) either create a custom class which implements the Generator interface or (2) generate the EMF models and convert them to another representation.

B.2 Ordering of the Match Set

The matches in the match set may be returned in any collection (e.g. a list or a set) in any order, given that the collection is unique. In order to ensure that the benchmark is *repeatable*, this collection is copied to a sorted list. The sorting is carried out using by defining the ordering between matches.

To *compare* matches $M_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $M_2 = \langle b_1, b_2, \ldots b_n \rangle$, we take the first elements in each match (a_1 and b_1) and compare their identifiers. If the first elements are equal, we compare the second elements (a_2 and b_2) and so on until we find two different model elements. This is guaranteed by the fact that the collection is unique, so it cannot contain two identical matches.

For example, for the RouteSensor query, a match set may be returned by tool *A* as list

$$(\langle \mathsf{route}:8, \mathsf{sensor}:12, \mathsf{switchPosition}:4, \mathsf{sw}:10 \rangle; \langle \mathsf{route}:5, \mathsf{sensor}:1, \mathsf{switchPosition}:13, \mathsf{sw}:7 \rangle)$$

and by tool *B* as set

$$\{\langle \mathsf{route}:5, \mathsf{sensor}:1, \mathsf{switchPosition}:13, \mathsf{sw}:7 \rangle; \langle \mathsf{route}:8, \mathsf{sensor}:12, \mathsf{switchPosition}:4, \mathsf{sw}:10 \rangle\}$$

For both implementations, the framework creates a *sorted copy*, resulting in the list

$$(\langle \mathsf{route}:5, \mathsf{sensor}:1, \mathsf{switchPosition}:13, \mathsf{sw}:7 \rangle; \langle \mathsf{route}:8, \mathsf{sensor}:12, \mathsf{switchPosition}:4, \mathsf{sw}:10 \rangle)$$

The ordered list is also used to ensure that the transformations are performed on the same model elements, regardless of the return order of the match set.

B.3 Building the Projects

The Train Benchmark case defines a framework and application programming interface that enables the integration of additional tools. The reference implementation contains a benchmark suite for queries implemented in Java and EMF-IncQuery. Both the framework and the reference implementations are written in Java 7.

For building the projects, we used Apache Maven[1], one of the most widely used Java build systems. The build is configured so that the binaries are able to run without an Eclipse application. A significant

[1] `https://maven.apache.org/`

proportion of modeling tools are integrated to the Eclipse plug-in environment. In order to support such systems, our projects also have a plug-in nature. This way, they can be integrated with Eclipse (and OSGi) plug-ins as well and can be built without Maven.

B.4 Running the Projects

The scripts can parametrized by a simple JSON configuration file which defines:

- the range of the instance models from `minSize` to `maxSize`,
- the list of queries specified (Section 2.4),
- the list of tools,
- the number of runs,
- the number of repair–recheck iterations,
- the change set strategies,
- the JVM arguments (e.g. maximum heap memory).

The default configuration is stored in the `config/config.json` file. Please use this as a basis for your configuration.

```
{
  "MinSize": 1,
  "MaxSize": 2,
  "Queries": ["PosLength", "RouteSensor", "SwitchSensor", "SwitchSet", "SemaphoreNeighbor"],
  "Tools": [<your tool>],
  "ChangeSets": ["fixed", "proportional"],
  "Runs": 1,
  "IterationCount": 5,
  "JVM": {"vmargs": "-Xmx4G"}
}
```

B.5 Interpreting the Output

Measurements are automatically recorded by our benchmark framework and stored in TSV (Tab-Separated Values) format. This can be used to automatically create diagrams with the provided R[2] script and provide comparable plots. For publishing performance results, please stick to the format generated by the framework.

Table 2 shows an example output. The ChangeSet defines the change set size (*fixed* or *proportional*, see Figure 2). The Train Benchmark is executed 5 times, the index of the current run is stored in the RunIndex attribute. The Query is executed by the Tool on the model with the given Size. The validation errors are repaired in multiple iterations, the index of the current iteration is shown in the Iteration attribute. Multiple values (MetricValue) of different metrics (MetricName) are measured during the benchmark. The execution time (*time*) and memory consumption (*memory*) for the *read*, *check*, *repair* and *recheck* phases are collected. The name the current phase is defined by the PhaseName. Additionally, the result set size (*rss*) is stored for the *check* phase and the iterations in the *recheck* phase.

C Evaluation Criteria

The solutions are checked and evaluated for functional, usability and performance aspects.

[2] `https://www.r-project.org/`

C.1 Correctness and Completeness of Model Queries and Transformations

The goal of the correctness check is to determine if the different model query and transformation tasks are correctly and fully implemented in the submitted solutions. We provide the number of invalid model elements in several models detected by our reference implementation for each query and iteration step. If the result sizes are consistently equal, the solution is considered to be correct.

The expected results are available at `https://github.com/FTSRG/trainbenchmark-ttc/tree/master/expected-results`.

Each task is scored independently $0-3$ points by the following rules:

- **0 points:** The task is not solved.
- **$1-2$ points:** The task is partially solved, the solution provides the subset or the superset of the expected results.
- **3 points:** The task is completely and correctly solved.
- **-1 point:** Only the query is implemented, but the transformation is not.

Correctness and completeness: 5 tasks $\times$ 3 points $=$ 15 points

C.2 Conciseness

The validation rules are frequently changed and extended, therefore it is important to be able to define queries and transformations in a concise manner. These properties are scored based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more concise than it would be in a general-purpose imperative language (e.g. Java), or the task is partially solved and the result set needs additional processing.
- **2 points:** The task is solved, the query and the transformation is defined in a declarative, visual or other query language, but the specification is hard to formulate.
- **3 points:** The solution is compact, the query and the transformation are defined in a concise manner.
- **-1 point:** Either the query or the transformation is implemented.

Conciseness: 5 tasks $\times$ 3 points $=$ 15 points

C.3 Readability

The readability and descriptive power of each query and transformation is scored with respect to a model validation use case. The score represents how well model queries are used as model constraints, and how well repair operations can be expressed by model transformations. The score is given based on the following rules:

- **0 points:** The task is not solved.
- **1 point:** The task is solved, but the solution is not significantly more readable than it would be in a general-purpose imperative language (e.g. Java), or the task is just partially solved. For example, a typical EMF validator should get 1 point.

- **2 points:** The task is solved, the query and the transformation follows the description of the constraint and repair rule, but it is difficult to comprehend the meaning of the solution. For example, a foreign key constraint checked by a query formulated in SQL should get 2 points.
- **3 points:** The solution could be presented in the documentation of the modeling domain, and it is easier to comprehend than a textual description in natural language. For example, a solution similar to the graphical notation used in this paper should get 3 points.
- **−1 point:** If the language is only able to express either the constraint (e.g. OCL) or the repair operation.

Readability: 5 tasks × 3 points = 15 points

C.4 Performance on Large Models

The goal of the performance measurements is to check the applicability of the submitted solutions on large industrial models. During the performance tests the execution times will be measured for different scenarios and increasing model sizes.

Please restrict your benchmarks to those input models that can be processed within 5 minutes or less. Runs that take longer than 5 minutes will not be considered in the evaluation. Please provide a solution that can run on an x64-based Linux system with 4+ GB of memory, and that can be started on the command-line. This will be important to reproduce your results on a remote testing system. Please document the setup of your solution and the requirements to the system environment.

We defined two validation scenarios, based on the phases defined in Section 2.3:

batch The model is loaded (read) and validated (check).

repeated The model is loaded (read) and validated (check), then the model is edited (repair) and revalidated (recheck) 10 times.

The performance of the solutions are compared in 20 *tournaments*:

- The tournaments are calculated for the 5 tasks. If a solution skips a task, it is not considered in the tournament.
- Each solution is measured for both *batch* and *repeated* validation.
- Each solution is measured for both *fixed* and *proportional* change sets.

A solution gets from 0 to 1 points for a tournament which is launched for increasing model sizes. The score is based on the maximum size that the solution is able to handle, and its execution time relative to the fastest solution. Each measurement is executed 5 times and the median value is taken.

- The model size is increased as long as there is a solution that is able to solve it in the given time limit. This results in *rounds* $k = 1,2,3,\ldots,n$ for sizes 2^{k-1} $(1,2,4,\ldots,2^{n-1})$.
- For each tournament, a solution earns a score between 0 and 1, determined by

$$\frac{\sum_{k=1}^{n} score(k)}{\sum_{k=1}^{n} k},$$

where

$$score(k) = \begin{cases} score_{size}(k) \times score_{time}(k), & \text{if the solution runs correctly and within the given time limit,} \\ 0, & \text{if the solution fails to run correctly or exceeds the given time limit,} \end{cases}$$

and $\sum_{k=1}^{n} k = n \cdot (n+1)/2$ is used for normalizing the result.

- For each round k from 1 to n, if a solution is able to complete the validation, it is rewarded k points:
  - round 1 (size 1): the winner earns 1 point,
  - round 2 (size 2): the winner earns 2 points,
  - round 3 (size 4): the winner earns 3 points,
  - ...
  - round n (size 2^{n-1}): the winner earns n points.

  The formula is specified as:

$$score_{size}(k) = k$$

- The fastest solution in each round earns 1 point, the other solutions earn partial points, based on the proportion of the current solution's execution time to the fastest execution time. The logarithm of this ratio for base 2 defines the score. For example:
  - if a solution takes 2× as long, it earns ½ points,
  - if a solution takes 4× as long, it earns ⅓ points,
  - if a solution takes 8× as long, it earns ¼ points,
  - and so on.

  The formula is specified as:

$$score_{time}(k) = \frac{1}{1 + \log_2\left(\frac{\text{the solution's execution time in round } k}{\text{the fastest execution time in the round } k}\right)}$$

In conclusion, a solution earns up to 20 points for **performance**:

5 tasks × 2 validation scenarios × 2 change set sizes × up to 1 points = 20 points

C.5 Overall Evaluation

The scores of each aspect of the submitted solution are summarized to derive the final score (max. 65 points) used for ranking the submitted solutions.

ChangeSet	RunIndex	Tool	Size	Query	PhaseName	Iteration	MetricName	MetricValue
fixed	1	EMFIncQuery	1	PosLength	check	0	rss	43
fixed	1	EMFIncQuery	1	PosLength	recheck	1	rss	33
fixed	1	EMFIncQuery	1	PosLength	recheck	2	rss	23
fixed	1	EMFIncQuery	1	PosLength	recheck	3	rss	13
fixed	1	EMFIncQuery	1	PosLength	recheck	4	rss	3
fixed	1	EMFIncQuery	1	PosLength	recheck	5	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	6	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	7	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	8	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	9	rss	0
fixed	1	EMFIncQuery	1	PosLength	recheck	10	rss	0
fixed	1	EMFIncQuery	1	PosLength	read	0	time	754739233
fixed	1	EMFIncQuery	1	PosLength	read	0	memory	6711048
fixed	1	EMFIncQuery	1	PosLength	check	0	time	51752
fixed	1	EMFIncQuery	1	PosLength	check	0	memory	6582280
fixed	1	EMFIncQuery	1	PosLength	recheck	1	time	5116
fixed	1	EMFIncQuery	1	PosLength	recheck	1	memory	2848944
fixed	1	EMFIncQuery	1	PosLength	recheck	2	time	4304
fixed	1	EMFIncQuery	1	PosLength	recheck	2	memory	2823352
fixed	1	EMFIncQuery	1	PosLength	recheck	3	time	8533
fixed	1	EMFIncQuery	1	PosLength	recheck	3	memory	2798328
fixed	1	EMFIncQuery	1	PosLength	recheck	4	time	4362
fixed	1	EMFIncQuery	1	PosLength	recheck	4	memory	2781144
fixed	1	EMFIncQuery	1	PosLength	recheck	5	time	4086
fixed	1	EMFIncQuery	1	PosLength	recheck	5	memory	2780248
fixed	1	EMFIncQuery	1	PosLength	recheck	6	time	4723
fixed	1	EMFIncQuery	1	PosLength	recheck	6	memory	2780344
fixed	1	EMFIncQuery	1	PosLength	recheck	7	time	8350
fixed	1	EMFIncQuery	1	PosLength	recheck	7	memory	2780440
fixed	1	EMFIncQuery	1	PosLength	recheck	8	time	12007
fixed	1	EMFIncQuery	1	PosLength	recheck	8	memory	2780536
fixed	1	EMFIncQuery	1	PosLength	recheck	9	time	4107
fixed	1	EMFIncQuery	1	PosLength	recheck	9	memory	2780632
fixed	1	EMFIncQuery	1	PosLength	recheck	10	time	21459
fixed	1	EMFIncQuery	1	PosLength	recheck	10	memory	2780776
fixed	1	EMFIncQuery	1	PosLength	repair	1	time	2861134
fixed	1	EMFIncQuery	1	PosLength	repair	1	memory	2855192
fixed	1	EMFIncQuery	1	PosLength	repair	2	time	3558045
fixed	1	EMFIncQuery	1	PosLength	repair	2	memory	2824640
fixed	1	EMFIncQuery	1	PosLength	repair	3	time	1090021
fixed	1	EMFIncQuery	1	PosLength	repair	3	memory	2800656
fixed	1	EMFIncQuery	1	PosLength	repair	4	time	1062007
fixed	1	EMFIncQuery	1	PosLength	repair	4	memory	2781272
fixed	1	EMFIncQuery	1	PosLength	repair	5	time	1235721
fixed	1	EMFIncQuery	1	PosLength	repair	5	memory	2780336
fixed	1	EMFIncQuery	1	PosLength	repair	6	time	8123
fixed	1	EMFIncQuery	1	PosLength	repair	6	memory	2780360
fixed	1	EMFIncQuery	1	PosLength	repair	7	time	3636
fixed	1	EMFIncQuery	1	PosLength	repair	7	memory	2780456
fixed	1	EMFIncQuery	1	PosLength	repair	8	time	14451
fixed	1	EMFIncQuery	1	PosLength	repair	8	memory	2780552
fixed	1	EMFIncQuery	1	PosLength	repair	9	time	2880
fixed	1	EMFIncQuery	1	PosLength	repair	9	memory	2780648
fixed	1	EMFIncQuery	1	PosLength	repair	10	time	3767
fixed	1	EMFIncQuery	1	PosLength	repair	10	memory	2780744

Table 2: Example output of the benchmark measurements.

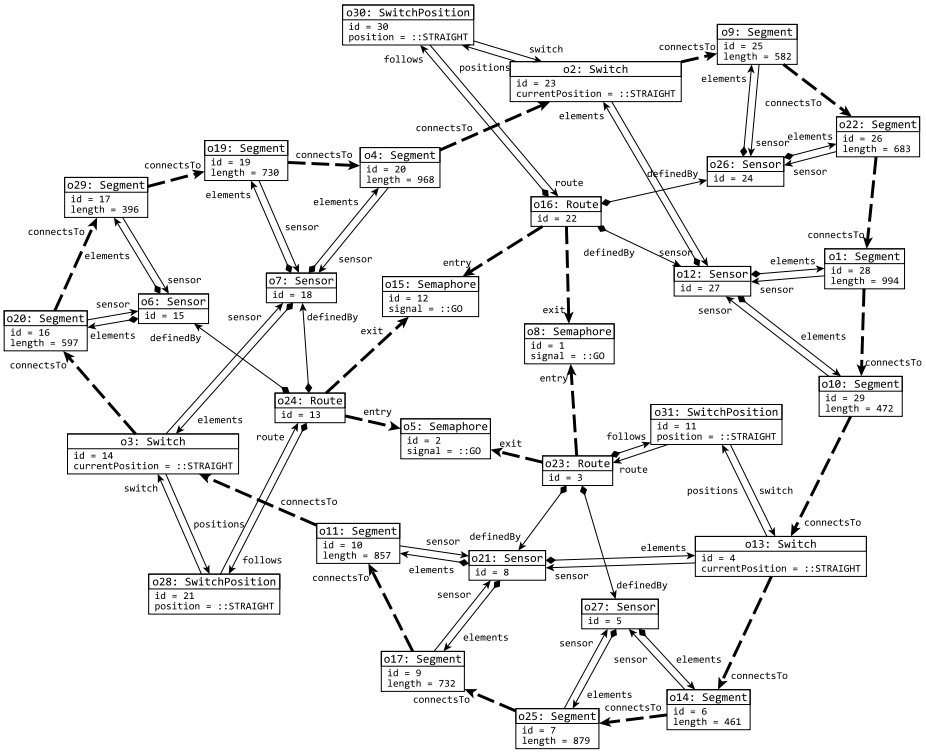


Figure 4: Well-formed railway instance model.

An NMF solution to the Train Benchmark Case at the TTC 2015

Georg Hinkel
Forschungszentrum Informatik (FZI)
Haid-und-Neu-Straße 10-14, Karlsruhe, Germany
hinkel@fzi.de

Lucia Happe
Karlsruhe Institute of Technology (KIT)
Am Fasanengarten 5, Karlsruhe, Germany
lucia.kapova@kit.edu

Model validation in model-driven development gains in importance as the systems grow in size and complexity. In this situation an efficiency of validation execution and an immediate feedback whether a recent manual edit operation broke a validation rule is desirable. To increase efficiency, incremental model validation tries to minimize the proportions of the model that have to be rechecked by reusing previous validation results. As a benchmark for efficiency of validation tools, the Train Benchmark Case at the Transformation Tool Contest 2015 was created. In this paper, we present a solution using NMF Expressions, a tool for incremental evaluation of arbitrary expressions on the .NET platform.

1 Introduction

This paper proposes a solution for the Train Benchmark Case[1] at the Transformation Tool Contest (TTC) 2015. Our solution is publicly available on CodePlex[1] and SHARE[2] and built upon the .NET Modeling Framework[3] (NMF) and especially on *NMF Expressions*[4]. NMF is a tool suite on the .NET platform to support model-driven engineering. Its metamodel NMeta is largely compatible with Ecore so that Ecore metamodels can be transformed to NMeta with a compliant XMI format, i.e. models according to an Ecore metamodel can be deserialized using the transformed NMeta metamodel.

NMF Expressions is designed for implicitly incremental evaluation of arbitrary (lambda calculus) expressions. This is done based on a theoretical foundation of representing incremental computation systems as a monad. The implicit approach means that developers specify the expressions in a batch mode whereas the incrementality is added through the monad. As a consequence, the syntax is very understandable as also remarked by the peer reviewers.

So far, few companies have adopted MDE as their main development paradigm with one of the major reasons being the lack of tool support [2], [3]. Developers are used to an excellent tool support for languages like Java or C# which many MDE tools cannot bear to meet. Furthermore, studies as e.g. by Meyerovich [4] suggest that developers only change their primary programming language when a project requires them to or they can reuse a large proportion of code. We see no reason why this should not extend to model validation tools and thus we are seeking for the ways to let developers specify these expressions in their primary languages.

Our goal is to hide the incrementality concerns from the developer, who only has to specify the validation expression, and automate the incrementalization of the validation expression, aiming for a declaritive usage of the C# language.

[1] http://ttc2015trainbenchmarknmf.codeplex.com
[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64-TTC15_NMF.vdi
[3] http://nmf.codeplex.com
[4] http://nmfexpressions.codeplex.com

In this paper, we evaluate the efficiency of incremental validation with *NMF Expressions*. The rest of this paper is structured as follows: Section 2 gives a very short introduction to *NMF Expressions* and Section 3 explains our solution. Finally Section 5 summarizes the paper.

2 NMF Expressions

The goal of *NMF Expressions* is to give developers an automated tool at hand providing them with advantages of incremental evaluation for arbitrary expressions. Unlike many other approaches, our approach works implicitly, so developers only have to specify their expressions and *NMF Expressions* takes care of how to turn this into an algorithm that will evaluate the expression in an incremental fashion. On the other hand, the traditional batch mode specification is still available so that *NMF Expressions* yields a choice whether to run a given expression incrementally or in batch mode.

In the incremental mode, the approach creates a dynamic dependency graph from a given expression and observes changes. These changes originate from elementary update notifications and are propagated through the dependency graph. Operating on the .NET platform, *NMF Expressions* uses the industry standard `INotifyPropertyChanged` and `INotifyCollectionChanged` interfaces to record elementary changes. These are also required by a lot of other tools including the modern UI libraries on the .NET platform. As a consequence of the theoretical foundation using monads, the dependency graph contains specialized nodes for optimized incrementalization of queries.

While *NMF Expressions* works with arbitrary model representations implementing the interfaces for elementary change propagation, we use the model representation of *NMF*. That is, we transformed the given Ecore metamodel of the railway domain into an NMeta metamodel and generated model representation code. *NMF* thus offers us a deserialization mechanism to load the resulting models as objects into memory.

3 Solution with NMF Expressions

The intended usage of *NMF Expressions* in incremental mode is that users would modify the model in some editor through a sequence of change operations, each of which providing elementary change notifications. Then, *NMF Expressions* would use the elementary change notifications and combine them to provide immediate feedback whether the most recent model manipulation has caused some validation rule to fail for some model elements. Currently, *NMF Expressions* always minimizes the model elements that it has to look at, even at the cost of high memory usage. However, in the Train Benchmark, the only model manipulations we can see are the repair operations, so for us the benchmark does not really reflect the situation for which we have designed *NMF Expressions*.

In incremental mode, *NMF Expressions* creates a cache for the selected expressions and maintains this cache. This maintenance happens automatically as *NMF Expressions* adds computational effort to the (in-memory) online model manipulation. In this case solution, we created expressions for the validation patterns so *NMF Expressions* caches the invalid elements continuously. However, this means that the phases drawn from the case description get blurred. In particular, the check phases get meaningless as the updated results are always available and could be used for immediate feedback, while more computational effort is put to the model manipulation such as the modify operations.

Because *NMF Expressions* allows to use the same specification both in a classic batch manner as also incrementally, our solution can also be configured to run in batch mode without any changes to the

PosLength

```
Fix(pattern: rc.Descendants().OfType<Segment>()
               .Where(seg => seg.Length <= 0),
   action: segment => segment.Length = -segment.Length + 1);
```

SwitchSensor

```
Fix(pattern: rc.Descendants().OfType<Switch>()
               .Where(sw => sw.Sensor == null),
   action: sw => sw.Sensor = new Sensor());
```

SwitchSet

```
var routes = rc.Routes.Concat(rc.Invalids.OfType<Route>());
Fix(pattern: from route in routes
             where route.Entry != null
               && route.Entry.Signal == Signal.GO
             from swP in route.Follows.OfType<SwitchPosition>()
             where swP.Switch.CurrentPosition != swP.Position
             select swP,
   action: swP => swP.Switch.CurrentPosition = swP.Position);
```

RouteSensor

```
Fix(pattern: from route in routes
             from swP in route.Follows.OfType<SwitchPosition>()
             where swP.Switch.Sensor != null &&
               !route.DefinedBy.Contains(swP.Switch.Sensor)
             select new { Route = route, Sensor = swP.Switch.Sensor },
    action: match => match.Route.DefinedBy.Add(match.Sensor),
```

SemaphoreNeighbor

```
Fix(pattern: from route1 in routes
             from route2 in routes
             where route2.Entry != route1.Exit
             from sensor1 in route1.DefinedBy
             from te1 in sensor1.Elements
             from te2 in te1.ConnectsTo
             where te2.Sensor == null
               || route2.DefinedBy.Contains(te2.Sensor)
             select new { Route = route2, Semaphore = route1.Exit },
    action: match => match.Route.Entry = match.Semaphore);
```

patterns. When executed in batch mode, *NMF Expressions* simply forwards the call to the LINQ to objects implementation. Besides a negligible runtime compilation effort, this utilizes the highly optimized platform LINQ implementation.

The patterns are **enumerable expressions** where developers can choose at runtime whether the pattern should be executed in batch mode or whether *NMF Expressions* should register for elementary change notifications to keep a cache of the result up to date. To specify patterns, we created a small method `Fix` that captures them.

```
public void Fix<T>(IEnumerableExpression<T> pattern, Action<T> action) {
  var patternInc = pattern.AsNotifiable();
  foreach (T element in patternInc) action(element);
  patternInc.CollectionChanged += (o,e) => {
    if (e.NewItems != null)
      foreach (T element in e.NewItems)
        action(element);
  }}
```

Listing 1: A simplified implementation of the `Fix` function

The easiest implementation for the `Fix` function repairing any validation error as soon as they occur

would be the one presented in Listing 1. In Line 2, we tell *NMF Expressions* that we want to obtain incremental updates for the given pattern. Line 3 repairs all occurences existing so far and Lines 4-8 handle new pattern matches. For the benchmark, we adopted the `Fix` function to account for the benchmark phases. In particular, the implemented version takes a third parameter to allow us to sort matches. Since these sort keys offer little insight, we omit them in the pattern presentation.

In the following we will present the solution to the tasks, following the structure of the case description, though with omitted sort keys.

Please note that the parameter names such as `pattern` or `action` are optional, we only included them for better understandability.

The solutions to *SwitchSet*, *RouteSensor* and *SemaphoreNeighbor* use the query syntax of C#. This syntax is translated to the method chaining syntax by mapping the query keywords like `from` or `where` to method calls of *NMF Expressions*. Such query expressions are commonality on the .NET platform and thus easy to write and understand by most developers.

Note that the order in which the statements occur does make a difference. In particular, e.g. lines 2 and 3 of the *SwitchSet* solution could logically be interchanged but cause a slightly different implementation. *NMF Expressions* currently does not optimize the query for performance.

In the solution for *SemaphoreNeighbor* we can observe that *NMF Expressions* is not able to inverse directed references. We argue that such inversion is always limited to a particular scope, which is unclear from the context. If the context was clear, the reference should have been navigable in both directions in the metamodel. As this is not the case, we have to cross join the two respective routes and filter them on the semaphores.

4 Evaluation

To evaluate our solution, we ran it in comparison to the reference implementations in Java and EMF-IncQuery [5]. The measurements were taken on a system with an Intel i5-4300U processor in a system equipped with 12GB RAM running on Windows 8.1 Pro, .NET 4.5 and Java 1.8 update 45. The results for recheck and repair are shown in Figure 1. The *SemaphoreNeighbor* ran out of memory for larger models. A discussion is omitted for space limitations.

In all four presented queries, both versions are up to multiple magnitudes faster than the plain Java solution. In medium-sized models, the incremental version also beats EMF-IncQuery, in the *SwitchSet* pattern it is even faster on the larger models.

5 Summary

In this paper, we presented an NMF solution to the Train Benchmark case at the TTC 2015.

The queries and repair transformations demonstrate why we have sticked to the C# language. We think that it is very hard to get a more concise textual solution for this case. At the same time, developers get the full tool support from e.g. Visual Studio and the query syntax that we use is used by thousands of developers already and widely understood.

The performance figures shows that the incremental version of our solution outperforms the batch mode execution of the same solution in all cases. At the same time, our solution yields a batch mode execution in cases where only a single analysis run is needed. This version outperforms the classic Java solution in several orders of magnitude. On the other hand, for large models our incremental solution

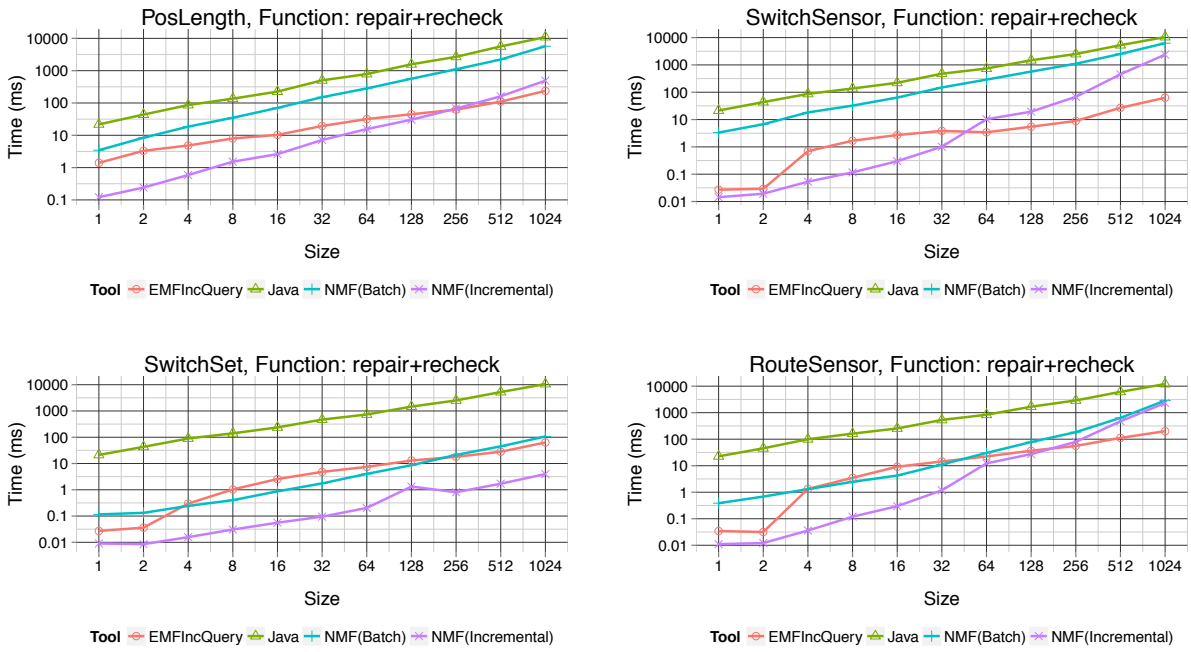


Figure 1: Performance Results for the NMF solution versions compared to the reference solutions in Java and EMF-IncQuery

cannot keep up with the EMF-IncQuery solution, except for one pattern where the NMF solution is slightly faster.

The biggest advantage of our solution is that it gives both a batch mode solution and an incremental solution our of the same pattern specifications. Thus, the same analysis code can be used in the case setting where incrementality is a clear advantage, or in a batch mode, e.g. when memory is a sparse resource or the analysis results are only required once.

References

[1] G. Szárnyas, O. Semeráth, I. Ráth, and D. Varró, "The TTC 2015 Train Benchmark Case for Incremental Model Validation," in *8th Transformation Tool Contest (TTC 2015)*, 2015.

[2] M. Staron, "Adopting model driven software development in industry–a case study at two companies," in *Model Driven Engineering Languages and Systems*, Springer, 2006, pp. 57–72.

[3] P. Mohagheghi, W. Gilani, A. Stefanescu, and M. A. Fernandez, "An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases," *Empirical Software Engineering*, vol. 18, no. 1, pp. 89–116, 2013.

[4] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, ACM, 2013, pp. 1–18.

[5] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös, "Incremental evaluation of model queries over emf models," in *Model Driven Engineering Languages and Systems*, Springer, 2010, pp. 76–90.

Solving the TTC Train Benchmark Case with FunnyQT

Tassilo Horn
Institute for Software Technology, University Koblenz-Landau, Germany
horn@uni-koblenz.de

This paper describes the FunnyQT solution to the TTC 2015 Train Benchmark transformation case. The solution solves all core and all extension tasks, and it won the *overall quality award*.

1 Introduction

This paper describes the FunnyQT[1] [1, 2] solution of the TTC 2015 Train Benchmark Case [3]. All core and extension tasks have been solved. The solution project is available on Github[2], and it is set up for easy reproduction on a SHARE image[3]. This solution won the *overall quality award* for this case.

FunnyQT is a model querying and transformation library for the functional Lisp dialect Clojure[4]. Queries and transformations are Clojure programs using the features provided by the FunnyQT API.

Clojure provides strong metaprogramming capabilities that are used by FunnyQT in order to define several *embedded domain-specific languages* (DSL) for different querying and transformation tasks.

FunnyQT is designed with extensibility in mind. By default, it supports EMF models and JGraLab TGraph models. Support for other modeling frameworks can be added without having to touch FunnyQT's internals.

The FunnyQT API is structured into several namespaces, each namespace providing constructs supporting concrete querying and transformation use-cases, e.g., model management, functional querying, polymorphic functions, relational querying, pattern matching, in-place transformations, out-place transformations, bidirectional transformations, and some more. For solving the train benchmark case, especially its in-place transformation DSL has been used.

2 Solution Description

In this section, the individual tasks are discussed one by one. They are all implemented as in-place transformation rules supported by FunnyQT's *funnyqt.in-place* transformation DSL. The rules' repair actions simply call the CRUD functions of the EMF-specific *funnyqt.emf* namespace.

Task 1: PosLength. The transformation rule realizing the *PosLength* task is given below.

```
1 (defrule pos-length {:forall true :recheck true} [g]
2   [segment<Segment>
3    :when (<= (eget-raw segment :length) 0)]
4   (eset! segment :length (inc (- (eget-raw segment :length)))))
```

[1] http://funnyqt.org
[2] https://github.com/tsdh/ttc15-train-benchmark-funnyqt
[3] The SHARE image name is ArchLinux64_TTC15-FunnyQT_2
[4] http://clojure.org

The `defrule` macro defines a new in-place transformation rule with the given name (`pos-length`), an optional map of options (`{:forall true, ...}`) a vector of formal parameters (`[g]`), a pattern (`[segment<Segment>...]`), and one or many actions to be applied to the pattern's matches (`(eset! ...)`). The first formal parameter must denote the model the rule is applied to, so here the argument `g` denotes the train model when the rule is applied using `(pos-length my-train-model)`.

The pattern matches a node called `segment` of metamodel class `Segment`. Additionally, the segment's length must be less or equal to zero as defined by the `:when` constraint. The action says that the segment's `length` attribute should be set to the incremented negation of the current length.

The normal semantics of applying a rule is to find one single match of the rule's pattern and then execute the rule's actions on the matched elements. The `:forall` option changes this behavior to finding all matches first, and then applying the actions to each match one after the other. FunnyQT automatically parallelizes the pattern matching process of such forall-rules under certain circumstances like the JVM having more than one CPU available and the pattern declaring at least two elements to be matched.

The `:recheck` option causes the rule to recheck if a pre-calculated match is still conforming the pattern just before executing the rule's actions on it. This can be needed for forall-rules whose actions possibly invalidate matches of the same rule's pattern, e.g., when the application of the action to a match m_i cause another match m_j to be no valid match any longer[5].

Task 2: SwitchSensor. The transformation rule realizing the *SwitchSensor* task is given below.

```
5 (defrule switch-sensor {:forall true :recheck true} [g]
6   [sw<Switch> -!<:sensor>-> <>]
7   (eset! sw :sensor (ecreate! nil 'Sensor)))
```

It matches a switch `sw` which is not contained by some sensor. The exclamation mark of the edge symbol `-!<:sensor>->` specifies that no such reference must exist, i.e., it specifies a negative application condition. The action fixes this problem by creating a new `Sensor` and assigning that to the switch `sw`.

Task 3: SwitchSet. The `switch-set` rule realizes the *SwitchSet* task. Its definition is given below.

```
8 (def Signal-GO (eenum-literal 'Signal.GO))

9 (defrule switch-set {:forall true :recheck true} [g]
10   [route<Route> -<:entry>-> semaphore
11    :when (= (eget-raw semaphore :signal) Signal-GO)
12    route -<:follows>-> swp -<:switch>-> sw
13    :let [swp-pos (eget-raw swp :position)]
14    :when (not= (eget-raw sw :currentPosition) swp-pos)]
15   (eset! sw :currentPosition swp-pos))
```

It matches a `route` with its entry `semaphore` where the semaphore's signal is `Signal.GO`. The route follows some switch position `swp` whose switch `sw`'s current position is different from that of the switch position. The fix is to set the switch's current position to the position of the switch position `swp`.

Note that there are no metamodel types specified for the elements `semaphore`, `swp`, and `sw` because those are already defined implicitly by the references leading to them, e.g., all elements referenced by a route's `follows` reference can only be instances of `SwitchPosition` according to the metamodel. FunnyQT doesn't require the transformation writer to encode tautologies in her patterns[6].

[5] This cannot happen for the `pos-length` rule, however the case description demands matches to be revalidated before applying the repair actions.

[6] In fact, if there are types specified, those will be checked. So omitting them when they are not needed also results in slightly faster patterns.

Extension Task 1: RouteSensor. The extension task *RouteSensor* is realized by the `route-sensor` rule given below.

```
16 (defrule route-sensor {:forall true :recheck true} [g]
17   [route<Route> -<:follows>-> swp -<:switch>-> sw
18    -<:sensor>-> sensor --!<> route]
19   (eadd! route :definedBy sensor))
```

It matches a `route` that follows some switch position `swp` whose switch `sw`'s `sensor` is not contained by the `route`. The repair action is to assign the `sensor` to the `route`.

Extension Task 2: SemaphoreNeighbor. The second and last extension task *SemaphoreNeighbor* is realized by the `semaphore-neighbor` rule defined as shown below.

```
20 (defrule semaphore-neighbor {:forall true :recheck true} [g]
21   [route1<Route> -<:exit>-> semaphore
22    route1 -<:definedBy>-> sensor1 -<:elements>-> te1
23    -<:connectsTo>-> te2 -<:sensor>-> sensor2
24    --<> route2<Route> -!<:entry>-> semaphore
25    :when (not= route1 route2)]
26   (eset! route2 :entry semaphore))
```

It matches a route `route1` which has an exit `semaphore`. Additionally, `route1` is defined by a sensor `sensor1` which contains some track element `te1` that connects to some track element `te2` whose sensor is `sensor2`. This `sensor2` is contained by some other route `route2` which does not have `semaphore` as entry semaphore. The fix is to set `route2`'s entry reference to `semaphore`.

2.1 Deferred Rule Actions

As mentioned above, the normal semantics of a forall-rule is to compute all matches of the rule's pattern first (possibly in parallel), and then apply the rule's actions on every match one after the other. However, the case description strictly separates the computation of matches from the repair transformations.

FunnyQT also provides stand-alone patterns. Using them, one could have defined patterns for finding occurrences of the five problematic situations in a train model, and separate functions for the repair actions where the latter receive one match of the corresponding pattern and fix that.

But for in-place transformation rules, FunnyQT also provides *rule application modifiers*. Concretely, any in-place transformation rule `r` can be called as `(as-pattern (r model))` in which case it behaves as a pattern. That is, where a normal rule would usually find one match and apply its actions on it and a forall-rule would usually find all matches and apply its actions to each of them, when called with `as-pattern`, a rule simply returns the sequence of its matches. With a normal rule, this sequence is a lazy sequence, i.e., the matches are not computed until they are consumed. With a forall-rule, the sequence is fully realized, i.e., all matches are already pre-calculated (possibly in parallel).

The second FunnyQT rule application modifier is `as-test`, and this is what is highly suitable for this transformation case. When a rule `r` is applied using `(as-test (r model))`, it behaves almost as without modifier but instead of applying the rule's actions immediately, it returns a closure of arity zero (a so-called *thunk*) which captures the rule's match and the rule's actions. Invoking the thunk causes the actions to be applied on the match. Thus, the caller of the rule gets the information if the rule was applicable at all, and if it was applicable, she can decide if she wants to apply it or not. And when she applies it, the pattern matching part is already finished and only the actions are applied on the pre-calculated match the thunk closes over.

In case of a forall-rule `r`, `(as-test (r model))` doesn't return a single thunk but a vector of thunks, one thunk per match of the rule's pattern. This is exactly what is needed for solving this transformation

case. Using this feature, a final function is defined that receives a rule `r` and a train model `g` and executes the rule as a test.

```
27 (defn call-rule-as-test [r g]
28   (as-test (r g)))
```

This function is then called with the transformation rules from the Java trainbenchmark framework. The given rule gets applied and returns a sequence of thunks which will apply the actions to the match they are wrapping. Thus, the only thing the framework has to do is to apply the thunks corresponding to the matches which are going to be repaired in the current repair phase.

These 28 lines of Clojure code form the complete functional part of the FunnyQT solution that solves all core and extension tasks. There is also a plain-Java glue project which implements the interfaces required by the benchmark framework and simply delegates to the Clojure/FunnyQT part of the solution. This glue project is briefly discussed in the following section.

2.2 Gluing the Solution with the Framework

Typically, open-source Clojure libraries and programs are distributed as JAR files that contain the source files rather than byte-compiled class files. This solution does the same, and that JAR is deployed to a local Maven repository from which the Maven build infrastructure of the benchmark framework can pick it up.

Then, in the FunnyQT glue project the rules and functions from above are referred to like shown in the next listing.

```
private final static String SOLUTION_NS = "ttc15-train-benchmark-funnyqt.core";
Clojure.var("clojure.core", "require").invoke(Clojure.read(SOLUTION_NS));
final static IFn POS_LENGTH = Clojure.var(SOLUTION_NS, "pos-length");
...
final static IFn CALL_RULE_AS_TEST = Clojure.var(SOLUTION_NS, "call-rule-as-test");
```

In line 2, the solution namespace `ttc15-train-benchmark-funnyqt.core` is required[7]. The `Clojure` class provides a minimal API for loading Clojure code from Java. When requiring a namespace as above, it will be parsed and compiled to JVM byte-code just in time[8].

Thereafter, the solution's in-place transformation rules and the `call-rule-as-test` function are referred to. `IFn` is a Clojure interface whose instances are Clojure functions that can be called using the `invoke()` method as can be seen in the definition of the glue project's `BenchmarkCase.check()` method shown below.

```
@Override
protected final Collection<Object> check() throws IOException {
    matches = (Collection<Object>) FunnyQTBenchmarkLogic.CALL_RULE_AS_TEST
                                    .invoke(rule, this.resource);
    // If the rule has no matches it returns nil/null but the framework
    // wants a Collection.
    if (matches == null) {
        matches = new LinkedList<Object>();
    }
    return matches;
}
```

In that code, `rule` is one of the rule `IFn`s `POS_LENGTH`, `SWITCH_SET`, et cetera, and they are called via the `call-rule-as-test` function to make them return one thunk per match instead of performing the rules' repair actions immediately.

The implementation of the `BenchmarkCase.modify()` method is even simpler.

[7] `require` is kind of Clojure's equivalent to Java's `import` statement.

[8] If the Clojure code was distributed in a pre-compiled form, the resulting classes would simply be loaded.

```
@Override
protected final void modify(Collection<Object> matches) {
    for (Object m : matches) {
        ((IFn) m).invoke();
    }
}
```

Since the rules are called as tests and thus return thunks that apply the rule's actions, those simply need to be invoked.

3 Evaluation & Conclusion

The FunnyQT solution implements all core and all extension tasks exactly as demanded by the case description, thus it is *complete*. When run in the benchmark framework, all assertion it checks are satisfied, thus the solution is also *correct*.

The FunnyQT solution consists of 28 NCLOC of FunnyQT/Clojure code for the five rules with their patterns and repair actions, and the function `call-rule-as-test`. Therefore, it is very *concise*.

Readability is a very subjective matter, and not everyone is fond of Lisp syntax. However, there are some strong points with respect to readability. (1) The queries (patterns) and repair actions are bundled in one in-place transformation rule each keeping the definition of cause and effect localized. (2) FunnyQT's pattern matching DSL used to specify the rules' patterns is both concise and readable. It should be easy to understand for graph transformation experts especially if they have used other textual graph transformation languages such as *GrGen.NET* before. It should also be easy to understand for any Clojure programmer because it strictly conforms to the style guidelines and best practices there.

FunnyQT implements pattern matching as a local search. Thus, each *recheck* phase take approximately as much time as the initial *check* phase. In contrast, with an incremental approach like *EMF-IncQuery*, the rechecking the patterns is not needed because all matches of all patterns are cached and updated when the model changes. This makes FunnyQT not especially suited for incremental model validation scenarios. However, given the fact that the evaluation of forall-patterns is automatically parallelized on multi-core machines, the *performance* is still reasonable. The benefit of FunnyQT's search-based approach is that it has far less memory requirements than an incremental approach. When comparing the performance with the EMF-IncQuery solution on an 8-core machine with 32 GB RAM, FunnyQT was only about 15% slower and could still transform models which already caused an `OutOfMemoryError` with EMF-IncQuery. But of course, when increasing the number of iterations, the performance benefit of incremental approaches will increase, too.

References

[1] Tassilo Horn (2013): *Model Querying with FunnyQT - (Extended Abstract)*. In Keith Duddy & Gerti Kappel, editors: *ICMT*, *Lecture Notes in Computer Science* 7909, Springer, pp. 56–57.

[2] Tassilo Horn (2015): *Graph Pattern Matching as an Embedded Clojure DSL*. In: *International Conference on Graph Transformation - 8th International Conference, ICGT 2015, L'Aquila, Italy, July 2015*.

[3] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation**. In: *Transformation Tool Contest 2015*.

The ATL/EMFTVM Solution to the Train Benchmark Case for TTC2015

Dennis Wagelaar
HealthConnect
Vilvoorde, Belgium
dennis.wagelaar@healthconnect.be

This paper describes the ATL/EMFTVM solution of the TTC 2015 Train Benchmark Case. A complete solution for all tasks is provided, three of which are discussed with regard to the three provided evaluation criteria: Correctness and Completeness of Model Queries and Transformations, Applicability for Model Validation, and Performance on Large Models.

1 Introduction

This paper describes a solution of the TTC 2015 Train Benchmark Case [4] made with ATL [2] and the EMF Transformation Virtual Machine (EMFTVM) runtime engine [5]. The Train Benchmark Case consists of several model validation and model repair tasks: three main tasks and two extension tasks. All of these tasks are run again increasing model sizes in order to measure the performance of each solution for the case. A complete solution for all tasks is provided, and is available as a GitHub fork of the original assignment[1]. Section 2 of this paper describes the ATL transformation tool and its features that are relevant to the case. Section 3 describes the solution to the case, and section 4 concludes this paper with an evaluation.

2 ATL/EMFTVM

ATL is a rule-based, hybrid model transformation language that allows declarative as well as imperative transformation styles. For this TTC solution, we use the new EMF Transformation Virtual Machine (EMFTVM). EMFTVM includes a number of language enhancements, as well as performance enhancements. For this TTC case, specific performance enhancements are relevant.

2.1 JIT compiler

EMFTVM includes a Just-In-Time (JIT) compiler that translates its bytecode to Java bytecode. EMFTVM bytecode instructions are organised in *code blocks*, which are executable lists of instructions. When a code block is executed more often than a predefined threshold, the JIT compiler triggers, and will generate a Java bytecode equivalent for the EMFTVM code block.

2.2 Lazy evaluation

EMFTVM includes an implementation of the OCL 2.2 standard library [3], and employs lazy evaluation for the collection operations (e.g. `select`, `collect`, `flatten`, `isEmpty`, etc.). That operations invoked

[1] `https://github.com/dwagelaar/trainbenchmark-ttc`

on collections are only (partially) executed when you evaluate the collection. For example, the `lazytest` query in Listing 1 invokes `collect` on a Sequence of all numbers from 0 to 100, which replaces each value in the Sequence by its squared value, but eventually only returns the last value of the Sequence. `collect` returns a lazy Sequence, which is just waiting to be evaluated. Only when `last` is invoked, the `square` operation is invoked on the last element of the input Sequence. As a result, `square` is only invoked once.

```
1 query lazytest = Sequence{0..100}->collect(x | x.square())->last();
2 helper context Integer def : square() : Integer =
3   (self * self).debug('square');
```

Listing 1: Lazy collections in ATL

2.3 Caching of model elements

Model transformations usually look up model elements by their type or meta-class. In the Eclipse Modeling Framework (EMF) [1], this means iterating over the entire model and filtering on element type. Often, an element look up by type is made repeatedly on the same model. In the case of this benchmark, the same query/transformation is run multiple times on the same model. For this reason, EMFTVM keeps a cache of model elements by type for each model. This cache is automatically kept up to date when adding/removing model elements through EMFTVM. The cache is built up lazily, which means that a full iteration over the model must have taken place before the cache is activated for that element type. This prevents a build up of caches that are never used.

3 Solution Description

The Train Benchmark Case involves first querying a model for constraint violations, and then repairing some of those constraint violations that are randomly selected by the benchmark framework. This means that the matching phase and the transformation phase, which are normally integrated in ATL, are now separated by the benchmark framework. The framework first launches the matching phase, and collects the found matches. After that, it randomly selects a number of matches, and feeds them into the transformation phase.

ATL provides a **query** construct that allows one to query the model using OCL and return the resulting values. The selected matches are fed back into the ATL VM through a helper attribute, specified in the framework repair transformation module shown in Listing 2. The benchmark framework copies the returned lazy collection into a regular `java.util.ArrayList`, which ensures that the performance measurements are valid.

The Repair transformation module contains a helper attribute `matches`, which is used to inject the matches selected by the benchmark framework. Furthermore, it contains a lazy rule `Repair`, which does nothing in this framework transformation. The `Repair` rule is invoked by every element in `matches` by the `Main` endpoint rule. The `Main` endpoint rule is automatically invoked. Normally, ATL transformations use matched rules that are automatically triggered for all matching elements in the input model(s). However, this benchmark requires the elements to transform to be set explicitly. Hence the need for this framework transformation module. All specific repair transformation modules are *superimposed* [6] onto the framework transformation module, and redefine the `Repair` rule. This means that for each task we only need to define an ATL query and a `Repair` rule. Because of space constraints, two out of five tasks will be discussed in this paper.

```
module Repair;
create OUT: RAILWAY refining IN: RAILWAY;
helper def : matches : Collection(OclAny) = Sequence{};
lazy rule Repair {
  from s: OclAny
}
endpoint rule Main() {
  do {
    for (s in thisModule.matches) {
      thisModule.Repair(s);
    }
  }
}
```

Listing 2: Framework repair transformation module in ATL

3.1 Task 1: PosLength

Listing 3 shows the ATL query for Poslength. It simply collects all Segment instances with a length of zero or smaller. Listing 4 shows the ATL repair transformation module for Poslength. It imports the framework Repair transformation module from Listing 2, and redefines the `Repair` rule. As no new elements need to be created, an imperative **do** block is used to make the required modification directly on the source element. The `<:=` assignment operator is used instead of the `<-` binding operator, such that the implicit source-to-target tracing is skipped.

```
query PosLength = RAILWAY!Segment.allInstances()->select(s | s.length <= 0);
```

Listing 3: PosLength query in ATL

```
module PosLengthRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s: RAILWAY!Segment
  do   { s.length <:= -s.length + 1; }
}
```

Listing 4: PosLength repair transformation module in ATL

3.2 Task 2: SwitchSensor

Listing 5 shows the ATL query for SwitchSensor. It collects all Switch instances for which the sensor is not set. Listing 6 shows the ATL repair transformation module for SwitchSensor. This time, the `Repair` rule also contains a **to** section that creates a new Sensor instance `se`. In the **do** section, this Sensor is assigned to the sensor reference of the input Switch element.

3.3 Extension Task 1: RouteSensor

Listing 7 shows the ATL query for RouteSensor. The query collects Tuples of each match, where a match is defined by Route `r`, SwitchPosition `p`, Switch `sw`, and Sensor `s`. A Tuple is created for each SwitchPosition connected to a Sensor that is not connected to the Route, for each Route that has Sensors connected to it. Listing 8 shows the ATL repair transformation module for RouteSensor. The `Repair` rule takes the Tuple match as input element, and adds the Sensor in the match to the Route's definedBy sensors.

```
query SwitchSensor = RAILWAY!Switch.allInstances()->select(s | s.sensor.oclIsUndefined());
```

Listing 5: SwitchSensor query in ATL

```
module SwitchSensorRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s: RAILWAY!Switch
  to   se: RAILWAY!Sensor
  do   { s.sensor <:= se; }
}
```

Listing 6: SwitchSensor repair transformation module in ATL

4 Evaluation and Conclusion

The solutions for the Train Benchmark Case are evaluated on three criteria: (1) *Correctness and Completeness of Model Queries and Transformations*, (2) *Applicability for Model Validation*, and (3) *Performance on Large Models*. We will now discuss how the ATL solution aims to meet these criteria.

4.1 Correctness and Completeness

The benchmark framework provides a set of expected query/transformation results, against which the output of the ATL solution can be compared. The `ATLTest` JUnit test case verifies that the output of the ATL solution matches the reference solution. The test results of each build are kept in the cloud-based Travis continuous integration platform[2]. This independent platform provides an objective proof that the ATL solution unit tests are passing.

4.2 Applicability

In order for a solution to be applicable for model validation, it must be concise and maintainable. Even though ATL is not primarily intended for interactive querying and transformation, it was easy to fit the ATL implementation into the benchmark framework. Simple queries are trivially expressed in OCL, using a functional programming style (PosLength, SwitchSensor). Complex queries that return tuples as matches (SwitchSet, RouteSensor, SemaphoreNeighbor) require a navigation strategy to be implemented. All repair phase transformations are all simple, single rule transformation modules that are *superimposed* onto a single framework Repair transformation module (see Listing 2). Query matches are

[2] `https://travis-ci.org/dwagelaar/trainbenchmark-ttc`

```
query RouteSensor = RAILWAY!Route.allInstances()
  ->select(r | r.definedBy->notEmpty())
  ->collect(r |
    r.follows->select(p |
      not p.switch.oclIsUndefined() and
      not p.switch.sensor.oclIsUndefined() and
      r.definedBy->excludes(p.switch.sensor)
    )->collect(p |
      Tuple{r = r, p = p, sw = p.switch, s = p.switch.sensor}
    )
  )->flatten();
```

Listing 7: RouteSensor query in ATL

```
module RouteSensorRepair;
create OUT: RAILWAY refining IN: RAILWAY;
uses Repair;
lazy rule Repair {
  from s : TupleType(r : RAILWAY!Route, p : RAILWAY!SwitchPosition, sw : RAILWAY!Switch,
                     s : RAILWAY!Sensor)
  do   { s.r.definedBy <:= s.r.definedBy->including(s.s); }
}
```

Listing 8: RouteSensor repair transformation module in ATL

provided via the rule **from** part, whereas the model element modification is done in a **do** block. Any new elements are specified in the **to** block.

4.3 Performance

In the ATL language, performance is achieved by using helper attributes instead of operations where possible, as helper attribute values are cached; accessing a helper attribute more than once on the same object will not trigger evaluation again, but just returns the cached value. EMFTVM also applies certain performance optimisations: complex code blocks are JIT-compiled to Java bytecode, which in turn may be JIT-compiled to native code by the JVM. Collections and boolean expressions are evaluated lazily, preventing unnecessary navigation. Finally, model elements are cached by their type, making repeated lookup of all instances of a certain metaclass more performant.

References

[1] Frank Budinsky, David Steinberg, Ed Merks, Ray Ellersick & Timothy J. Grose (2003): *Eclipse Modeling Framework*. The Eclipse Series, Addison Wesley Professional. Available at `http://safari.awprofessional.com/0131425420`.

[2] Frédéric Jouault, Freddy Allilaire, Jean Bézivin & Ivan Kurtev (2008): *ATL: A model transformation tool*. *Science of Computer Programming* 72(1-2), pp. 31–39, doi:10.1016/j.scico.2007.08.002.

[3] Object Management Group, Inc. (2010): *OCL 2.2 Specification*. Available at `http://www.omg.org/spec/OCL/2.2/PDF`. Version 2.2, formal/2010-02-01.

[4] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation*. In: *Proceedings of TTC 2015*. Available at `https://github.com/FTSRG/trainbenchmark-ttc/raw/master/paper/trainbenchmark-ttc.pdf`.

[5] Dennis Wagelaar, Massimo Tisi, Jordi Cabot & Frédéric Jouault (2011): *Towards a General Composition Semantics for Rule-Based Model Transformation*. In Jon Whittle, Tony Clark & Thomas Kühne, editors: *Proceedings of MoDELS 2011*, *Lecture Notes in Computer Science* 6981, Springer-Verlag, pp. 623–637, doi:10.1007/978-3-642-24485-8_46. Available at `ftp://progftp.vub.ac.be/tech_report/2011/vub-soft-tr-11-07.pdf`.

[6] Dennis Wagelaar, Ragnhild Van Der Straeten & Dirk Deridder (2009): *Module superimposition: a composition technique for rule-based model transformation languages*. *Software and Systems Modeling* 9(3), pp. 285–309, doi:10.1007/s10270-009-0134-3.

Train Benchmark Case: an EMF-INCQUERY Solution*

Gábor Szárnyas Márton Búr István Ráth
Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
szarnyas@mit.bme.hu, marton.bur@inf.mit.bme.hu, rath@mit.bme.hu

This paper presents a solution for the Train Benchmark Case of the 2015 Transformation Tool Contest, using EMF-INCQUERY.

1 Introduction

This paper describes a solution for the TTC 2015 Train Benchmark Case [6]. The source code of the solution is available as an open-source project.[1] There is also a SHARE image available.[2]

2 EMF-INCQUERY

Automated model transformations are frequently integrated with modeling environments, requiring both high performance and a concise programming interface to support software engineers. The objective of the EMF-INCQUERY [2] framework is to provide a declarative way to define queries over EMF models. EMF-INCQUERY extended the pattern language of VIATRA2 with new features (including transitive closure, role navigation, match count) and tailored it to EMF models [4]. EMF-INCQUERY is developed with a focus on *incremental query evaluation*, however, the most recent version is also capable of evaluating queries with a *local search-based* algorithm.

2.1 Incremental Pattern Matching

EMF-INCQUERY uses the Rete algorithm [1] to perform incremental pattern matching. The Rete algorithm uses tuples to represent the model objects, attributes, references and partial matches in the model. The algorithm defines an asynchronous network of communicating nodes. The network consists of three types of nodes. Input nodes are responsible for indexing the model by type, i.e. they store the appropriate tuples for the objects and references. They are also responsible for producing the update messages and propagating them to the worker nodes. Worker nodes perform a transformation on the output of their parent node(s) and propagate the results. Partial query results are represented in tuples and stored in the memory of the worker node, thus allowing for incremental query reevaluation. Production nodes are terminators that provide an interface for fetching the results of the query and the changes introduced by the latest transformation. Moreover, parallelization possibilities of the algorithm were already investigated in [3].

*This work was partially supported by the MONDO (EU ICT-611125) project and Red Hat Inc.

[1] https://github.com/FTSRG/trainbenchmark-ttc

[2] http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=ArchLinux64_TrainBenchmark-EIQ.vdi

The incremental pattern matcher provides quick reevaluation for complex queries. However, it does so at the expense of high memory consumption as the partial results are stored in the Rete network.

2.2 Local Search-Based Pattern Matching

Local search-based pattern matching (LS) is commonly used in graph transformation tools. Along with the incremental query engine, EMF-INCQUERY also provides a local search-based pattern matcher.

The matching process consists of four steps. (1) At first, in a preprocessing step the patterns are *normalized*: the constraint set is minimized, variables that are always equal are unified and positive pattern calls are flattened. These normalized patterns are evaluated by (2) the *query planner*, using a specified cost estimation function to provide search plans: totally ordered lists of search operations used to ensure that the constraints from the pattern definition hold. From a single pattern specification multiple search plans can be derived, thus pattern matching includes (3) *plan selection* based on the input parameter binding and model-specific metrics. Finally, (4) *the search plan is executed* by a plan interpreter evaluating the different operations of the plans. If an operation fails, the interpreter backtracks; if all operations are executed successfully, a match is found.

Compared to the incremental query engine, the search-based algorithm requires less memory [7] and is therefore capable of performing queries on larger models if there is not enough memory available for the incremental engine.

2.3 Defining the Pattern Matching Strategy

Currently, the pattern matching strategy has to be determined by the developer by specifying the query backend of EMF-INCQUERY. Developing a hybrid pattern matching engine is subject to future work. This will allow the user to use annotations to define the evaluation strategy for each pattern. There are also plans to develop an adaptive query engine (Section 5).

2.4 Pattern Match Representation

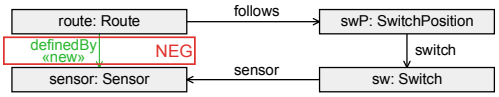
For each query, EMF-INCQUERY generates a set of utility classes. These classes store the model objects in the match and provide a convenient interface for reading and transforming the matches. These classes are used for implementing the transformation operations (Section A.1).

3 Solution

The case defines a well-formedness validation scenario set in the domain of railway systems [6]. The case provides a synthetic instance model generator which is capable of generating models of various sizes. For the solution, we used the metamodel defined in the case description without any modifications or extensions.

The solution was developed in the Eclipse IDE. For setting up the development environment, please refer to the readme file. The projects are not tied to the Eclipse environment and can be compiled with the Apache Maven build automation tool. This offers a number of benefits, including portability and the possibility of continuous integration. The solution is written in Java 7. The patterns are defined in INCQUERY Pattern Language (IQPL) [4].

3.1 Example Query: RouteSensor



We describe the implementation of the RouteSensor query in detail. The other queries and transformations are implemented in a similar manner. The implemented application uses the Java classes generated by EMF-INCQUERY and the hand-coded transformation logic introduced below. First it finds the matches of the queries, then the corresponding transformation step is applied for each match. The code of patterns and the transformation definitions are listed in Section A.1.

```
1  pattern routeSensor(route, sensor, switchPosition, sw)
2  {
3    Route.follows(route, switchPosition);
4    SwitchPosition.^switch(switchPosition, sw);
5    TrackElement.sensor(sw, sensor);
6    neg find definedBy(route, sensor);
7  }
8
9  pattern definedBy(route, sensor)
10 {
11   Route.definedBy(route, sensor);
12 }
```

Listing 1: Pattern of the RouteSensor query.

The RouteSensor query looks for sensors that are connected to a switch, but the sensor and the switch are not connected to the same route. The query in IQPL is listed in Listing 1. The positive conditions are defined by using the appropriate classes and references, while the negative application condition (NAC) is defined as a negative find operation (`neg find`) for a separate query.

During the repair operation, for the selected matches, the missing definedBy edge is inserted by connecting the route to the sensor. The Java transformation code implementing the transformation is listed in Listing 2. The transformation uses the match object returned by EMF-INCQUERY.

```
1 public void transform(final Collection<Object> matches) {
2   for (final Object match : matches) {
3     final RouteSensorMatch rsm = (RouteSensorMatch) match;
4     rsm.getRoute().getDefinedBy().add(rsm.getSensor());
5   }
6 }
```

Listing 2: Transformation of the RouteSensor query.

3.2 Query Evaluation Strategies for the RouteSensor Pattern

We use the RouteSensor query to provide an overview of the various query evaluation strategies used in EMF-INCQUERY.

3.2.1 Incremental Evaluation

The Rete network derived from the RouteSensor query is shown in Figure 7. For the sake of clarity, we simplified the Rete network by removing some implementation-specific details. The evaluation in the Rete network starts with the *input nodes* (switch, follows, sensor, definedBy), which are indexing the model by collecting the appropriate tuples. The *worker nodes* are responsible for performing the relational operations, *join* and *antijoin* in this case. The join nodes have a pair of tuple masks (e.g. $\langle 2,3 \rangle$ and $\langle 0,1 \rangle$) to determine the attributes used in the join operation. The match set of the pattern is stored in the *production node*.

3.2.2 Local Search-Based Evaluation

The search plan generated for evaluating the RouteSensor query is shown in Figure 8. This screenshot is taken from the *Local Search Debugger* view of EMF-INCQUERY. The search plan is presented in the upper-left part, while the found matches with the variable substitutions are shown below the search plan in a table viewer. The Zest-based graph viewer in the right visualises a match based on the selection of the table viewer.

4 Evaluation

In this section, we present the benchmark environment and evaluate the results.

4.1 Benchmark Environment

The benchmarks were performed on a 64-bit Ubuntu Server 14.04 virtual machine deployed on a private cloud. The machine used a quad-core 2.50 GHz Xeon L5420 processor and 16 GB of memory. We used Oracle JDK 8 and set the available heap memory to 15 GB.

4.2 Benchmark Results

To present the results, we use the reporting framework of the Train Benchmark. The framework generates plots to visualise the execution time of the phases defined in the benchmark. The plots showing each query are included in Section A.2. On each plot, the x axis shows the problem size, i.e. the size of the instance model, while the y axis shows the aggregated execution time of a certain phases, measured in milliseconds. Both axes use logarithmic scale.

4.2.1 Benchmark Results for the RouteSensor Query

For the sake of conciseness, we only discuss the results for the RouteSensor query in detail.

The results for the *batch validation* are shown in Figure 1. The results suggest that—given enough memory—both the incremental and the local search-based (LS) strategies are able to run the query and the transformation for the largest model. The *first validation* takes consistently longer for the incremental strategy as for the LS strategy. This is caused by the fact that the incremental strategy builds the Rete algorithm during the read phase. However, the difference is small as the *first validation* time largely consists of deserializing the EMF model.

The execution times of the *revalidation* are shown in Figure 2. The execution time of the incremental strategy linearly correlates with the size of the change set. This implies that for a *fixed* change set, the incremental strategy is able to perform the transformation in constant time, while execution time for the LS strategy correlates with the model size. For the *proportional* change set, the revalidation time is a low-degree polynomial of the model size for both strategies, however, it is an order of magnitude faster for the incremental strategy than for the LS.

4.3 Comparison of the Query Evaluation Strategies

Section A.2 shows the detailed results for all queries and both evaluation strategies. In the *first validation* (Figure 3 and Figure 5), the evaluation strategies show similar performance characteristics as both have

to compute the complete result set of the query. The execution times for both strategies show that the most complex query is SemaphoreNeighbor, while the simplest one is SwitchSensor.

As expected, the execution times of the *revalidation* are different for the two strategies. Figure 4 shows that for the incremental strategy the execution time correlates with the size of the match set (instead of the model size). This can be observed when comparing the execution times for the *fixed* and the *proportional* change sets. Figure 6 shows that the execution time for the LS strategy is determined by the model size and is not affected by the size of the change set.

These results imply that the optimal evaluation strategy depends on the specific workload profile. If there is enough memory available and the transformations operate on a small amount of model elements, it is recommended to use the incremental strategy. If the transformations often change a large proportion of the model elements, the LS strategy is recommended.

5 Summary and Future Work

The paper presented a solution for the Train Benchmark case of the 2015 Transformation Tool Contest.

There is ongoing work to develop a hybrid query engine [5] for EMF-INCQUERY. This will allow the user to use annotations on the patterns for specifying the desired query evaluation strategy. There are also plans to develop an adaptive query engine which will use query optimisation heuristics to determine the appropriate strategy based on the query, the model and the available resources.

Acknowledgements. The authors would like to thank Zoltán Ujhelyi for providing valuable insights into EMF-INCQUERY.

References

[1] Gábor Bergmann (2013): *Incremental Model Queries in Model-Driven Design*. Ph.D. dissertation, Budapest University of Technology and Economics, Budapest. Available at `http://home.mit.bme.hu/~bergmann/download/phd-thesis-bergmann.pdf`.

[2] Gábor Bergmann, Ákos Horváth, István Ráth, Dániel Varró, András Balogh, Zoltán Balogh & András Ökrös (2010): *Incremental Evaluation of Model Queries over EMF Models*. In: *MODELS*, Springer, Springer, doi:http://dx.doi.org/10.1007/978-3-642-16145-2_6.

[3] Gábor Bergmann, István Ráth & Dániel Varró (2009): *Parallelization of Graph Transformation Based on Incremental Pattern Matching*. *Electronic Communications of the EASST, Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques* 18. Available at `http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/265/249`.

[4] Gábor Bergmann, Zoltán Ujhelyi, István Ráth & Dániel Varró (2011): *A Graph Query Language for EMF Models*. In: *Theory and Practice of Model Transformations, Fourth Intl. Conf., LNCS* 6707, Springer.

[5] Ákos Horváth, Gábor Bergmann, István Ráth & Dániel Varró (2010): *Experimental Assessment of Combining Pattern Matching Strategies with VIATRA2*. *International Journal on Software Tools for Technology Transfer* 12(3-4), pp. 211–230, doi:10.1007/s10009-010-0149-7. Available at `http://dx.doi.org/10.1007/s10009-010-0149-7`.

[6] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation*. In: *8th Transformation Tool Contest (TTC 2015)*.

[7] Zoltán Ujhelyi, Gábor Szőke, Ákos Horváth, Norbert István Csiszár, László Vidács, Dániel Varró & Rudolf Ferenc (2015): *Performance Comparison of Query-based Techniques for Anti-Pattern Detection*. *Information and Software Technology*, doi:10.1016/j.infsof.2015.01.003. In press.

A Appendix

A.1 Patterns and Transformations

A.1.1 PosLength

```
pattern posLength(segment)
{
  Segment.length(segment, length);
  check(length <= 0);
}
```

Listing 3: Pattern of the PosLength query.

```
public void transform(final Collection<Object> matches) {
  for (final Object match : matches) {
    final RouteSensorMatch rsm = (RouteSensorMatch) match;
    rsm.getRoute().getDefinedBy().add(rsm.getSensor());
  }
}
```

Listing 4: Transformation of the PosLength query.

A.1.2 SwitchSensor

```
pattern switchSensor(sw)
{
  Switch(sw);
  neg find hasSensor(sw);
}

pattern hasSensor(sw)
{
  TrackElement.sensor(sw, _);
}
```

Listing 5: Pattern of the SwitchSensor query.

```
public void transform(final Collection<Object> matches) {
  for (final Object match : matches) {
    final SwitchSensorMatch ssm = (SwitchSensorMatch) match;
    final Sensor sensor = RailwayFactory.eINSTANCE.createSensor();
    ssm.getSw().setSensor(sensor);
  }
}
```

Listing 6: Transformation of the SwitchSensor query.

A.1.3 SwitchSet

```
pattern switchSet(semaphore, route, switchPosition, sw)
{
  Route.entry(route, semaphore);
  Route.follows(route, switchPosition);
  SwitchPosition.^switch(switchPosition, sw);

  Semaphore.signal(semaphore, ::GO);
  SwitchPosition.position(switchPosition, swPP);
  Switch.currentPosition(sw, swCP);

```

```
  swPP != swCP;
}
```

Listing 7: Pattern of the SwitchSet query.

```
public void transform(final Collection<Object> matches) {
  for (final Object match : matches) {
    final SwitchSetMatch ssm = (SwitchSetMatch) match;
    ssm.getSw().setCurrentPosition(ssm.getSwitchPosition().getPosition());
  }
}
```

Listing 8: Transformation of the SwitchSet query.

A.1.4 RouteSensor

The RouteSensor query is discussed in detail in Section 3.1.

A.1.5 SemaphoreNeighbor

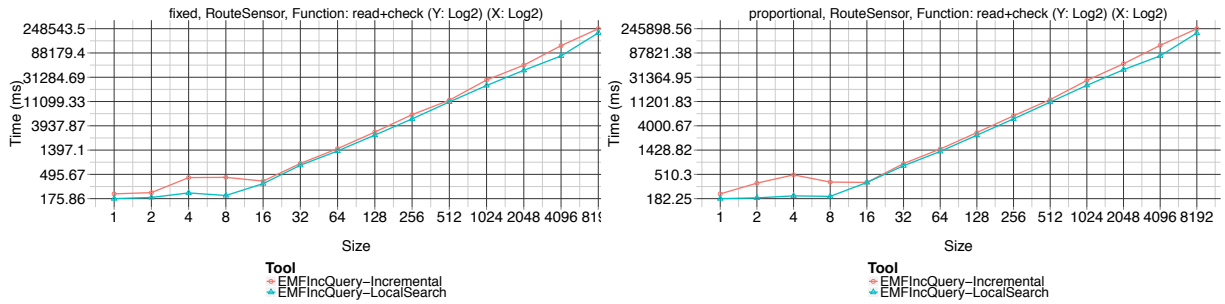
```
pattern semaphoreNeighbor(semaphore, route1, route2, sensor1, sensor2, te1, te2)
{
  Route.exit(route1, semaphore);
  Route.definedBy(route1, sensor1);
  TrackElement.sensor(te1, sensor1);
  TrackElement.connectsTo(te1, te2);
  TrackElement.sensor(te2, sensor2);
  Route.definedBy(route2, sensor2);
  neg find entrySemaphore(route2, semaphore);

  route1 != route2;
}

pattern entrySemaphore(route, semaphore)
{
  Route.entry(route, semaphore);
}
```

Listing 9: Pattern of the SemaphoreNeighbor query.

```
public void transform(final Collection<Object> matches) {
  for (final Object match : matches) {
    final SemaphoreNeighborMatch snm = (SemaphoreNeighborMatch) match;
    snm.getRoute2().setEntry(snm.getSemaphore());
  }
}
```

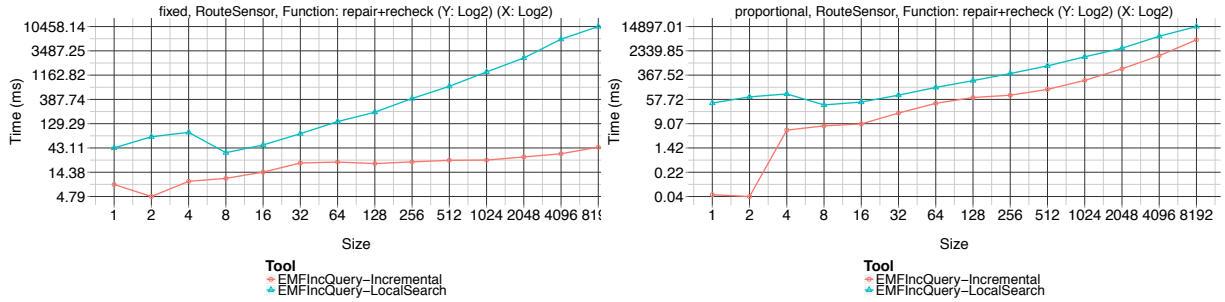
Listing 10: Transformation of the SemaphoreNeighbor query.

A.2 Detailed Benchmark Results



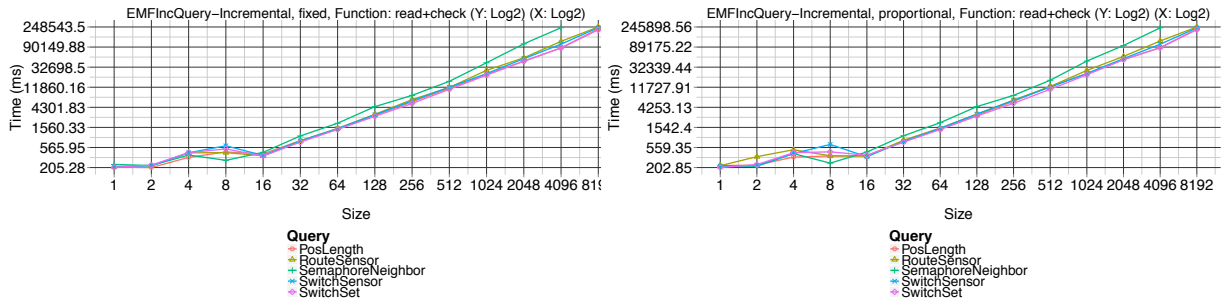
(a) Fixed change set (b) Proportional change set

Figure 1: First validation times for the RouteSensor query.



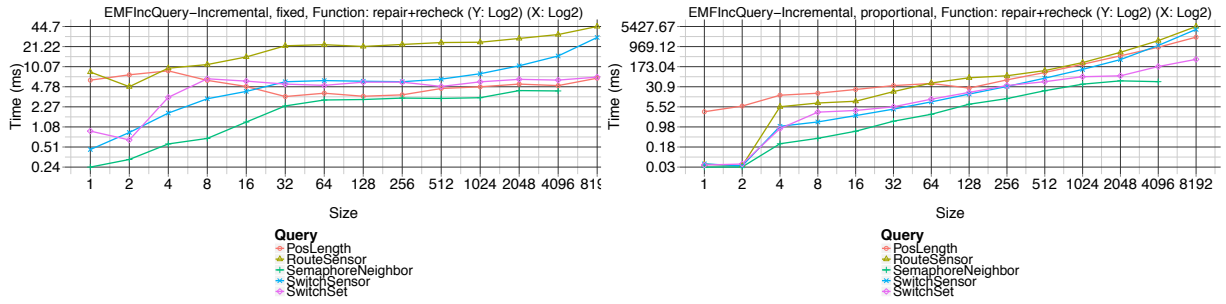
(a) Fixed change set (b) Proportional change set

Figure 2: Revalidation times for the RouteSensor query.



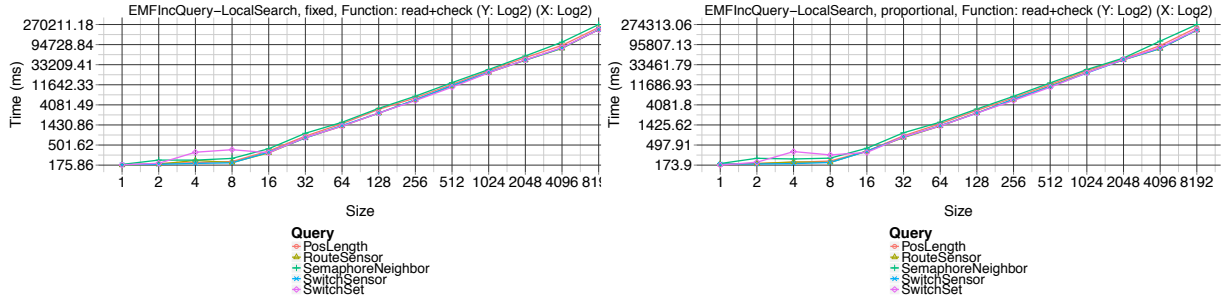
(a) Fixed change set (b) Proportional change set

Figure 3: First validation times for the *incremental* query evaluation strategy.



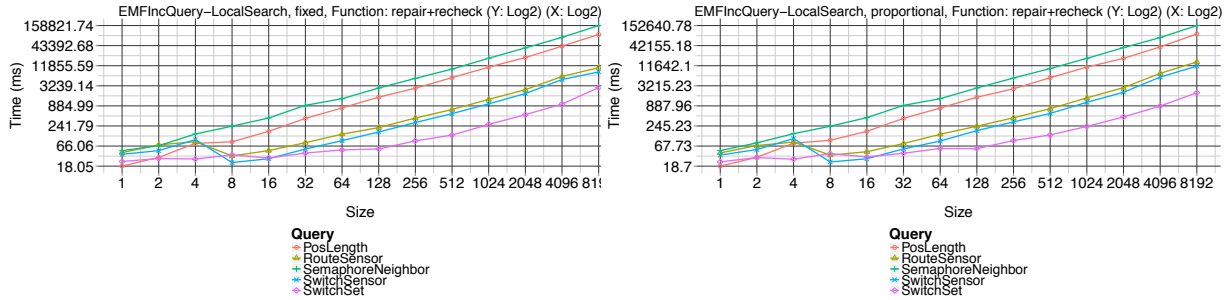
(a) Fixed change set (b) Proportional change set

Figure 4: Revalidation times for the *incremental* query evaluation strategy.



(a) Fixed change set (b) Proportional change set

Figure 5: First validation times for the *local search-based* query evaluation strategy.



(a) Fixed change set (b) Proportional change set

Figure 6: Revalidation times for the *local search-based* query evaluation strategy.

A.3 Rete Network

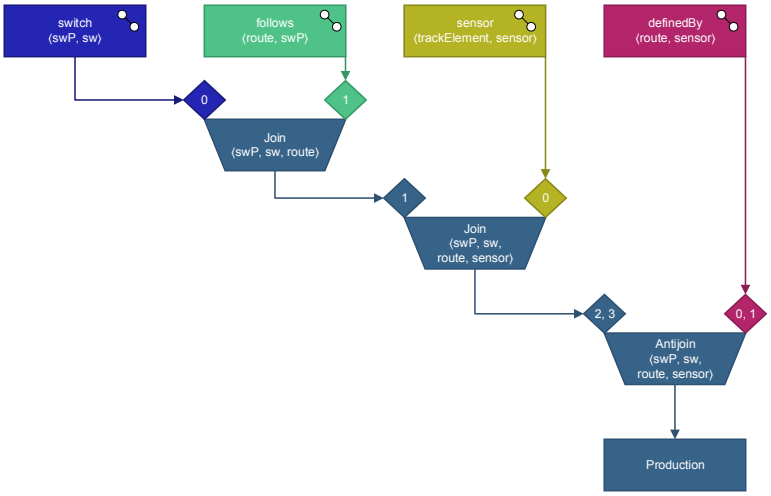


Figure 7: The Rete network for the RouteSensor query.

A.4 Local Search Plan

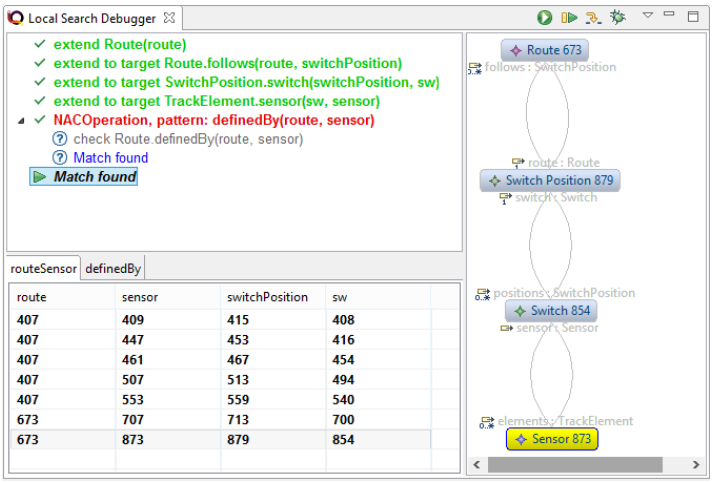


Figure 8: The search plan and the matches for the RouteSensor query.

Solving the TTC'15 Train Benchmark Case Study with SIGMA

Filip Křikava
Faculty of Information Technology
Czech Technical University, Czech Republic
filip.krikava@fit.cvut.cz

This paper describes a solution for the *Transformation Tool Contest 2015* (TTC'15) Train Benchmark case study using SIGMA, a family of Scala internal *Domain-Specific Languages* (DSLs) that provides an expressive and efficient API for model consistency checking and model transformations.

1 Introduction

The purpose of the TTC'15 Train Benchmark case study [3] is to systematically assess the scalability of consistency checking and repair of large scale models. It presents a scenario from the railway domain for which the solution requires to implement 5 constraints and repair operations of increasing complexity. An associated framework is then used to evaluate the correctness and performance of the solutions over large model instances.

In this paper we present our solution using SIGMA [1], a family of Scala[1] internal DSLs for model manipulation tasks such as model validation, model to model (M2M), and model to text (M2T) transformations. Scala is a statically typed production-ready *General-Purpose Language* (GPL) that supports both object-oriented and functional styles of programming. It uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages. Furthermore, it is supported by the major integrated development environments bringing EMF modeling to other IDEs than traditionally Eclipse (*e.g.* IntelliJ IDEA was used for this solution).

SIGMA DSLs are embedded in Scala as a library allowing one to manipulate models using high-level constructs similar to the ones found in the external model manipulation DSLs. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, with a reduced learning overhead as well as improved usability and performance.

The solution is based on the *Eclipse Modeling Framework* (EMF) [2], which is a popular metamodeling framework widely used in both academia and industry, and which is directly supported by SIGMA. The complete source code is available on Github[2] in the fork of the original case study repository.

[1] http://scala-lang.org

[2] https://github.com/fikovnik/trainbenchmark-ttc in the `hu.bme.mit.trainbenchmark.ttc.benchmark.sigma` module

2 Solution Description

The solution for this transformation case study consist of a set of queries that check for violations of a number of model constrains and repair transformations that in turn fixes them. SIGMA provides a dedicated model consistency checking DSL with the ability to provide quick fixes repairing invariant validations. However, given the benchmark framework used in the case study, we decided to provide a more dedicated support for the given query/repair tasks in a form of an internal DSL. The reason is that (1) it allows for an easy comparison between the reference implementations in Java and EMF-IncQuery and (2) it shows the expressiveness of the language allowing one in few lines of code to bridge the gap between the problem-level abstractions (query, repair transformation) and the implementation-level concepts (*e.g.*, classes, higher-order functions). We therefore only rely on the SIGMA operations for model navigation (*i.e.* projecting information from models) and modification (*i.e.* changing model properties or elements). Essentially, these operations bridge the model classes (Ecore classes in this case) to be compatible with Scala allowing for example one to use the powerful Scala collection library.

On the top of SIGMA, we have created a small internal DSL that allows us to solve the given benchmark cases in an expressive and compact way. Following the case study description, the top-level domain concept is a *constraint*. A constraint is composed of a model *query* that finds all model instances violating a certain model restriction and a *repair* transformation correcting the failed instances. Concretely, a query is a function that given a model element—*i.e.* a context of the constraint in the classical model consistency checking—returns a set of *matches*. A match can either be a single instance or a tuple of instances of model elements that are related to the violations.

The following description of the solution is split in two parts: (1) the core part that describes the queries and repair transformations, (2) the integration part gives an overview how it has been integrated in the case study source code.

2.1 Queries and Repair Transformations DSL

A typical way of creating an internal DSL in Scala is by designing a library that allows one to write fragments of code with domain-specific syntax. These fragments are woven within Scala's own syntax so that it appears different.

One way to represent the above concepts is using a Scala case class:

```
1 case class Constraint[A <: EObject, B <: AnyRef](
2   query: (A) => Iterable[B],
3   repair: (B) => Unit
4 )
```

This defines a case class with a field for both query and repair. A case class in Scala is like a regular class with some additional properties out which, in our case, the important one is that it can be instantiated without the `new` keyword and thus limiting the language noise. The two type parameters `A, B` specify the model context for the query and the types of matches the query produces. The input type is further constrained to be a subtype of an `EObject`. The query and repair are defined as functions `A` → `Iterable[B]` and `B` → `Unit` where `Unit` is like `void` in Java.

In some cases the match returned by the query is of the same type as the query context. The query can be therefore simplified to a boolean expression selecting instances on which it evaluates to true. For these types of queries we provide a dedicated construct called `BooleanConstraint`:

```
case class BooleanConstraint[A <: EObject : ClassTag](
  query: (A) => Boolean,
  repair: (A) => Unit
)
```

For example, the first query, *PosLength*, which finds all the segments with negative length can be written as:

```
BooleanConstraint[Segment](
  query = segment => segment.length < 0,
  repair = segment => segment.length += -segment.length + 1
)
```

We do not have to specify the types of the parameter nor the result as they will be inferred by the Scala compiler.

Another example using more complex expression is the *SwitchSet* constraint:

```
Constraint[SwitchPosition, (Semaphore, Route, SwitchPosition, Switch)](
  query = swP => {
    for {
      semaphore <- Option(swP.route.entry) if semaphore.signal == Signal.GO
      sw = swP.switch if sw.currentPosition != swP.position
    } yield (semaphore, swP.route, swP, sw)
  },

  repair = {
    case (_, _, swP, sw) => sw.currentPosition = swP.position
  }
)
```

This is a more complex constraint that matches a tuple of model elements. It is using a *for comprehension*, a lightweight notation for expressing sequence comprehensions[3]. Scala for comprehensions have the form `for (enumerators) yield e`, where enumerators refers to a list of enumerators. An *enumerator* is either a generator which introduces new variables, or it is a filter. A comprehension evaluates the body *e* for each binding generated by the enumerators and returns a sequence of these values.

In this concrete example, the generator is the optional value of the `Route.entry` reference. It either generates a single value in the case the actual instance contains one or it does not produce anything. There is a small inconsistency in the model, the `Route.entry` should have the cardinality set to `0..1` instead of `1`, and that is why we need to explicitly convert the reference to an `Option`.

Finally, the repair function is defined using a pattern matching construct allowing us to concisely assign variables from the matching tuple.

[3] http://docs.scala-lang.org/tutorials/tour/sequence-comprehensions.html

2.2 Operationalization and Integration

The integration consists in making our solution work within the provided benchmark framework. First, next to constraint syntax, we also need to define its semantics. For that we define a validator which operationalizes the DSL executing the checks and consequent repairs of the incorrect model instances. It is defined as an abstract class with two methods that correspond to the two operations:

```
abstract class Validator[A <: EObject, B <: AnyRef] {
  def check(container: EObject): Iterator[B]
  def repair(matches: Iterator[B]): Unit
}
```

The implementation is straight forward. For all elements contained in a `container`, we first collect all instances of the required context type and then query them using the query function provided by the given constraint. The repair simply executes the constraint repair function on the matching element.

```
case class ConstraintValidator[A <: EObject, B <: AnyRef](constraint: Constraint[A, B])
  extends Validator[A, B] {

  override def check(container: EObject) =
    container.eAllContents collect { case x: A => x } flatMap constraint.query

  override def repair(matches: Iterator[B]) =
    matches foreach constraint.repair
}
```

Finally, we instantiate all the constraints, plugs them into the validator and connects the result to the provided framework. The integration schema is shown in Appendix B. We also create a `SigmaBenchmarkComparator` that is used to compare the matches as required by the case study. It is a general comparator that either compares single instances (results from boolean constraints violations) or tuples (regular constraints violations).

3 Evaluation

In this section we provide an evaluation of our solution following the categories specified in the case study.

Correctness and Completeness of Model Queries and Transformations. We developed a solution for all of the tasks required by the case study and the solution passes the provided tests.

Conciseness. The solution itself consists of 52 lines of Scala code the internal DSL developed for this case study. The DSL itself has been implemented using 20 lines of Scala code using SIGMA. The integration part consists of three files with the total of 65 lines. All measures are source lines only excluding comments and new lines. Given these measures, we believe that the code is rather concise.

Readability. Next to being concise, the solution is also quite expressive. This means that the given problem (queries and repair transformations) naturally maps into the implementation. The higher-level abstraction provided by both SIGMA and the internal DSLs helps to facilitate it making

a significant improvement over the Java reference implementation. The code is also type-safe as Scala is statically typed language. A notable consequence is that it is very easy to use the DSL with an IDE like Eclipse or IntelliJ that provides a robust code completing functionalities, outline views and other features increasing one's productivity.

In summary, while readability is a subjective matter and largely depends on the background and experience of users, we believe that SIGMA scores well. Thanks to the syntax of Scala which is close to one of Java/C++ and hence shall be familiar to many developers. The expressiveness of the first-order logic collection operation should be familiar to anyone knowing OCL or any other function language.

Performance on Large Models. The tests have been performed on an 2.3 GHz Intel Core i7 machine with 16 GB of RAM being dedicated to the JVM process. We ran our solution together with the reference implementation in Java. We used the model instances from size 1 to 8192 and set 8GB memory to be dedicated to the JVM. The performance is similar to the Java reference implementation which has been expected due to the fact that Scala compiles directly to Java bytecode and we use the same underlying libraries for accessing EMF models. This shows that we can leverage from concise and expressive queries without sacrificing performance. The overhead of using SIGMA is mostly on the compile time where the implicit conversions are inlined by the Scala compiler.

It is important to note that we do not developed any extra functionality for these benchmarks—*i.e.* no caching or incremental validations. On the other hand, functional approach we have selected makes it perfect for further parallelization.

4 Conclusion

This paper presents a solution for the Train Benchmark case study of the 2015 Transformation Tool Contest. It demonstrates some of the features of the SIGMA internal DSLs for model manipulation as well as the extensibility, expressiveness and scalability of the Scala host language. The solution is realized as a tiny internal DSL in Scala that mixes in SIGMA common infrastructure for EMF model querying and manipulation. There is a significant improvement in the readability and conciseness of the solution, yet the performance is similar to the reference Java version.

Acknowledgments. This work is partially supported by the Datalyse project[4].

References

[1] Filip Krikava, Philippe Collet & Robert France (2014): *SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations*. In: *Proceedings of 17th International Conference on Model-Driven Engineering Languages and Systems*, MODELS, Valencia.

[2] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional.

[3] Gábor Szárnyas, Oszkár Semeráth, István Ráth & Dániel Varró (2015): *The TTC 2015 Train Benchmark Case for Incremental Model Validation**. In: *Transformation Tool Contest 2015*.

[4] http://www.datalyse.fr

A Constraints

In the following we describe the individual constraints that were part of the case study (case study tasks) except the *PosLength* and *SwitchSet* which have already been shown above (*cf.* Section 2).

— *SwitchSensor*

```
BooleanConstraint[Switch](
  query = switch => switch.sensor.isEmpty,
  repair = switch => switch.sensor = Sensor()
)
```

The `isEmpty` is a method that is defined on an `Option` type (coming from the standard Scala library) representing a type which may or may not have a value. Since in the model, the *sensor* reference of the `Switch` class is defined as optional (with cardinality `0..1`), in SIGMA we represent the reference using the `Option` class. Not only makes this the cardinality expressed in the type definition, but it also prevents some of the `NullPointerExceptions` caused by traversing unset references. Technically, this is realized by implicit conversions (*cf.* Krikava *et al.* [1]).

— *RouteSensor*

```
Constraint[Route, (Route, Sensor, SwitchPosition, Switch)](
  query = route => {
    for {
      swP <- route.follows
      sw = swP.switch
      sensor <- sw.sensor if !(route.definedBy contains sensor)
    } yield (route, sensor, swP, sw)
  },

  repair = {
    case (route, sensor, _, _) => route.definedBy += sensor
  }
)
```

The implementation is similar to the the previous case. It is also based on a for comprehension and closely follows the description of the query.

— *SemaphoreNeighbor*

```
Constraint[Route, (Semaphore, Route, Route, Sensor, Sensor, TrackElement, TrackElement)](
  query = route1 => {
    for {
      sensor1 <- route1.definedBy if route1.exit != null
      te1 <- sensor1.elements
      te2 <- te1.connectsTo
      sensor2 <- te2.sensor
      route2 <- sensor2.sContainer[Route] if route1 != route2
      semaphore = route1.exit if semaphore != route2.entry
    } yield (semaphore, route1, route2, sensor1, sensor2, te1, te2)
  },

  repair = {
    case (semaphore, _, route2, _, _, _, _) => route2.entry = semaphore
  }
)
```

Again based on the for comprehension. Additionally, we provide a shortcut using the `route1.exit != null` so immediately skip the route instances that do not have an exit semaphore set.

B Integration with the Train Benchmark Framework

Figure 1 shows the various layers of integration of the solution into the train benchmark framework.

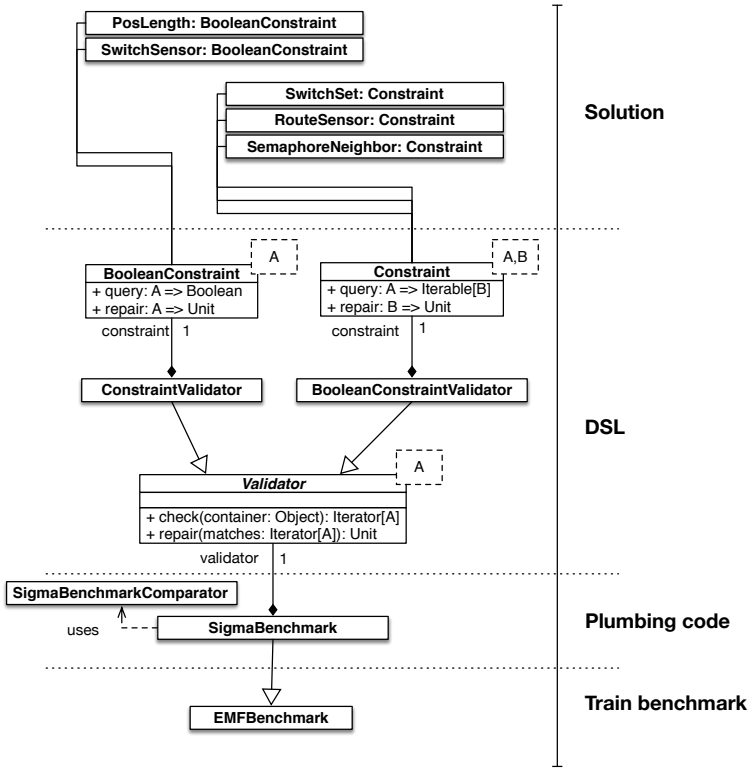


Figure 1: Integration schema

C Performance Comparison

The performance comparison charts (*cf.* Figures 2, 3, 4 and 5) have been generated by the case study benchmark. They present a performance comparison between SIGMA and the reference implementation in Java on the model instances from size 1 to 8192 using 8GB memory dedicated to the JVM. The corresponding results are shown in the figures 2 and 3. We compare them to the Java solution which is shown in the figures 4 and 5.

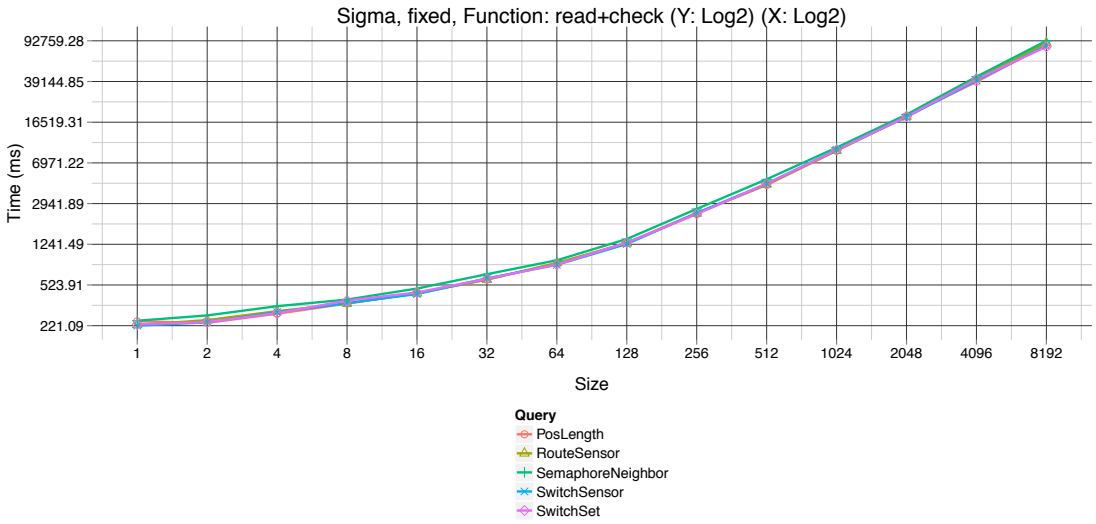


Figure 2: SIGMA fixed validation batch

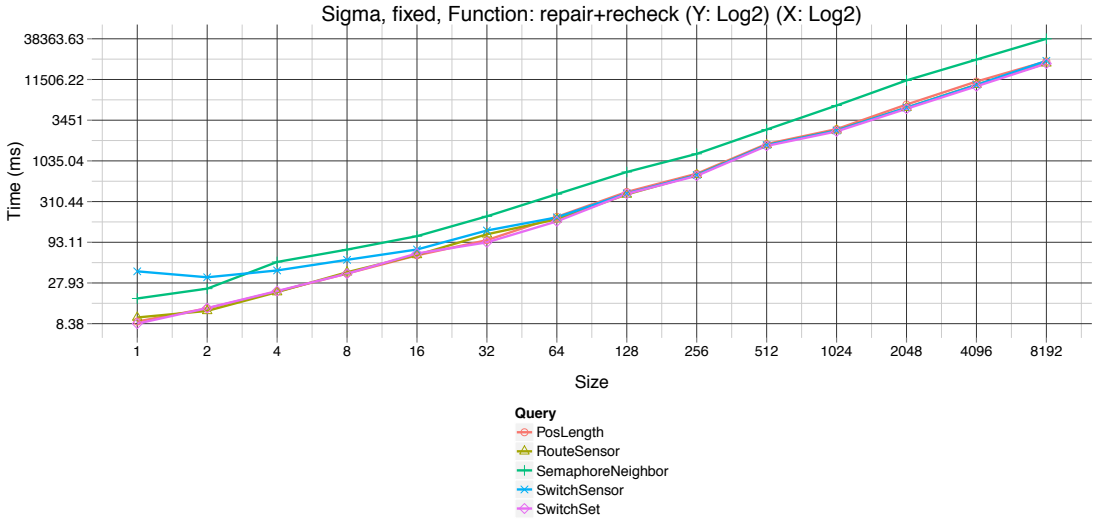


Figure 3: SIGMA fixed revalidation

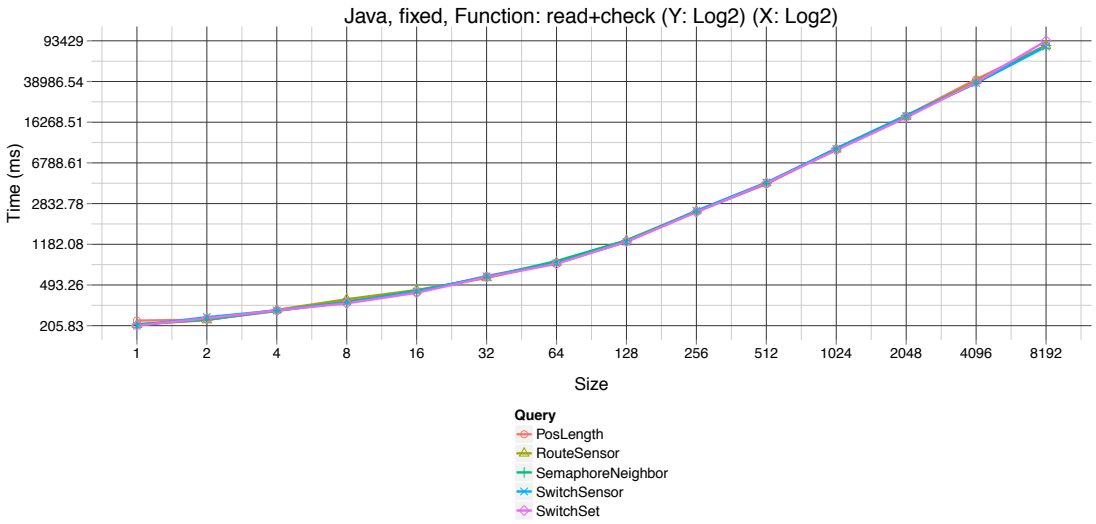


Figure 4: Java fixed validation batch

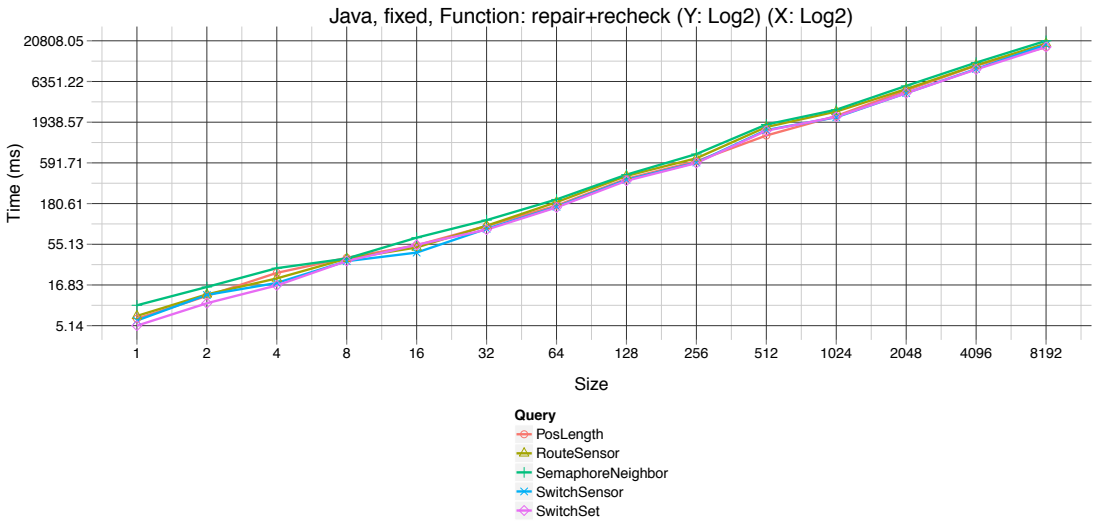


Figure 5: Java fixed revalidation