

Transforming Process Models: executable rewrite rules versus a formalized Java program

Pieter Van Gorp, Rik Eshuis

Eindhoven University of Technology
School of Industrial Engineering

Abstract. In the business process management community, transformations for process models are usually programmed using imperative languages (such as *Java*). The underlying mapping rules tend to be documented using informal visual rules whereas they tend to be formalized using mathematical set constructs. In the Graph and Model Transformation communities, special purpose languages and tools (such as *GrGen*¹) are being developed to support the direct execution of such mapping rules. As part of our ongoing effort to bridge these two communities, we have implemented a transformation from petri-nets to statecharts (*PN2SC*) using both approaches. By relying on technical comparison criteria and by making the solutions available for online replay, we illustrate that rule-based approaches require less specification effort due to their more declarative specification style and automatic performance optimizations. From a tool perspective, *GrGen* has better visualization and debugging support whereas *Java* tools support evolution better.

1 Introduction

This paper contributes to the emerging field of transformation engineering. We define transformation engineering as the discipline of (i) *decomposing* complex transformation problems into manageable parts, (ii) making reproducible decisions when *designing* a transformation solution, (iii) *benchmarking* and *selecting* transformation *languages* and *tools*, and (iv) *verifying* transformation *results*. Although transformations are already developed for decades in various communities (such as the compiler community, the program transformation community and the business process management (BPM) community), it is relatively new to study the strengths and weaknesses from transformation approaches across community boundaries.

In this paper, we illustrate how a transformation program from the BPM domain can be systematically compared with a behaviorally equivalent solution based on graph rewriting techniques. This provides novel insights in the strengths and weaknesses of the two approaches. The transformation program is written in Java whereas the graph rewriting solution is based on GrGen but the results can easily be extended for other solutions based on similar platforms.

¹ We use the short *GrGen* name to denote *GrGen.NET 2.6* [6].

As a conceptual framework, we rely on the “*taxonomy of model transformation*” [12,17]. In general, taxonomies are used for a wide variety of purposes [7]. This specific taxonomy has for example been used in education as well as in research (see [19] for examples).

We improve the structure of the taxonomy by classifying transformation solutions on four taxonomy levels (the *conceptual*, the *design*, the *language*, and the *tool* level) and put quantitative results in a proper qualitative perspective. The organization in four levels is often inspiring, as one can for example observe that a particular limitation of a Java based transformation program is not a consequence of using that language but merely the consequence of a design decision. This paper does not present the complete taxonomy. [17] provides a wider and deeper coverage. Moreover, [19] provides more details on the case study.

In the next section, we present related work from the field of transformation engineering. Section 3 introduces the case study that we solved using Java as well as using GrGen. Section 4 describes the solutions, Section 5 elaborately evaluates them and Section 6 presents our conclusions. All results from this paper can be reproduced in an online virtual machine [18].

2 Related Work

This paper contributes to the aforementioned field of transformation engineering. This field is emerging from a series of satellite events from graph and model transformation conferences. The “*graph transformation tools contest*” in 2007 used a transformation from *UML* activity diagrams to *CSP* as a case study to compare 11 graph transformation based approaches [21]. The 2008 edition of that contest did not involve a case study related to BPM. The 2009 edition of that contest used a transformation from *BPMN* to the *BPEL* as a case study to compare 10 transformation approaches [4]. Our participation in these events has lead to refinements of the taxonomy that supports in this paper.

This two transformation approaches that are evaluated in this paper have been selected with care: for the 2007 edition of the contest, there was no Java based solution available for comparison. This was a severe drawback, since most contributions to BPM conferences are based on Java based implementations. Using *GrGen* to represent the wide range of graph transformation tools makes sense, since the *GrGen* platform is recognized as state-of-the-art in that area [20].

This paper clearly separates the underlying mapping rules from design, language and tool related features. We have explicitly defined a set of “*core*” mapping rules for which we compare the two implementations in detail. In contrast, solutions to the *BPMN2BPEL* case from the 2008 contest implement different sets of mapping rules, which makes it unfair to compare the solutions from a non-functional point of view. Focusing on non-functional properties is important since in transformation engineering all platforms under study tend to be equally expressive. Therefore, any set of conceptual mapping rules can eventually be implemented using any of the evaluated platforms, which means that their fundamental differences relate to non-functional properties only.

van Amstel et al. have proposed a set of metrics to quantify the quality of model transformation programs [1]. In summary, that work should be extended with metric implementations for more transformation languages. This would enable the *quantification* of quality attributes such as size and complexity. For this paper, we only quantify transformation performance and size. As described in Section 5.5, we currently quantify transformation size using the very basic Lines Of Code (LOC) metric but we are collaborating with van Amstel on the development of more advanced measurement instruments.

3 Case Study: Translating Petri-nets to Statecharts

This section introduces the PN2SC case study that we have solved using Java and GrGen. We assume that the reader is at least aware of petri-nets [13] and statecharts [10] (the language of the input and output models of our case study). Others are invited to consider [19] first. We do not assume in-depth knowledge of advanced petri-net analysis methods or of high level petri-nets. Similarly, we only assume basic familiarity with statecharts. Having two executable *Petri-Net to Statechart* (PN2SC) implementations is *not* considered a BPM contribution in itself, and we also refer the reader to [5] for (i) a proof on the completeness of the reduction rules for a particular subclass of petri-nets and (ii) a discussion of the practical consequences thereof. Notice again though that our classification method can be applied to other mapping problems, languages and tools.

3.1 Example Input and Output Model

Fig. 1 is based on the running example from the original paper on the PN2SC mapping [5]. Black dots visualize *tokens*. These tokens represent data. Circles visualize *places*. Places can hold tokens. A particular distribution of tokens across a petri-net (often called a *marking*) represents a particular process *state*. Black bars visualize *transitions*. A transition represents a process *activity*. When such activities occur, the net moves to a new state, that is: tokens are redistributed across the net. More specifically, a transition t that *fires* moves tokens from the places which have an incoming arc to t (the *input places*) to places which have an outgoing arc from t (the *output places*).

A transition can only fire if all of its input places hold a token. In such a case, the transition is said to be enabled. [5] restricts the class of input petri-nets for *PN2SC* to those which never reach a state with more than one token in a place.

A key feature of the translation PN2SC is that it maps a petri-net to a statechart in a structure-preserving way, such that the statechart syntax resembles the petri-net syntax. The translation is behavior-preserving according to the petri-net *token game* (i.e., standard) semantics. It can be used as a foundation for implementing translations from event-driven petri-nets to event-driven statecharts. The translation enables the exchange of models between tools. For example, BPM designers can use petri-net tools to design and analyze business processes and use statechart-based tools to generate code from their design.

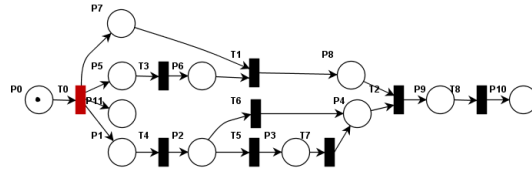


Fig. 1. Example petri-net model that can be used as input for PN2SC.

Fig. 2 ((a) and (b)) represents the running example in *statechart* syntax. More precisely, this diagram represents the above process model in the “state machine diagram” syntax from the Eclipse *MDT-UML2Tools* plugin. In the context of the *PN2SC* mapping, the relevant characteristic of the output formalism is that it has the notion of “*concurrent regions*” within “*compound states*”. In statechart terminology, states that hold concurrent regions are called “*AND nodes*” whereas the individual regions are called “*OR nodes*”. An *OR* node can in turn hold compound concurrent states (*AND* nodes) as well as “*basic nodes*”.

The black dot that is connected to *P0* represents an “*initial state*”. The top-most *AND* node (labeled *\$15832433*) holds only one *OR* node, which contains the one initial state. Thus, the process clearly has a single entry point. Fig. 2 ((a) and (b)) also shows two black dots surrounded by a circle. These dots represent “*final states*”. The transitions from our input petri-net are mapped to *hyper-edges*. Hyper-edges with exactly one input and output state are visualized as an atomic arc. Hyper-edges representing fork and join behavior are displayed as black bars (similar to transitions in petri-nets).

Notice that these hyper-edges can either be shown explicitly (as in Fig. 2 (a)) or transformed into additional start and final states within the concurrent *OR* regions (as in Fig. 2 (b)). States in which decisions are made or where conditional branched paths merge again are represented as rhombi (see the elements after state *P1* and state *P3*).

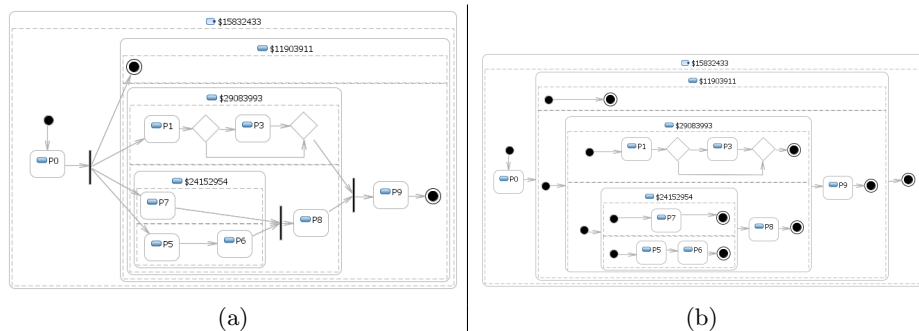


Fig. 2. Statechart representations of the running example.

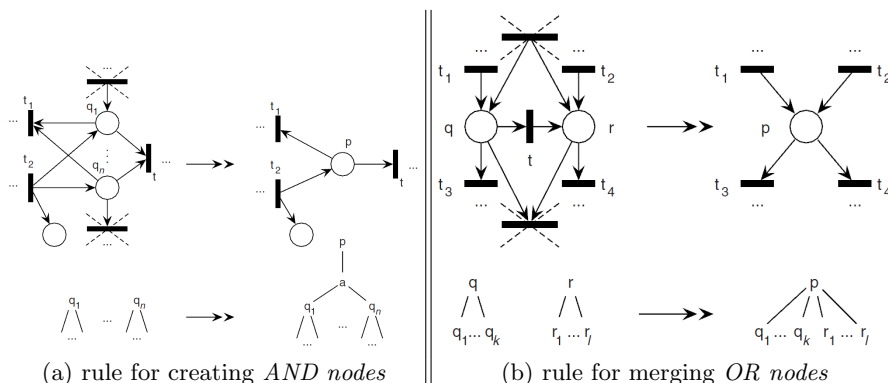


Fig. 3. Visual documentation for mapping rules.

3.2 Mapping Rules

This paper defines the “*core*” of the PN2SC mapping as the set of mapping rules that make that hierarchy explicit. Section 5.1 clearly separates that task from other subproblems (such as mapping to UML style.)

Fig. 3 is extracted from [5]. Remark that Fig. 3 is intended to *document* the mapping rules. [5] also presents a mathematical formalization of the rule preconditions as well as a pseudo code description of the rule side-effects and the rule scheduling algorithm. Other applications of this specification style can be found for example in [13] (to document petri-net reduction rules) and [3] (to document a mapping from BPMN to petri-nets).

The observation that several transformations in the BPM domain are documented in this manner is an important one, since (i) it indicates that the mathematical formalizations are not considered adequate documentation, and (ii) it provides a basis for judging the understandability of the implementations that are compared in this paper.

The upper part of the rules, as shown on Fig. 3, visualizes how elements from the input net should incrementally be folded into a single place. Since each rule application reduces the amount of possible rule matches, the rule set is known to be terminating. The bottom part of the rules, as shown on Fig. 3, visualizes how the hierarchical structure is introduced in the output model.

The rule shown on Fig. 3 (a) expresses that all statechart elements corresponding to an input place of an *AND join* transition should be *grouped*. The lower part shows how a new *AND* parent node is introduced in the statechart domain. The upper part shows restrictions on the applicability of the rule: notice how dashed crosses visualize what kind of transitions should not be present when applying this rule. When applying the rule, arcs between q_i and t_j are redirected to a newly created element p . This paper does not aim to make the full meaning of the rule clear; instead, it intends to give an idea of the nature of the rules (and the way in which they are documented) and refers to [5] for more details. Notice that the complete mapping also contains a rule for *AND splits*

(parallel *fork* patterns [15]). That rule resembles the one shown on Fig. 3 (b) but has the arcs connected to t in the reverse direction.

The rule shown on shown on Fig. 3 (b) shows that the algorithm does not accept arbitrary sequential structures: again, the dashed cross syntax is used to depict patterns that prevent the rule from matching. In summary, it seems desirable that these rules are implemented on a platform that supports the specification of *positive* as well as *negative* patterns.

4 Description of the Solutions

This section describes the two solutions that implement the rules from Section 3.2. The solutions are classified and compared in the next section. Fig. 4 shows a fragment from the Java solution while Fig. 5 shows the corresponding fragment from the GrGen solution.

To clarify the rationale behind the Java solution, we first explain a previous version thereof. In the original Java version, each of the two reduction rules ((a) and (b) from Fig. 3) was implemented by a separate method. The reduction procedure was started by invoking the method for rule b . In the method for rule b , transitions were processed one by one. For each transition it was checked whether the precondition for rule b was satisfied and if so, rule b was applied, and the method was called again recursively. If rule b was not applicable to any transition, then the method for rule a was invoked. In this method, each transition was processed to check whether the precondition for rule a was satisfied. If so, rule a was applied and the method for rule b was invoked again. If rule a was not applicable to any transition, the reduction procedure failed.

Clearly, in this initial design, the set of transitions was traversed many times and sometimes unnecessary. This observation led to the current version of the Java solution, which uses a search algorithm (see line 7 to 13 and lines 43 to 57) to select a transition to which the rules can be applied in order, that is, first rule a (see lines 14 to 39) is applied to the preset and postset, and next rule b . If one of the rules fails for the selected transition, the complete reduction procedure fails. In contrast, if in the original Java version all reduction rules failed on a certain part of the net, another part of the net was searched to which the reduction rules could be applied. The search algorithm has been formalized in pseudo code [5] too.

The Java solution represents a large class of other Java based transformation approaches in that input elements are traversed explicitly: a while loop (cfr., lines 2 to 41) iteratively checks whether there are still transitions (from the petri-net) that need to be transformed. A perhaps unconventional feature of the Java solution is that it heavily relies on vector indexing (cfr., lines 5, 8 and 26.) Since the formal description of the mapping does not rely on indices [5], one can conclude that the Java solution contains *redundant* technical details. A Java solution based on collection iterators would not have that disadvantage.

Another property of the Java solution is that it does not leverage Java classes to check the type-safety of the generated output elements. More specifically, the

```

1 public String reduce(){
2   while (trs.size()>0){
3     // find lower bound transition t
4     int i=1;
5     Transition t=(Transition)trs.get(0);
6     // t is lower bound
7     while (i<trs.size()){
8       Transition t2=(Transition)trs.get(i);
9       if (check(t2,t)){
10        t=t2;
11      }
12      i++;
13    }
14    Vector sources=t.getSources();
15    if (sources.size()>1){
16      if (checkTransitions(sources)){
17        Vector toreplace=new Vector(sources);
18        String tId = t.getId();
19        State newState = new State(tId);
20        newState.addChildren(toreplace);
21        newState.setAnd();
22        State newStParent= new State("xx_o");
23        newStParent.addChild(newState);
24        newStParent.setOr();
25        for (i=0;i<trs.size();i++){
26          Transition tx=(Transition)trs.get(i);
27          Vector sourcesx=tx.getSources();
28          if (contain(sourcesx,toreplace)){
29            tx.clearSources(toreplace,newStParent);
30          }
31          Vector targetsx=tx.getTargets();
32          if (contain(targetsx,toreplace)){
33            tx.clearTargets(toreplace,newStParent);
34          }
35        }
36        states.add(newStParent);
37        states.removeAll(toreplace);
38      }
39    }
40    ... // code for other rules
41  }
42 }
43 public boolean check(Transition t1, Transition t2){
44   Vector sources1=t1.getSources();
45   Vector targets2=t2.getTargets();
46   if (targets2.containsAll(sources1)
47       && sources1.size()<targets2.size()){
48     return true;
49   }
50   Vector sources2=t2.getSources();
51   Vector targets1=t1.getTargets();
52   if (sources2.containsAll(targets1)
53       && sources2.size()>targets1.size()){
54     return true;
55   }
56   return false;
57 }

```

Fig. 4. Java code for handling AND joins (cfr., Fig. 3a and 5).

setAnd and *setOr* calls (cfr., lines 21 and 24) are used for dynamic changing the metaclass of the transformed elements.

As a final observation, notice that the Java solution hard-codes one particular sequential order of execution for the mapping rules (cfr., line 40, which obviously represents several more lines of code.) This over-specification may seem harmless from a behavioral point of view but (i) new programmers may get the *wrong* impression that the rules are *required* to execute in that particular order and (ii) it makes rule oriented optimization (cfr., [11]) virtually impossible. Remark that this fundamental problem can only be overcome by embedding in Java a String-based graph transformation language interpreter. See section 5.3 for other language-specific characteristics and section 5.2 for characteristics that only relate to design choices of the transformation writer.

The GrGen “code” fragment shown on Fig. 5 may come as a surprise to some readers, since it is rather uncommon to program graph transformation systems using *textual* syntax. On the other hand, the fragment contains applications of language constructs that one may know from graph transformation systems based on visual transformation languages: for example, the “*negative*” construct supports the specification of negative application conditions [9]. In a nutshell, the construct provides formal support for the realization of the dotted crosses shown on Fig. 3.

The most commonly known characteristic of graph transformation rules is however that they consist of two dominant parts (a *left-hand* and a *right-hand* side.) The left-hand side of a rule describes the pattern that needs to be looked up in the input model (the host graph), the left-hand side describes the pattern that should be realized when *applying* the rule. For example, in the *trans_AND-join* rule, the left-hand side (between lines 2 and 4 of Fig. 5) specifies a pattern

```

1 rule trans_AND_join {
2   -:pre-> t:Transition <:-pre-; // at least two
3   negative { :RoguePrePlace(t); }
4   negative { :RoguePostPlace(t); }
5   modify {
6     p:Place -:pre-> t; // fold
7     // update tree
8     :HSCandState <:-:HSCcontains- :HSCorState
9     <:-:PN2HSC- p;
10    eval { p.name= "ANDJOIN_"+t.name; }
11    exec([handlePrePlace_AND_join(t,p)]);
12  }
13 }
14 rule handlePrePlace_AND_join(t:Transition,p:Place) {
15   q_i:Place -:pre-> t; // each place in preset
16   q_i -:PN2HSC-> or:HSCorState; // take OR
17   p -:PN2HSC-> :HSCorState -:HSCcontains->
18     parentAND:HSCandState;
19   modify {
20     or <:-:HSCcontains- parentAND; // move OR node
21     exec([move_incoming_arcs(q_i,p)]
22         | [move_outgoing_arcs(q_i,p)]
23         | [cleanupP(q_i)]); // INPUT DESTRUCTIVE
24   }
25 }
26
27 rule move_outgoing_arcs(srcPl:Place,trgPl:Place) {
28   otherTrans:Transition <-e1:pre- srcPl;
29   alternative {
30     // ALTERNATIVE PATTERN 1
31     NotYetPresent {
32       negative {
33         otherTrans <:-:pre- trgPl;
34       }
35       modify {
36         otherTrans <:-:pre- trgPl;
37       }
38     }
39     // ALTERNATIVE PATTERN 2
40     AlreadyPresent {
41       otherTrans <:-:pre- trgPl;
42       modify {
43         // do not link otherTrans to trgPl again!
44       }
45     }
46   }
47   modify {
48     delete(e1);
49   }
50 }

```

Fig. 5. GrGen Rule and Subrule for handling AND joins (cfr. Fig. 3a and 4).

consisting of a transition that has at least two incoming arcs. The right-hand side (between lines 6 and 11) specifies among other things that a new *Place* element should be inserted before that transition. That element is represented by variable p , exactly as in Fig. 3 (a). Also notice that elements that have no name in the informal specification (i.e., in the rules shown on Fig. 3) can be left anonymous in the formal GrGen specification too (e.g., the two anonymous edge variables of type pre on line 2.)

Unlike the imperative Java solution discussed above, GrGen programs require no explicit element traversal, nor element indices. Another characteristic of the GrGen solution is that it heavily relies on *subpatterns* (e.g., “*:RoguePrePlace(t)*” on line 3). Such subpatterns enable reuse and encapsulation. The example fragment also shows how the right-hand side of the *trans_AND_join* triggers another rule (the *handlePrePlace_AND_join* rule) using the *exec* statement (e.g., lines 11 and 21). Within *exec* statements, subrules can be orchestrated using various control flow constructs (for example “ $[]$ ” denotes the *concurrent* execution of a rule on *all* possible matches in the host graph).

Initially, the GrGen solution was implemented based on the Java solution, as described in [5], so using the search algorithm. It turned out that it was difficult to specify the search criterion in GrGen and the runtime performance was poor. We therefore decided to also implement a GrGen solution without the search algorithm. This new GrGen solution, which is presented in this paper, resembles the initial Java version, but with the distinction that in GrGen the ordering of the rules does not need to be explicitly specified by the user: the GrGen engine determines the ordering (scheduling) of the rules automatically. The new GrGen solution turns out to be much more efficient than the initial GrGen solution and even outperforms the optimized Java solution. In retrospect, we conclude that the development of model transformations using imperative programming languages like Java may bias researchers towards optimizations

that are specific to imperative programming and it may blur the distinction between conceptual mapping rules and technical details that are specific to an implementation technology.

5 Evaluation of the Solutions

This section classifies the solutions according to the *conceptual*, *design*, *language* and *tool* level and then presents quantitative results.

5.1 Conceptual Classification

Before comparing the two solutions further from a *design*, *language* or *tool* perspective, we need to ensure that the solution fragments under consideration are responsible for exactly the same *conceptual* tasks. Since we want to focus our comparison on the realization of the core mapping rules that were presented in Subsection 3.2, we first need to decompose the large PN2SC translation problem into more manageable parts. We also need to check the correctness of both solutions.

Decomposition of the PN2SC Case Study Different subproblems can be classified by transformation type (*translation*, *reiteration*, or *rephrasing*), input/output type (*text-to-model*, *model-to-model*, or *model-to-text*) and abstraction effect (*horizontal* or *vertical*) [19].

The core transformation is of type *translation*, since the input modeling language (petri-nets) is clearly different from the output modeling language (statecharts). We also agree to treat the core as a *model-to-model* transformation, which implies for example that input parsing and output serialization is considered a separate text-to-model and model-to-text transformation steps. We also agree that adapting the input and output elements to the proposed meta-model is not the responsibility of the core solution. Similarly, the core does not handle adaptation to UML syntax, removal of hyper-edges (i.e., the step between Fig. 2 (a) and (b)), or adaptation to standards such as XMI.

In [19], we also characterize these subproblems, since that helps defining their scope without becoming overly technical. For the sake of this paper, the primary purpose of the decomposition into subproblems is however that it enables the comparison of integrated transformation solutions from an end-user perspective. More specifically, in Section 5.1, we can now outline the completeness of the integrated transformation chains in which the two core solutions are embedded.

Completeness and Correctness of the two Solutions Table 1 shows that besides the core problem, both solutions tackle a series of other challenges. The table contains a cross in a cell if the solution from that row implements the feature from that column (see [19] for details). Several features support the exchange of models for benchmarking purposes: the read/write PNML and read/write GRPN features have enabled us to test both solutions on (conceptually)

Table 1. Completeness of the solutions from a User perspective.

	core	r. PNML	w. PNML	PN MM'	r. GRPN	w. GRPN	w. GRSC	rm. HE	to UML style	w. XMI
Java	x	x	x			x	x			
GrGen	x		x	x	x	x		x	x	x

the same set of input models. The GrGen solution is unique in that it implements several extensions to the basic mapping rules from [5]: it implements the “remove hyper-edges” feature (as discussed in the context of Fig. 2 (b)) as well as the features related to the UML.

The correctness of both solutions was verified by means of input/output testing. For this purpose, we have composed a test-suite consisting of real-life process models (among others: models from the SAP reference model [16]), manually fabricated process models (to test specific patterns) as well as automatically fabricated process models. The latter type of test models were used to evaluate the performance and scalability of the solutions. This type of models was generated based on models from a petri-net benchmark by Bergmann et al. [2,19].

5.2 Design Specific Classification

Classifying the design of a transformation is a key ingredient for making fair comparisons between transformation solutions: it would for example be unfair to generalize for example the quantitative results from Section 5.5 towards conclusions about *any other* Java and GrGen transformation. In this section, we use standard transformation jargon to classify those design choices that could have been made otherwise in Java and GrGen.

The Java solution has been designed as an endogenous, in-place model transformation: firstly, the metamodel for input and output models is realized as a data structure that has no clear separation between petri-net and statechart related properties. Secondly, input model elements are destroyed during transformation execution, since they become an integral part of the output model. In contrast, the GrGen solution has been designed as an exogenous, out-place model transformation: firstly, input and output models have two distinct metamodels, and secondly output elements are populated by copied values from the input elements. During transformation execution, traceability links are maintained between input elements and the corresponding output elements.

For the sake of comparability and simplicity, the GrGen solution destroys the input petri-net during transformation execution too: more specifically, the folding operations that are visualized on Fig. 3 are realized by removing input elements. We mention this simplification, since it may be undesirable when the transformation would be deployed in an integrated modeling environment, where input and output models are supposed to co-evolve.

Both the Java and the GrGen solution automatically transform *all* elements from the input model. Again it is well possible to change both solutions into a variant that only transforms a controlled subset of the input model. However,

taking the GrGen fragment from Fig. 5 as an example, this would require the *trans_AND_join* rule to have a parameter of type *Transition*. This parameter would need to be instantiated explicitly by another rule. Similar extensions would need to be made to the Java solution. These changes would have little impact on simple size metrics such as *LOC* but may significantly impact metrics that were proposed for transformation complexity (such as *val-in/out* and *fan-in/out* [1].) In general, different design choices will have an impact on the usability of the solution fragments under consideration. Before making quantitative comparisons concerning size and complexity, one needs to take these differences into account. Moreover, these choices are *design* (not *language* or *tool*) specific.

The Java and GrGen *designs* are similar in terms of other taxonomy elements [19]: both solutions are unidirectional, they do not support change propagation and they do not support tolerating inconsistencies (e.g., indicating that some non-safe petri-net elements should be ignored.) The main threat for performing a further evaluation of both approaches is that the Java design is endogenous/in-place whereas the GrGen solution is exogenous/out-place: an exogenous solution in Java would for example require more metamodel definition effort. We have not created an endogenous/in-place variant of the Java solution since that would only strengthen the results of our quantitative analysis (cfr., Section 5.5), rather than raise new insights.

5.3 Language Specific Classification

This section discusses those solution properties that are a direct consequence of the underlying transformation (programming) languages. This paper focuses on those properties that convey interesting differences between the two solutions. In those cases that both Java and GrGen suffer from a limitation that has already been addressed by other transformation languages, the reader is referred to the related literature.

The primary strength of the GrGen language over Java relates to the declarative constructs for pattern matching and rule scheduling. As an example of the pattern matching part, consider line 2 from Fig. 5, which very concisely specifies that this rule for *and joins* applies only to those transitions that have at least two incoming *pre* arcs. As an example of the GrGen rule scheduling operators, consider the following fragment, which models explicitly that the *trans_place* and *trans_transition* rules need to be executed before the other rewrite rules, whereas the order in which the transformation system iterates over the *transform_singleton_pre_post*, *trans_AND_split* and *trans_AND_join* rules is left implicit (by using the “\$” operator). Since Java only offers the sequential “;” operator, it is less declarative from the rule scheduling perspective:

```
[trans_place]&&[trans_transition]&&[trans_post]&&[trans_pre]&&[trans_token]
| (transform_singleton_pre_post $|| trans_AND_split $|| trans_AND_join)*
```

From the perspective of *genericity* and *modularization* however, Java is a more expressive language than GrGen. GrGen for example has no support for

rule *specialization* (a form of genericity) and also lacks mechanisms to hide or import rules using module constructs. In [19], we refer to *QVT*, *Story Diagrams*, *Triple Graph Grammars* (TGGs) as examples of rule based languages that have already overcome these limitations.

These languages also have more declarative support for *traceability*, *incremental updates* and *bidirectionality* than both GrGen and Java. Notice that neither the GrGen nor the Java solution from this paper implements these features. Although one can explicitly implement these features using any general purpose language, any TGG or QVT/Relations solution would inherently support them.

A cognitive analysis of the solutions based on Green and Petre [8] has primarily learned us that GrGen specifications are closer to the problem domain: all variables shown on Fig. 5 map directly to variables from the informal rule shown on Fig. 3. In contrast, there are various helper variables (e.g., *i* and *tId*) needed in the corresponding Java fragment shown on Fig. 4.

The cognitive analysis also enables us to name a problem that has caused some errors during the solution development: the GrGen language is inconsistent about its default matching semantics. As concrete evidence from consequences for the PN2SC case, [19] shows a GrGen implementation fragment that is surprisingly more complex than its mathematical counter-part.

Moreover, the GrGen textual syntax enforces some *pre-mature commitment* (as defined by [8]) with regards to the order of statement specification: even though the aforementioned “\$” operator provides semantic support for conveying that rules are allowed to be executed in any order, a textual syntax forces one to trigger rules in a particular (and artificial) syntactic order. Finally, the highly declarative and textual operators for rule scheduling may require *hard mental operations* (as defined by [8]) for developers with a history in using more imperative languages.

Except for the inconsistent default matching semantics, Java exposes the same cognitive problems as GrGen. Both languages also lack support for the “*secondary notation*” [8], while visual languages typically enable one to use layout and color to convey additional semantical clues. Finally, remark that more Java programmers are available on the job market but many may be interested in learning language such as GrGen as more experimental results become available.

5.4 Tool Specific Classification

In this section, we only discuss those tool features that we have found most useful during the development of the PN2SC solutions.

Firstly, we rate the debugger usability of Eclipse lower than that of the GrGen suite, for two reasons: (i) the Java debugger from Eclipse has no rule-oriented user interface so developers should know which particular methods or parts thereof happen to implement a transformation rule and (ii) the Eclipse debugger shows irrelevant technical variables on the same level as pattern variables.

Secondly, Eclipse does not provide a generic visualizer for the transformed models. The Java solution does include integration code for the *dot* visualiza-

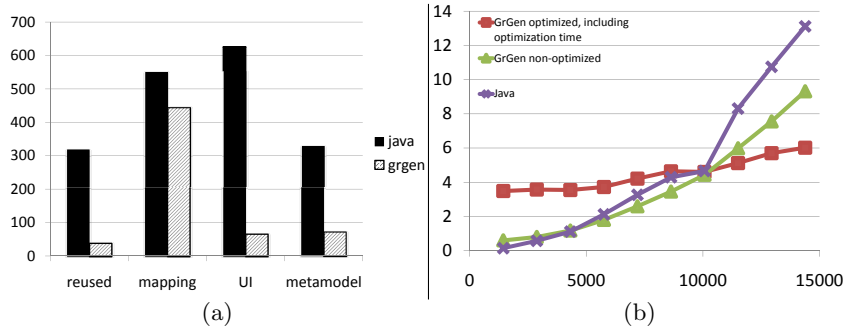


Fig. 6. Size (LOC) and performance (execution time seconds) of the solutions.

tion framework but since Eclipse is unaware thereof there is for example no integration with the Eclipse debugger. Moreover, the *dot* framework exposes performance problems that do not occur when using the GrGen suite.

From an interoperability perspective, Eclipse provides some support for standards such as XMI or MOF. Since the Java solution pre-dates the related Eclipse plugins, the core of the Java solution to PN2SC has been written without taking that into account. At the time of writing, only the GrGen solution provides XMI support. Remarkably, the GrGen suite has once provided tool-level support for XMI too, but the functionality was broken in up-to-date versions of the tool. Therefore, explicit XMI related rules had to be written for the PN2SC case.

As a threat to the generalizability of our experimental observations, we highlight that we did not need to perform large changes to the solution designs once they were finished. Therefore, we did not need much tool support for refactoring. However, we do observe – *independently of our experiment* – that the refactoring support for Java is very elaborate in tools such as Eclipse whereas it is not supported at all for GrGen.

5.5 Quantitative Evaluation

Fig. 6 (a) shows that the size the implementations of the core mapping rules only differs by a factor five. The main difference in size (and hence specification effort) relates to user interface (UI) aspects: for the Java solution, all user interface widgets for loading the input model and triggering the transformation is programmed specifically for the case study. The Java code also integrates with the *dot* framework. That code is reused from another tool (see the leftmost bar in Fig. 6 (a)) and therefore it is not counted as part of the solution specific UI code. GrGen programs can rely on a flexible shell that makes user interface code unnecessary. Moreover, GrGen includes a visualization engine with a very concise configuration language. As a result, the Java solution contains about ten times as much case study specific UI code, compared to the GrGen solution. Fig. 6 (a) also shows that GrGen offers a concise language for metamodel definition.

Fig. 6 (b) displays the runtime performance of the two solutions, based on the automatically generated suite of test-model that is discussed in Section 5.1. Although Fig. 6 (b) has a limited scale, we have used test-models of 100, 200, ... elements to about 300.000 elements. All results can be reproduced via an online virtual machine [18]. In general, the performance of both solutions scales linearly for models of normal size (requiring always less than a second.) For models with thousands of elements, the GrGen solution exhibits an $\mathcal{O}(x^2)$ time complexity.

Notice on Fig. 6 (b) that for models consisting of less than 10.000 elements, the time that is required for analyzing the input model and generating optimized code does not outweigh the speedup. Therefore, for small input models one should turn GrGen engine optimizations off. The Java solution cannot process more than about 15.000 elements, due to limitations of the address space of the 32 bits Java virtual machine that we have used. Fig. 6 (b) also indicates that for inputs models with more than 10.000 elements, the required processing time increases significantly too. For models of some hundreds of thousands of elements, the GrGen engine optimizations reach a speedup factor of almost two.

6 Conclusions

From a completeness perspective, both solutions solve the *core mapping problem* of the PN2SC case study adequately for a head-to-head comparison. Their designs are comparable and this paper carefully analyzes their differences. Most importantly, the Java design contains a case-specific optimization algorithm whereas the GrGen solution only relies on engine optimizations. Several *design* decisions could have been made differently by other transformation writers so conclusions about our specific solution sizes should not be taken out of context.

From the perspective of the transformation *language* though, the GrGen solution does have more declarative constructs for pattern specification and rule scheduling. Conversely, Java has more advanced constructs for genericity and modularity. From a cognitive perspective, the GrGen language enables specifications that are closer to the problem domain. This paper also describes the threats related to comparing the proposed Java and GrGen solutions with solutions based on languages with built-in support for traceability, change propagation and bidirectionality. One should extend the Java and GrGen solutions with such features before incorporating them in our quantitative evaluation. Finally, from a transformation *tool* perspective, we observe that the GrGen solution is supported by more usable debugging and visualization tools. Java on the other hand has better support for refactoring.

The Java solution requires more specification effort without yielding a better runtime performance. Remarkably, the Java solution has even been revised for performance reasons. Its mathematical formalization had to co-evolve, leading to considerable overall complexity. The GrGen development did not require such iterations. We therefore conclude that for developing mappings such as the PN2SC translation, rule-based languages and tools are more appropriate.

References

1. M. F. Amstel, C. F. Lange, and M. G. Brand. Using metrics for assessing the quality of ASF+SDF model transformations. In *ICMT '09*, pages 239–248. Springer, 2009.
2. G. Bergmann, Á. Horváth, I. Ráth, and D. Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Tántzer, editors, *ICGT*, volume 5214 of *LNCS*, pages 396–410. Springer, 2008.
3. R. M. Dijkman, M. Dumas, and C. Ouyang. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.
4. M. Dumas. Case study: BPMN to BPEL model transformation. In *5th International Workshop on Graph-Based Tools, Satellite workshop to TOOLS 2009*, 2009.
5. R. Eshuis. Translating safe petri nets to statecharts in a structure-preserving way. In A. Cavalcanti and D. Dams, editors, *FM*, volume 5850 of *LNCS*, pages 239–255. Springer, 2009. Extended as Beta WP 282 at Eindhoven University of Technology.
6. R. Geißand M. Kroll. GrGen.net: A fast, expressive, and general purpose graph rewrite tool. In Rensink and Tántzer [14], pages 568–569.
7. J. Graef. Managing taxonomies strategically. *Montague Institute Review*, 2001.
8. T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
9. A. Habel, R. Heckel, and G. Tántzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.
10. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
11. A. Horváth, G. Bergmann, I. Ráth, and D. Varró. Experimental assessment of combining pattern matching strategies with VIATRA2. *STTT*, 2010.
12. T. Mens and P. Van Gorp. A taxonomy of model transformation. In G. Karsai and G. Tántzer, editors, *Proc. GraMoT 2005*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier, Mar. 2006.
13. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
14. A. Rensink and G. Tántzer, editors. *AGTiVE 2007, Kassel, October 10-12, 2007, Revised Selected and Invited Papers*, volume 5088 of *LNCS*. Springer, 2008.
15. N. Russell, A. ter Hofstede, W. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPMcenter.org, 2006.
16. SAP. Discovering processes, SAP library. http://help.sap.com/SAPHELP_NW04S/helpdata/EN/26/1c6d42ab7fd142e10000000a1550b0/content.htm, 2010.
17. P. Van Gorp. *Model-driven Development of Model Transformations*. PhD thesis, University of Antwerp, Apr. 2008.
18. P. Van Gorp. Online demo: PN2SC in Java and GrGen. <http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdiID=343>, 2010.
19. P. Van Gorp and R. Eshuis. Transforming process models: executable rewrite rules versus a formalized java program. Technical Report Beta WP 315, Eindhoven University of Technology, 2010.
20. P. Van Gorp, S. Mazanek, and A. Rensink. Transformation Tool Contest – Awards. <http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/?page=Awards>, 2010.
21. D. Varró, M. Asztalos, D. Bisztray, et al. Transformation of UML models to CSP: A case study for graph transformation tools. In Rensink and Tántzer [14], pages 540–565.