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

Pieter Van Gorp, Rik Eshuis
p.m.e.v.gorp@tue.nl, h.eshuis@tue.nl

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. 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 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 clear comparison criteria and by making the solutions available for online replay, we illustrate that rule-based approaches enable the transformation writer to focus on what the transformation should produce whereas imperative approaches require a more low-level specification involving element traversal algorithms. A mathematical formalization of such code not necessarily solves that problem. We therefore conclude that for developing mappings such as the PN2SC translation, the use of the GrGen language and tool is more appropriate.

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 [3], the program transformation community [14] and the business process management (BPM) community [27]), 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 instrument for comparing these approaches is the *"taxonomy of model transformation"* [29,30,44]. In contrast to previously published work, this paper classifies the approaches on various levels (the *conceptual*, *design*, the *language*, and the *tool* level) and puts quantitative results in a proper qualitative perspective. Moreover, all results from this paper can be reproduced in an online virtual machine [45].

The taxonomy is based on various workshops and a substantial amount of reflection [29,30,44]. In general, a taxonomy can be used for a wide variety of purposes [19]. We have used this taxonomy for example to provide students a comprehensive overview of the field. It turns out that having some of the taxonomy's terms in mind even helps in decomposing too monolithic transformation problems. Most frequently though, the *"model transformation taxonomy"* is used it to assess the strengths and weaknesses of a new transformation approach (see [48,23,44] for examples).

In this paper, we apply the taxonomy to identify the strengths and weaknesses of two radically different transformation approaches. In this context, the organization in four levels (*conceptual*, *design*, *language* and *tool*) 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. The organization can also be useful to quickly prune candidates from a large set of seemingly similar transformation approaches. In this paper, we do not aim to present the complete taxonomy. We will use the parts that are of interest to our case study and we will clarify new terms where needed but refer to [44] for a more general and detailed coverage.

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 briefly describes these solutions. Section 5 contains the actual evaluation of the solutions. In Section 6, we reflect on our use of the taxonomy. In the two remaining sections, we present our conclusions and outline future work.

2 Related Work

Obviously, this is not the first paper that bridges the BPM and graph transformation communities. On the latest BPM conference for example, Oikawa et al. rely on graph-theoretical concepts to reason about the confluence and termination of their proposed set of transformation rules [35]. From the side of the latest International Graph Transformation Conference (ICGT), there is even a workshop on the relation between petri-nets and graph transformations. As an illustrative example, Biermann et al. show how their graph transformation *tool* can directly execute and verify graph transformation rules for petri-nets [5]. The novelty of our contribution is not the *application* of graph transformation techniques in a BPM context. Instead, it is the detailed comparison of two characteristic approaches that is new.

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 *Unified Modeling Language* (UML) activity diagrams to *Communicating Sequential Processes* (CSP) as a case study to compare 11 graph transformation based approaches [47,6]. 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 the *Business Process Modeling Notation* (BPMN) to the *Business Process Execution Language* (BPEL) as a case study to compare 10 transformation approaches [10]. Our participation in these events has lead to refinements of the taxonomy that supports in this paper. Moreover, this paper learns from these events in the following ways:

- This paper compares a graph transformation system based on the *GrGen*[1] system with a Java based transformation program. For the 2007 edition of the graph transformations tools 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 [1].
- 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 are equally expressive. Therefore, any set of conceptual mapping rules can eventually be implemented using any of the evaluated platforms and the added value of these platforms relates to other properties.

So far unrelated to these events, van Amstel et al. have proposed a set of metrics to quantify the quality of model transformation programs [2]. We are collaborating with van Amstel et al. to integrate that work with the model transformation taxonomy. In summary, that work should be extended with metric implementations for more transformation languages. In the long term, one will be able to (i) use our instrument first, to understand for which artefacts it is fair to make a metric-based comparison. Then, one would automatically compute van Amstel's metrics to *quantify* properties such as *transformation complexity.*

3 Case Study: petri-nets to statecharts

This section introduces the concrete transformation problem that this paper uses to compare a rule-based transformation solution with an imperative transformation program. Subsection 3.2 informally presents the underlying mapping rules that need to be realized by both solutions. Subsection 3.3 characterizes these mapping rules according to the model transformation taxonomy.

[1] We use the short *GrGen* name to denote *GrGen.NET* [16,7].

This paper assumes that the reader is at least aware of petri-nets [32] and statecharts [22] (the language of the input and output models of our case study). We do not assume in-depth knowledge of advanced petri-net analysis methods (such as coverability graphs, incidence matrices, ...) or any knowledge of high level petri-nets (color, time, hierarchy, ...). Similarly, we only assume basic familiarity with statecharts and do not rely on advanced concepts such as event processing. Instead, we present in subsection 3.1 a refresher of the basic Petri-Net execution semantics while introducing our running example. We also introduce statecharts by example and refer to [22] for a general treatment of the semantics.

Having two executable *Petri-Net to Statechart* (PN2SC) implementations is *not* considered a BPM contribution in itself, and we also refer the reader to [12] for a general description of why the proposed mapping makes sense from a theoretical as well as from a practical point of view. In summary however, the relevance of the mapping under consideration is that it makes concurrent regions syntactically explicit. This is for example useful for increasing the (human) understandability of large process models.

3.1 Example Input and Output Model

Fig. 1 is based on the running example from the original paper on the PN2SC mapping [12]. All six diagrams from the figure represent the same input Petri-Net. Black dots visualize so-called *tokens*. In all variants of petri-nets, tokens represent data. Circles visualize so-called *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 so-called *transitions*. A transition represents a process *activity*. When such activities occur, the net moves to a new state, that is: tokens a redistributed across the net. More specifically, a transition t that *fires* moves tokens from the places which have an incoming arc to t (the so-called *input places*) to places which have an outgoing arc from t (the so-called *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.

[12] restricts the class of input petri-nets for *PN2SC* to those which can never reach a state with more than one one token in a place. In the context of such so-called *safe* nets (very similar to so-called *condition/event* nets), places represent *conditions*. A condition is satisfied if the corresponding place holds a token and is not satisfied otherwise. Transitions represent *events*. An enabled transition now corresponds to a transition for which the input places (its conditions) are satisfied (i.e., hold a token).

The net on the top-left of the figure is in the state where only the *P0* holds a token. Therefore, only *T0* is enabled. The top-right diagram from Fig. 1 visualizes the state after firing *T0* (i.e., after the event *T0* occurred). In this state, *T3* and *T4* are enabled. Either transition can occur first but they cannot occur at exactly the same time (transition firing is *isolated* and *atomic*). The middle-left from Fig. 1 diagram shows the state after firing *T3* first whereas the middle-right diagram represents the state after firing *T4* first. The two bottom diagrams from

Fig. 1 visualize the states after firing *T5* (resp. *T6)* after the state visualized by the middle-right diagram.

This example execution trace is not complete but should provide sufficient understanding in the basic execution semantics of the input formalism for the *PN2SC* mapping.

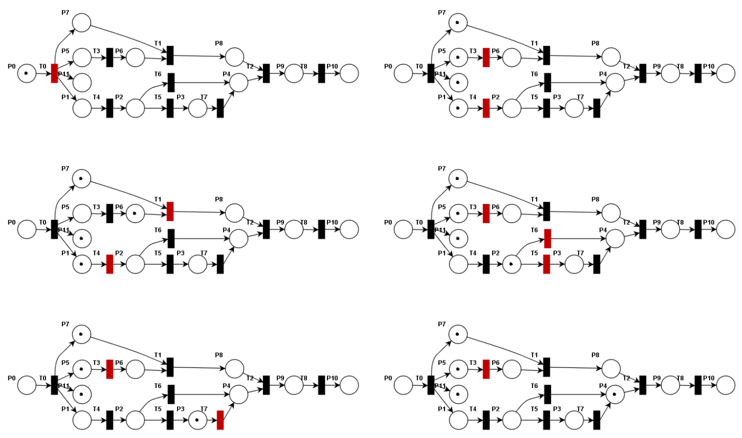


Fig. 1. Input Petri-Net, with fragment of illustrative execution trace.

A key characteristic of petri-nets in general (and the above example in specific) is that it is cumbersome for humans to identify which parts of the process can be active at the same point in time. In the above example, it is clear that the sequences after *T0* can be active concurrently. Then again, the above two sequences merge together at join point *T1*, whereas there is no transition that consumes tokens from *P11*. Additionally, transition *T2* represents a second synchronization point. Only trained and alert minds will quickly see that the arcs that merge in *P4* are unrelated to the parallel decomposition of the process. This effect is investigated empirically by Sarshar and Loos [40]. In summary, petri-nets are at a too low level of abstraction to convey the parallel decomposition of a process to the human mind.

Fig. 2 represents the same process model in *Statechart* syntax. More precisely, this diagram represents the above process model in the "state machine diagram" syntax from the Eclipse *MDT-UML2Tools* plugin [15]. 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 primitive node (called "basic nodes" in [12]).

The black dot that is connected to *P0* represents a so-called *“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 also shows two black dots surrounded by a circle. These dots represent so-called *“final states”*. The transitions from our input Petri-Net are mapped to so-called *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*).

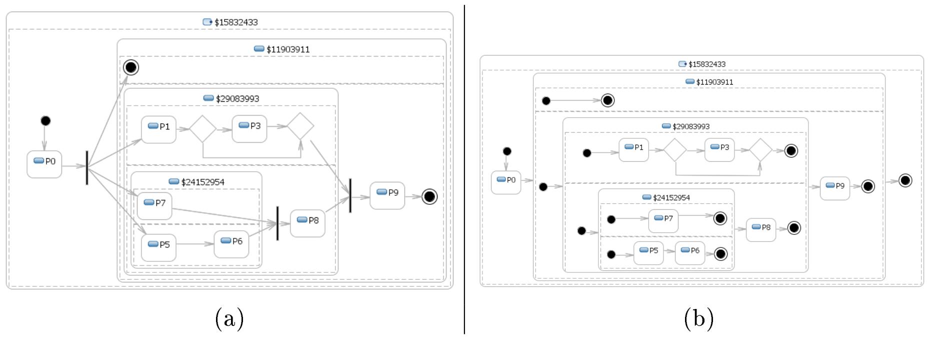
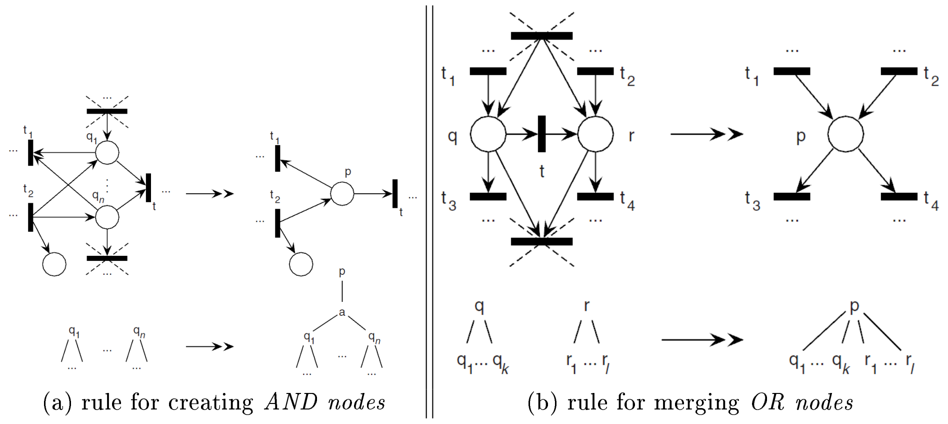


Fig. 2. Statechart representations of the running example.

Clearly, the major added value of representing Petri-Net models as Statechart based models relates to the AND/OR hierarchy: from diagrams such as the one shown on Fig. 2 one can quickly learn which regions can be executed in parallel. The explicit representation of decision nodes and conditional merge nodes can be considered of secondary importance. Similarly, the explicit representation of initial and final nodes adds less value than the hierarchical AND/OR visualization. Therefore, this article defines the “core” of the PN2SC mapping as the set of mapping rules that make that hierarchy explicit.

3.2 Mapping Rules

Fig. 3 is extracted from [12] and shows how mapping rules are typically *documented* in the BPM domain. Such documentation fragments are typically incomplete but their meaning is usually sufficiently clear for guiding an implementation. Other applications of this specification style can be found for example in [32] (to document Petri-Net reduction rules) and [9] (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 (set constructs) and the pseudo-code descriptions that are also contained in publications from that domain are not considered adequate documentation, and (ii) it provides a basis for judging the understandability of the implementations that are compared in this paper.



(a) rule for creating *AND nodes* (b) rule for merging *OR nodes*

Fig. 3. Visual documentation for mapping rules.

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 [11]. 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 [12] for more details. Notice that the complete mapping also contains a rule for *AND splits* (parallel *fork* patterns [38]). That rule resembled 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.

Finally, notice that the *documentation-oriented* rules presented in this subsection are incomplete. For example, [12] does not provide visual documentation for the case where the transition *t* loops back to its input place. Instead, it is accompanied by a technical report that defines the actual mapping rules in pseudo code [13]. Since that accurate (yet non executable) description is already more verbose than the informal rules presented in this section, it should not come as a surpise that the Java and GrGen implementations (cfr., Section 4.2) are also more verbose.

3.3 Decomposition and Classification of the PN2SC Case Study

The primary goal of this paper is to compare a Java based transformation solution with one that is based on graph transformations. As explained in Section 1, there are three levels for comparing transformation solutions (*design*, *language* and *tool* level.) Before comparing the two solutions (or any set of solutions), we need to ensure that the solution fragments under consideration are responsible for exactly the same tasks. Since we want to focus our comparison on the realization of the core mapping rules that were presented in the previous section, we first need to decompose the large PN2SC translation problem into more manageable parts.

PN2SC CASE	core	r./w. PNML	PN MM'	r./w. GRPN	w. GRSC	rm. HE	to UML style	w. XMI
type	translation	reiteration	reiteration	reiteration	reiteration	rephrasing	rephrasing	reiteration
M2M or M2T	M2M	T2M/M2T	M2M	T2M/M2T	M2T	M2M	M2M	M2T
abstraction dim.	vertical, up	horizontal	horizontal	horizontal	horizontal	vertical, up	vertical, up	horizontal
synt. or sem.	syntactical	syntactical	syntactical	syntactical	syntactical	syntactical	syntactical	syntactical

Table 1. Decomposition of the case study and classification of the sub-problems.

Table 1 clearly separates the *core* translation problem from aspects that are realized by the two solutions. The transformation type (*translation*, *reiteration*, *rephrasing*) indicates whether the sub-problem under consideration involves (i) translating model elements to another modeling language, (ii) “blindly" enumerating all model elements, or (iii) rephrasing model fragments using different model element types than those in the input model, but still relying only on constructs from the input modeling language.

The core problem is a translation, since the input modeling language (petrinets) is clearly different from the output modeling language (statecharts). Reading input models in standard Petri-Net Markup Language (PNML) is a reiteration since all elements from the input model should be retrieved verbatim for further processing. This transformation sub-problem is of kind text-to-model

(T2M) since the PNML inputs are stored as XML text whereas the output models are models (sets of elements and relationships, conforming to a metamodel).

Another sub-problem related to the PN2SC case study involves converting PNML inputs to the particular Petri-Net metamodel that is prescribed by [12]. In Table 1, that sub-problem is labeled as *"PN MM'"* and the classification clearly indicates that this sub-problem is of kind model-to-model. It is a simple reiteration of elements that simply changes some syntactic characteristics (typically, a change of metaclass) of the input elements.

Similar observations can be made about the parts of the transformation that deal with other file formats (GRPN[2], GRSC[3] and XMI[4] for serializing/reading petri-nets and respectively statecharts in/from file-formats for graph-based and model-based tools.) The sub-problem that is responsible for mapping statecharts “to UML style" is classified as a *rephrasing*, since it involves changing basic states into decision, start and final nodes. Although that significantly improves the readability of the Statechart models, it is not classified as a translation since the language of the input and output models is the same (UML Statemachines). The row labeled “abstraction dimension” makes explicit that this sub-problem does increase the level of abstraction of models. The column labeled *“rm. HE”* relates to the transformation of Hyper-edge nodes (*fork* and *join* constructs) into unconnected regions. A solution to that sub-problem can transform for example the model from Fig. 2 (a) into that of Fig. 2 (b). Table 1 clearly indicates that this is an extension to the *core* of the case study.

4 Description of Solutions

This section presents and compares the two solutions to the case study. We first indicate which parts of the case study are solved by which solution. Then, we zoom in on the solutions to the *core* sub-problem. Subsection 4.1 considers the completeness of the solutions, Subsection 4.2 shows code fragments to make the subsequent discussion understandable. Finally, Subsections 5.1, 5.2 and 5.3 classify the solutions from the design, language and tool perspective respectively.

4.1 Completeness and Correctness

Table 2 shows that besides the core problem, both solutions tackle a series of other challenges. Several features are support model-level interoperability for benchmarking purposes: the read/write PNML and read/write GRPN features have enabled us to test both solutions on (conceptually) the same set of input models.

A unique feature of the Java solution exporting statecharts to the so-called GRSC file format. This file format is in fact specific to the GrGen tool-suite and

[2] The GRgen Petri-Nets (GRPN) file format is based on the *Varró* benchmark [4].
[3] The GRgen StateCharts (GRSC) file format is syntactically comparable to GRPN.
[4] The XML Metadata Interchange (XMI) format is a metamodel-independent industry standard but we refer to the UML 2.1 variant from Eclipse [15] specifically.

the aforementioned export feature enables us to visualize the results of the Java solution using a tool from the GrGen suite.

The GrGen solution is unique in that it implements several extensions to the basic mapping rules from [12]: 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.

PN2SC Solution	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

Table 2. Completeness of the solutions from a User perspective.

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 [39]), 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 (cfr., Section 5.4.) This type of models was generated by taking as a seed a model from the *Varró* benchmark [4], and performing a clone and merge operations to incrementally compute models of twice the size but with a similar distribution of language constructs.

After fixing some errors, we agreed that the *core* parts of both solutions were adequate for a detailed comparison. For end-users, the Java solution differs from the GrGen solution in that it only produces an output model if a complete output Statechart can be derived. The GrGen solution is more resilient in that it can generate partial results in the case that no valid complete Statechart exists for the input. This occurs for example when the input Petri-Net is not safe. Notice that during process model development, this can occur by mistake and inspecting the partial output of the GrGen solution may help fixing modeling errors.

4.2 Example Code Fragments

Fig. 4 shows a fragment from the Java solution while Fig. 5 shows the corresponding fragment from the GrGen solution.

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. Notice that the Java execution infrastructure has no means to traverse the elements from the *transitions* collection automatically in an order that is optimized for the transformation task under consideration.

In the original Java version, each of the two reduction rules was implemented by a separate method. The reduction procedure was started by invoking the method for rule 2[5]. In the method for rule 2, transitions were processed one by one. For each transition it was checked whether the precondition for rule 2 was satisfied and if so, rule 2 was applied, and the method was called again recursively. If rule 2 was not applicable to any transition, then the method for rule 1[6] was invoked. In this method, each transition was processed to check whether the precondition for rule 1 was satisfied. If so, rule 1 was applied and the method for rule 2 was invoked again. If rule 1 was not applicable to any transition, the complete 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 1 is applied to the preset and postset, and next rule 2. 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 [12,13] too.

Initially, the GrGen solution was implemented based on the Java solution, as described in [12], 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. Apparently, the ordering for processing rules as determined by the GrGen engine is more efficient than a user specified, hard-coded ordering.

In retrospect, the search algorithm is redundant from a conceptual viewpoint, since it only influences the runtime of the Java solution. The pseudo code style used in [12,13] to formalize the reduction procedure facilitated the preservation of the search algorithm in the conceptual algoritm. 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.

A perhaps unconventional feature of the Java solution is that it heavily relies on vector indexing (cfr., lines 5, 8 and 26.) The use of indexed data structures is often driven by performance considerations (see for example [37]) but in this case it could also driven by the structure of the PNML format (in which elements have an XML ID.) Since the formal description of the mapping does not rely on indices [12], one can conclude that the Java solution contains *redundant*

[5] Rule 2 is called *transform_singleton_pre_post* in this paper.

[6] Rule 1 has variants 1a (called *trans_AND_join* in this paper) and 1b.

```
 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).

technical details. A Java solution based on collection iterators would not have that disadvantage.

A third 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 *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 automatic optimization based on dynamic rule scheduling (cfr., [24,18]) virtually impossible.

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 so-called negative application conditions [21]. 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_*-

```
 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).

AND_join rule, the left-hand side (between lines 2 and 4 of Fig. 5) specifies a pattern 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.)

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 (similar to methods or procedures in mainstream programming languages.)

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 (where "[]" for example denotes the *concurrent* execution of a rule on *all* possible matches in the host graph).

Unlike the imperative Java solution discussed above, GrGen programs require no explicit element traversal, nor element indices. Also unlike the Java solution, the GrGen solution does not implement a dedicated search optimization. Instead, it relies on the built-in optimizations from the underlying engine.

5 Evaluation of Solutions

Subsection 5.1 compares the Java and GrGen solution according to design decisions that could have been made differently by other transformation writers.

Subsection 5.2 discusses language specific solution characteristics and Subsection 5.3 compares the related tools. Finally, Subsection 5.4 presents the results of a our quantitative evaluation.

5.1 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 5.4 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.

	Java (core)	GrGen (core)	r./w. PNML (Java)	r./w. GRPN (GrGen)
scope	complete	complete	sub-model	complete
input effect	destructive	destructive	preserving	preserving
in-place or out-place	in-place	out-place	out-place	out-place
endogenous or exogenous	endogenous	exogenous	exogenous	endogenous
intra- or inter- technical space	intra	intra	inter	inter

Table 3. Classification of the transformation designs.

The first row (scope) from Table 3 indicates whether *all* elements (or a controlled subset thereof) from the input model are visited during pattern matching. This classification is important, since a *smaller* and *less complex* transformation specification can be constructed when one can assume that all elements from the input model are candidates for the transformation.

For the *core* challenge, both the Java and the GrGen solution assume that all input model elements are candidate inputs. Notice that it is well possible to change both solutions into the other variant. 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 (e.g., a rule *trans_PetriNet* that would have a parameter of type *Net*. Similar extensions would need to be made to the Java solution. These changes would have little impact on simple size metrics such as *"lines of code"* but significant impact on metrics that were proposed for transformation complexity (such as *val-in/out* and *fan-in/out* [2].)

The second row (input effect) indicates the state of the input model after the application of the transformation:

input-preserving all input elements are unaffected,
input-polluting some *observable* (as defined in the metamodel) properties from the input elements have changed,

input-destructive some (or all) input elements have been removed.

For both solutions, the *core* is input-destructive. This means that these solutions are probably not directly deployable in CASE tools, where the input models typically need to persist also after transformation application.

Notice that we also added columns for other parts than the *core*. This clarifies that the input-destructive nature of the core solutions can be bypassed by introducing some pre- and post-processing steps. More specifically, one can first serialize the input petri-nets to PNML or GRPN text, then parse it as a new model and transform that into a Statechart without destroying the original Petri-Net.

Notice again though that without additional mechanisms, that approach may break model-based traceability. Moreover, when comparing the size of the two *core* solutions to a (hypothetical) third solution that is input-preserving, one would for example need to take into account the size of the PNML related code on top of the size of the *core* code of the Java solution.

A related feature that distinguishes the Java and GrGen solutions is that the Java implementation transforms the input model elements in-place to elements that conform to the output metamodel. In contrast, the GrGen solution creates a new output element for each place, transition and arc from the input model. Therefore, it is classified as *out-place*. For the PN2SC case study, this is relevant since the conceptual mapping rules (as shown on Fig. 3) iteratively *fold* elements. For the Java solution, it is unclear whether Petri-Net or Statechart elements are being folded, especially since the solution is endogenous: the input and output metamodel is the same. For the GrGen solution, it is clear that Petri-Net elements are iteratively folded whereas the output Statechart elements are preserved across the folding operations.

Due to space considerations, we omit a further description of the cells from Table 3 (all taxonomy elements are defined in [44].) The key message from this section is that different design choices will have an impact on the usability of the solution fragments under consideration. In order to make comparisons concerning size and complexity, one first needs to take these differences into account. Moreover, it should be clear that these choices are *design* (not *language*) specific and conclusions should be derived accordingly.

5.2 Language Specific Classification

This section classifies the transformation (programming) languages from both solutions. Once more, the discussion is incomplete (other elements for classifying transformation languages can be found in [8] and [44]) but Tables 4 to 6 do summarize our key findings. The different tables also include rows for languages that perform better than the two solutions presented here, according to the criteria from the respective table. Again, a complete survey of available transformation approaches is far beyond the scope of this paper and the extra rows are primarily intended to stimulate future work on the PN2SC case study.

	change propagation	execution direction	rule scheduling	rule modularization
Java	explicit	explicit (source to target)	explicit	classes, packages, visibility
GrGen	explicit	explicit (source to target)	explicit, implicit	files, no scoping
Story Diagrams [18,31]	explicit	explicit	explicit, implicit	classes, packages, visibility
TGG [41,26]	implicit	implicit	implicit	packages, visibility
QVT/Rel. [33]	implicit	explicit, implicit	implicit	transformations

Table 4. Classification of the transformation languages (1/3).

The first feature column of the Table 4 reveals that neither Java nor GrGen has support for modeling *incremental updates* (a.k.a. *change propagation* in [33] or *target incrementality* in [8]) declaratively. It should be noted that the two solutions presented in this paper also have not explicitly programmed support for incremental updates. Therefore, when quantitatively comparing the size and complexity of these solutions against for example a Triple Graph Grammar (TGG) or QVT/Relations solution, conclusions should be drawn accordingly.

The second feature column reveals that neither Java nor GrGen has support for modeling bidirectional updates declaratively, which again should be taken into account in comparisons against TGG or QVT/R based solutions. Mind that bidirectional QVT/Relations transformations need further research [42]. Based on our knowledge of TGG engines, we predict that a bidirectional TGG implementation of the rules from Section 3 is feasible (at the cost of time and memory resources) and we invite fellow experts to explore this challenge further.

From the perspective of rule scheduling the GrGen language is particularly interesting, since it supports a declarative specification style (leaving it implicit in which order rules will be executed) as well as a more operational style (explicitly defining execution orders.)

From the perspective of rule modularization, the GrGen language clearly needs improvement, since it only supports file-based, unscoped rule modularization. Obviously, concepts such as packages are desirable when solving problems with the complexity of the case presented in this paper.

The *genericity* column from Table 5 indicates that, as a mature programming language, Java has several language constructs for factoring out commonality. GrGen does not provide any of the features mentioned for Java. Within the transformation community, Higher Order Transformation (HOTs) have been proposed as a promising mechanism in the context of genericity [25]. Unfortunately, the GrGen language does not (yet) enable the writing of higher order transformations. To illustrate that there are graph transformation languages without these limitations, Table 5 includes a row for Story Diagrams (that borrow genericity features from Java [18] and also support HOTs [46]).

	genericity	traceability	error-proneness
Java	specialization	explicit	vector operations
	overloading	(-)	(iteration & indexing)
	reflection		
GrGen	none of (see Java)	explicit	matching semantics
	no HOTs	(+)	(PACs↔NACs)
Story Diagrams [18,46]	GrGen issues solved	explicit, implicit	Java/GrGen issues solved
TGG [41,26]	rule specialization	implicit	Java/GrGen issues solved
QVT/Rel. [33]	rule specialization	implicit	*unknown*

Table 5. Classification of the transformation languages (2/3).

Traceability support needs to be programmed explicitly in both the Java as well as the GrGen approach. Concerning the specific solutions compared in this paper, the (-) from Table 5 indicates that the Java solution does not support traceability at all whereas the GrGen solution supports traceability during transformation execution. More specifically, the GrGen solution does maintain covert traceability links between source and target elements but since the solution is input-destructive (as indicated in Table 3), these links are no longer accessible after transformation execution. Table 5 indicates that the TGG and QVT/Relations languages provide more declarative support for traceability.

The *error-proneness* column (as well as all columns from Table 6) is based on the framework of cognitive dimensions by Green and Petre [20]. The error-prone vector operations are discussed in Section 4.2.

For GrGen, the matching semantics may lead to unexpected errors since it is rather uncommon compared to other graph transformation languages. More specifically, elements are by default matched isomorphically in positive as well as negative patterns (PACs and NACs). However, when embedding NACs in PACs (or other NACs), variables in the NAC pattern can be bound again to elements that are already bound to other variables. Although this semantics is properly documented [7], it has caused mistakes during the construction of our GrGen solution.

Fig 6 illustrates the fragment from the GrGen solution that we consider the most difficult to understand for transformation writers with a general computer science education background (i.e., with knowledge of mainstream imperative and functional languages.) Rule *transform_singleton_pre_post* implements the conceptual rule shown on the right of Fig. 3 (the variable names p, q, r, and t correspond between Fig 6 and Fig. 3.)

Notice for example the *hom* statements from line 6 to line 8. This expresses that the involved pairs of variables can be bound *homomorphically*. This is required to make the rule *transform_singleton_pre_post* applicable also for the case where the transition *t* starts and ends in one and the same place. All other variables are matched using the default *isomorphic* semantics.

```
1  rule transform_singleton_pre_post {
2    q:Place -pre1:pre-> t:Transition
3                           -post1:post-> r:Place;
4    q -tl1:PN2HSC-> or1:HSCorState;
5    r -tl2:PN2HSC-> or2:HSCorState;
6    hom(q,r);
7    hom(or1,or2);
8    hom(tl1,tl2);
9    negative {
10     i1:Place -:pre-> t
11              <-:pre- i2:Place; // singleton pre
12   }
13   negative {
14     o1:Place <-:post- t
15              -:post-> o2:Place; // singleton post
16   }
17   :NoForkJoinOnDifferentPlaces(q,r);
18   modify {
19     p:Place -:PN2HSC-> or4:HSCorState;
20     delete(pre1);
21     delete(post1);
22     delete(t);
23     exec(
24       [move_incoming_arcs(q,p)]
25     | [move_outgoing_arcs(q,p)]
26     | [move_incoming_arcs(r,p)]
27     | [move_outgoing_arcs(r,p)]
28     | moveAllOrChildrenFromTo(or1,or4)
29     | moveAllOrChildrenFromTo(or2,or4)
30     | cleanupP(q) | cleanupP(r)
31     | cleanupS(or1) | cleanupS(or2)
32     );
33   }
34 }
35 pattern NoForkJoinOnDifferentPlaces(prePl:Place,
        postPl:Place) {
36   alternative {
37     SelfLoopsUnconstrained {
38       hom(prePl,postPl);
39       if { prePl == postPl; }
40     }
41     ConstrainedOtherwise {
42       negative { // no JOIN from input/output
              places
43         prePl -:pre-> t2:Transition
44                      <-:pre- postPl;
45       }
46       negative { // no FORK to input/output places
47         prePl <-:post- t2:Transition
48                       -:post-> postPl;
49       }
50     }
51   }
52 }
```

Fig. 6. Example of error-proneness in GrGen.

As mentioned before, the exception to this default is that variables within embedded NACs are *hom* by default with all variables that were introduced so far. Consequently, in order to express that besides q (cfr., line 2) there is no other input place for transition t, it is *not* sufficient to specify *"negative { qOther: Place -:pre-> t }"* (since *qOther* is allowed to be bound to the same element as q). Fortunately, the correct solution is intuitive once the relation between NAC variables and others is understood: lines 9 to 12 can simply be interpreted as *"it should not occur that t has two or more incoming arcs"*. Then again, this detail may cause errors for transformation writers that are inexperienced with GrGen.

A perhaps more problematic code fragment is that of subpattern *NoForkJoinOnDifferentPlaces* (cfr., lines 35 to 52): this subpattern is used to express that for transitions between different places, a NAC should be enforced, whereas transitions that go to the one and same place as the one they start from (i.e., self-loop transitions) are unconstrained. We find lines 37 to 40 rather cumbersome, since the verbose *hom* and *if* combination would not be required when the matching semantics were homomorphic by default. Instead, the *ConstrainedOtherwise* subpattern would include *"if { prePl <> postPl }"*. Notice that in a mathematical specification[7] (or a specification in any mainstream programming language) two variables are also allowed to be bound to the same element unless specified otherwise. Remark that there are also graph transformation languages that have homomorphic matching semantics by default.

[7] The mathematical definition of the application condition for rule *transform_singleton_pre_post* is: $\exists t \in T(\exists q, r \in P(\bullet t = \{q\} \wedge t\bullet = \{r\} \wedge (q \neq r \rightarrow \neg\exists t' \in T(q, r \in \bullet t' \vee q, r \in t\bullet)))$, where T represents the set of input Transitions, P represents the set of input Places and $\bullet t$ and $t\bullet$ represent the pre- and post- arcs of t respectively.

	closeness of mapping	hard mental operations	pre-mature commitment	secondary notation
Java	farther	rule scheduling	explicit & fixed matching strategy	code comments only
GrGen	closer	control flow	order of textual statements	code comments only
Story Diagrams	same as GrGen	GrGen issue solved	GrGen issue solved	color, layout

Table 6. Classification of the transformation languages (3/3).

Table 6 outlines some other elements that are based on the cognitive dimensions framework. To understand why graph transformation languages enable a closer mapping between the transformation solutions and the descriptions from the problem domain, please do not take the detailed remarks related to Fig. 6 out of context and refer back to the discussion surrounding Fig. 4 and 5. That discussion illustrates that the GrGen solution does not require variables beyond those that are used in the informal mapping rules from Fig. 3 too. In contrast, the Java solution requires for example various variables for array indexing.

Also notice the GrGen cell for pre-mature commitment: indeed the order of the textual statements is a form of over-specification that can be avoided in (semantically comparable, yet) *visual* languages such as Story Diagrams. As an example, the *handlePrePlace_AND_join* rule shown on the right side of Fig. 5 contains a left-hand side consisting of three textual statements. When using a visual language, one could simply model one integrated pattern (the order of the textual statements does not matter anyhow.) Additionally, one could use layout and color to convey some additional semantical clues (use of the co-called *"secondary notation"* feature).

5.3 Tool Specific Classification

The first two feature columns from Table 7 rate those tool features that we have found most useful during the development of the PN2SC solutions. The rows then indicate how usable that feature was for a particular tool (such as *Eclipse* for Java). The first cell for example indicates that in the context of the PN2SC case study, we classify the debugger usability of Eclipse as low. The reasons are that (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 that (ii) the Eclipse debugger shows irrelevant technical variables on the same level as pattern variables.

The second cell on the first row indicates that Eclipse also does not provide a generic visualizer for the transformed models. The Java solution does include integration code for the dot framework but since Eclipse is unaware thereof there is for example no integration with the Eclipse debugger.

	Debugger Usability	Visualizer Usability	Model Interface	Standards
Java (Eclipse)	low	*no visualizer*	serialized	none
			in-memory	
GrGen (GrShell, yComp)	high	high	serialized	XMI (-)
Story Diagrams (Fujaba)	high	medium	serialized	XMI
			in-memory	JMI, EMF

Table 7. Classification of the transformation/programming tools.

The two last columns from Table 7 are used to classify the tools according to two tool features that relate to industrial scale model management. The *model interface* column reveals that the GrGen tool-suite (consisting of a scripting shell and a visualization component) does not support working with models that reside in-memory (e.g., loaded in the repository of a commercial CASE tool). As long as such a repository would be written in Java, the Eclipse tool does support that. Notice that other graph transformation tools (e.g., Fujaba [34]) also do support that.

The *Standards* column reveals that neither of the two evaluated tools provided usable facilities for working with model oriented standards such as XMI or MOF. Although such a feature has once been implemented for GrGen [17], the functionality was broken in up-to-date versions of the tool.

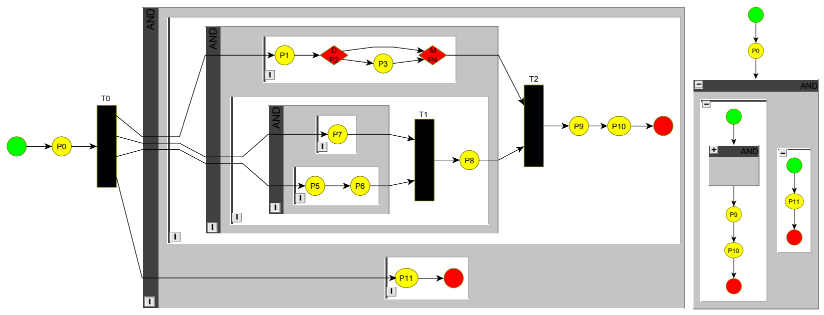


Fig. 7. Output for the running example, produced by the GrGen Solution.

As a brief illustration of the usability of the GrGen visualizer, consider Fig. 7. The figure shows the GrGen *yComp* visualization of the statechart output for our running example. The complete concrete syntax has been defined in a configuration file of about 40 lines. The left side of the figure shows the output after applying the core transformation (to introduce the AND states with their OR regions) and the mapping to UML style (to introduce the initial and final nodes.) The right side of the figure shows a feature of the yComp tool as well as the result of the *"rm. HE"* which is discussed in Section 3.3. The interesting yComp feature shown there is that hierarchical nodes can be (un)folded. This feature requires no effort from the transformation writer and is highly useful when exploring larger models.

As a threat to the generalizability of our experiment, we highlight that we did not have to perform large changes to the solution designs once they were finished. Therefore, we did not need much tool support for refactoring. Therefore, we indicate – *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.

5.4 Quantitative Evaluation

The previous sections enable the reader to assess the two solutions from a qualitative perspective. With that assessment in mind, the reader is invited to further explore the quantitative evaluation from this section. Subsection 5.4 presents the size of the evaluated solutions while Subsection 5.4 presents their runtime performance for a large set of test models. A qualitative evaluation of the solution complexity is desirable too but the related metrics from Amstel et al. (see [2]) are not yet applicable in a Java/GrGen context.

Size of Solutions Fig. 8 (a) shows the size of the two solutions, in terms of their lines of code (LOC). Obviously, more complex measurements are conceivable but only a LOC measurement is feasible today. Conceptually, one could also compare the *number of functions* (see [2]). In that case, one would treat Java methods and GrGen patterns/rules as functions. One would also need to quantify the size of the functions. That is far outside the scope of this work, also due to the unavailability of tools that automate the computation of the metrics.

Fig. 8 (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 a visualization framework. That code is reused from another tool (see the leftmost bar in Fig. 8 (a)) and therefore it is not counted as part of the solution specific UI code. As mentioned in Section 5.3, GrGen programs can rely on a flexible shell that makes user interface code unnecessary. Moreover, GrGen includes a configurable visualization engine, as mentioned in Section 5.3 too. As a result, the Java solution contains about ten times as much case study specific UI code, compared to the GrGen solution.

Fig. 8 (a) also shows that GrGen offers a very concise language for metamodel definition. Remark that this language may be hard to read, compared to popular alternatives such as KM3 or MOF but this did not cause problems in practice.

Performance Evaluation of Solutions Fig. 9 and Fig. 5.4 (b) display the runtime performance of the two solutions, based on the automatically generated

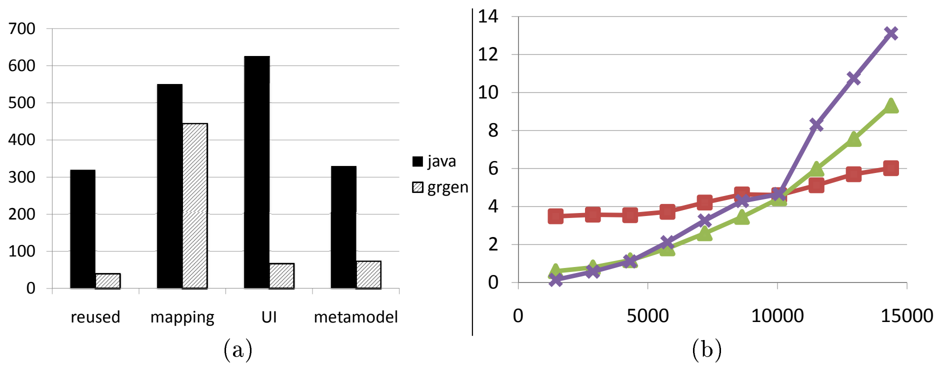


Fig. 8. Size (LOC) and execution time of the solutions.

suite of test-model that is discussed in Section 4.1. As shown on the horizontal axis, these test-models vary in size, from about 100 elements (left of Fig. 9) to about 300.000 elements (right of Fig. 9.) Remark that we have used a standard desktop machine for running the performance tests.

On the graph from Fig. 8 (b) as well as on both graphs from Fig. 9, the purple curve (with cross-shaped data points) represents the Java solution. The green curve (with triangular data points) represents the GrGen solution with engine optimizations turned off. The red curve (with square data points) represents the GrGen solution with engine optimizations turned on. Notice on Fig. 8 (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. This effect is even more visible on the left graph from Fig. 9. Therefore, for small input models one should turn 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. Notice that Fig. 8 indicates that for inputs models with more than 10.000 elements, the required processing time increases significantly as well. The graph on the right of Fig. 9 shows how the GrGen solution scales beyond models of that size: for huge models, the engine optimizations always give a speedup factor of almost two.

To give an idea of the scalability, we remark that the (engine-optimized) transformation requires 600 seconds (10 minutes) for models of about 150.000 elements whereas it requires about 40 minutes for a model of twice that size. More generally, one can draw the following conclusions: the performance of both solutions scales linearly for models of normal size (requiring always far less than a second.) For models with thousands of elements, the GrGen solution exhibits an $\bigcirc(x^2)$ time complexity.

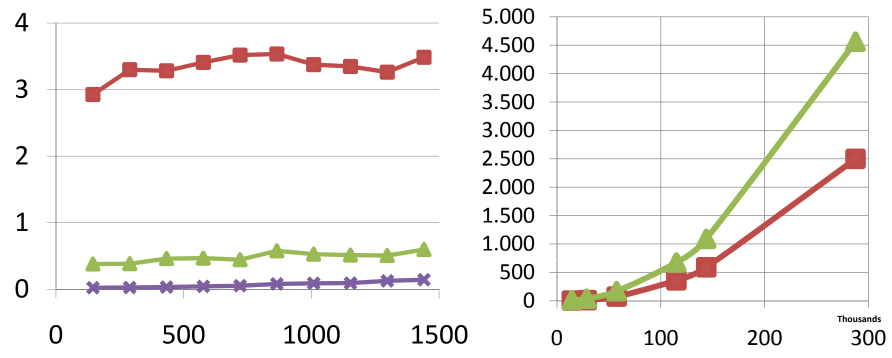


Fig. 9. Execution time of the solutions (number of input elements, seconds.)

6 Discussion

According to the knowledge of the authors, this is the first paper that investigates the strengths and weaknesses of two transformation approaches while preserving a clear separation between *conceptual*, *design*, *language*, and *tool* elements. In many other publications, these elements are treated together, which makes it difficult to assess the generalizability of the results. The main reason for discussing *conceptual* elements (e.g., classifying the core conceptual mapping rules) are that:

1. it emphasizes that both implementations should realize the same conceptual mapping rules (cfr., Fig. 3),
2. it helps decomposing a set of seemingly intermingled features into clearly separated sub-challenges (cfr., Table 1),
3. it enables a comparison of the solutions from an end user-oriented perspective (cfr., Table. 2.)

In relation to (1), we emphasize that (hypothetically other) solutions that rely on different mapping rules should not draw unfair conclusions about elements from other taxonomy levels (design, language and tool). For example, it is quite easy to change the conceptual mapping rules such that the folded elements are *annotated*, rather than *removed*. This enables one to derive a straight-forward design that is *not* input-destructive anymore, for both Java and GrGen. When the conceptual mapping rules are not allowed to be changed in such a manner, it becomes much more challenging to develop a small and simple design that is not input-destructive.

The main reason for discussing design elements is that one should separate the skills of the transformation writer from the potential of the transformation language and tool. Our primary driver for discussing language and tool elements separately is that the Transformation Tool Contest has learned us that several languages are semantically almost equivalent and very often new languages are supported by inferior, *ad-hoc*, tools. Classifications such as the one presented in this paper may help in deciding which features from languages (or tools) should be adopted by (or from) other approaches. After learning from one another, some research initiatives could be merged. This may be essential for disseminating knowledge and skills across communities and towards industry.

This paper illustrates that it is very hard to provide a holistic view of all four perspectives within the page limit of a typical research paper. Therefore, we underline the importance of adopting standard terminology. Moreover, in future work we may dedicate a complete paper to one or two perspectives and refer to this paper to provide an integrated view.

7 Conclusions

From a completeness perspective, both the Java solution and the GrGen solution solve the *core* mapping problem of the PN2SC case study adequately for a head-to-head comparison.

From a solution design perspective, both approaches share the property that they destroy the input Petri-Net while producing the output Statechart model. Unlike the GrGen solution, the Petri-Net solution does not provide traceability information during transformation execution. This introduces a risk for a metrics-based comparison of these solutions in the sense that the Java solution would become larger (or more complex) when extending it with such traceability support. On the other hand, the Java design contains an explicit optimization whereas the GrGen solution only relies on engine optimizations. Finally, one can remark that only the GrGen design is exogenous whereas the Java design is endogenous. Typically, exogenous approaches require more specification effort, since two metamodels need to be defined. All these design decisions could have been made differently by other transformation writers. We emphasize that such decisions influence a quantitative (metrics-based) evaluation.

Moreover, some other languages provide built-in support for more advanced designs. This paper illustrates this by comparing Java and GrGen with other transformation languages: we explain for example why it would be unfair to compare the size or complexity of the Java and GrGen solutions with that of a (hypothetical) TGG solution, since the latter would for example support change propagation.

Both Java and GrGen enforce some hard mental operations and both languages are error-prone. However, this paper also explains why the GrGen language enables a solution specification that is closer to the problem domain. 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 much better support for refactoring.

With all these considerations in mind, one can put the results of our quantitative analysis in perspective. The Java solution requires more specification effort without yielding a better runtime performance. Therefore, for developing mappings such as the PN2SC translation, the use of the GrGen language and tool is more appropriate.

8 Future Work

Due to the lack of more literature and experience, we cannot identify yet which properties of the PN2SC case could potentially favor the GrGen approach (if that is the case at all.) Nevertheless, in our future work, we aim to investigate mapping problems for which imperative programming languages are allegedly more appropriate. We will characterize solutions again using the proposed four-level framework. As more experimental results become available, we aim to investigate correlations between problem characteristics and solution techniques. Such insights should enable the development of guidelines for selecting (or developing new) designs, languages and tools.

Additionally, we learn from the observation that the GrGen engine optimizations are especially relevant for transforming huge graphs: we aim to apply GrGen on process mining problems, since that involves the transformation of huge logs of dynamic information [43]. Moreover, in rather unstructured domains such as health care [28,36], there is a still a clear need for better abstraction rules and the use of declarative languages may speed up the development thereof.

Finally, it should be investigated how graph transformation languages with a visual syntax can be mapped to GrGen (or other textual languages with the appropriate semantics.) That is already known to be useful for transformation language integration [46], and this paper indicates that it is also useful for a cheap (yet effective) comparison of the specification effort (via the LOC metric.)

References

1. Transformation Tool Contest – Awards. `http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/?page=Awards`, April 2010.
2. Marcel F. Amstel, Christian F. Lange, and Mark G. Brand. Using metrics for assessing the quality of asf+sdf model transformations. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 239–248, Berlin, Heidelberg, 2009. Springer-Verlag.
3. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
4. Gábor Bergmann, Ákos Horváth, István Ráth, and Dániel Varró. A benchmark evaluation of incremental pattern matching in graph transformation. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2008.
5. E. Biermann, C. Ermel, T. Modica, and P. Sylopp. Implementing Petri Net Transformations using Graph Transformation Tools. In P. Baldan and B. König, editors, *Proc. Third International Workshop on Petri Nets and Graph Transformations.* EC-EASST, 2008. To appear.
6. Dénes Bisztray. Verification of architectural refactorings: Rule extraction and tool support. In *ICGT'08: Proceedings of the 4th international conference on Graph Transformations*, pages 475–477, Berlin, Heidelberg, 2008. Springer-Verlag.
7. Jakob Blomer, Rubino Geiß, and Edgar Jakumeit. The GrGen.NET user manual, refers to release 2.6 beta 8. `http://www.info.uni-karlsruhe.de/software/grgen/GrGenNET-Manual.pdf`, April 2010.

8. K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.
9. Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in bpmn. *Inf. Softw. Technol.*, 50(12):1281–1294, 2008.
10. Marlon Dumas. Case study: Bpmn to bpel model transformation, 2009.
11. Hartmut Ehrig, Karsten Ehrig, Juan de Lara, Gabriele Täntzer, Dániel Varró, and Szilvia Varró-Gyapay. Termination criteria for model transformation. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.
12. Rik Eshuis. Translating safe petri nets to statecharts in a structure-preserving way. In Ana Cavalcanti and Dennis Dams, editors, *FM – Formal Methods, Second World Congress*, volume 5850 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2009.
13. Rik Eshuis. Translating safe petri nets to statecharts in a structure-preserving way. Technical Report Beta Working Paper 282, Eindhoven University of Technology, 2009.
14. M. S. Feather. A survey and classification of some program transformation approaches and techniques. In *The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation*, pages 165–195, Amsterdam, The Netherlands, The Netherlands, 1987. North-Holland Publishing Co.
15. Tatiana Fesenko, Tatiana Fesenko, Michael Golubev, Lars Vogel, Sergey Gribovsky, Kenn Hussey, and Nick Boldt. MDT-UML2Tools.
16. Rubino Geißand Moritz Kroll. GrGen.net: A fast, expressive, and general purpose graph rewrite tool. pages 568–569, 2008.
17. Tom Gelhausen, Derre Bugra, and Rubino Geiš. Customizing grgen.net for model transformation. In *Workshop on Graph and Model Transformation (GraMoT)*, 2008. accepted for publication.
18. Holger Giese, Stephan Hildebrandt, and Andreas Seibel. Feature report: Modeling and interpreting EMF-based story diagrams. In *Proceedings of the 7th International Fujaba Days*. Technische Universiteit Eindhoven, November 2009.
19. Jean Graef. Managing taxonomies strategically. Montague Institute Review, March 2001.
20. T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.
21. Annegret Habel, Reiko Heckel, and Gabriele Täntzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3-4):287–313, 1996.
22. David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
23. Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Cope - automating coupled evolution of metamodels and models. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag.
24. 'Akos Horv'ath, G'abor Bergmann, Istv'an R'ath, and D'aniel Varr'o. Experimental assessment of combining pattern matching strategies with VIATRA2. 2010.
25. Ákos Horváth, Dániel Varró, and Gergely Varró. Automatic generation of platform-specific transformation. *Info-Communications-Technology*, LXI(7):40–45, 2006.
26. Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 285–294. ACM, 2007.

27. Niels Lohmann, Eric Verbeek, and Remco Dijkman. Petri net transformations for business processes — a survey. pages 46–63, 2009.
28. R. S. Mans, Helen Schonenberg, Minseok Song, Wil M. P. van der Aalst, and Piet J. M. Bakker. Application of process mining in healthcare - a case study in a dutch hospital. In Ana L. N. Fred, Joaquim Filipe, and Hugo Gamboa, editors, *BIOSTEC (Selected Papers)*, volume 25 of *Communications in Computer and Information Science*, pages 425–438. Springer, 2008.
29. Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. 04101 discussion – a taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
30. Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In Gabor Karsai and Gabriele Täntzer, editors, *Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152 of *Electronic Notes in Theoretical Computer Science*, pages 125–142. Elsevier, March 2006.
31. Bart Meyers and Pieter Van Gorp. Towards a hybrid transformation language: Implicit and explicit rule scheduling in story diagrams. In *Sixth International Fujaba Days*, Dresden, September 2008.
32. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
33. Object Management Group. Meta object facility (mof) 2.0 query/view/transformation, v1.0. `http://www.omg.org/spec/QVT/1.0/`, April 2008.
34. University of Paderborn, University of Kassel, University of Darmstadt, and University of Bayreuth. Fujaba Tool Suite. `http://www.fujaba.de/`, April 2010.
35. Márcio K. Oikawa, ao E. Ferreira, Jo Simon Malkowski, and Calton Pu. Towards algorithmic generation of business processes: From business step dependencies to process algebra expressions. In *BPM '09: Proceedings of the 7th International Conference on Business Process Management*, pages 80–96, Berlin, Heidelberg, 2009. Springer-Verlag.
36. Silvana Quaglini. Process mining in healthcare: A contribution to change the culture of blame. In Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 308–311. Springer, 2008.
37. Christopher Riley, Siddhartha Chatterjee, and Rupak Biswas. High-performance java codes for computational fluid dynamics. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 143–152, New York, NY, USA, 2001. ACM.
38. N. Russell, A.H.M. ter Hofstede, W.M.P. van der Aalst, and N. Mulyar. Workflow control-flow patterns: A revised view. Technical Report BPM-06-22, BPMcenter.org, 2006.
39. SAP AG. Discovering processes, SAP library. `http://help.sap.com/SAPHELP_NW04S/helpdata/EN/26/1c6d42ab7fd142e10000000a1550b0/content.htm`, April 2010.
40. Kamyar Sarshar and Peter Loos. Comparing the control-flow of epc and petri net from the end-user perspective. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 434–439, 2005.
41. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science WG*

1994, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1995.

42. Perdita Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of 10th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2007*, volume 4735, pages 1–15. Springer LNCS, October 2007.
43. W. M. P. van der Aalst and A. J. M. M. Weijters. Process mining: a research agenda. *Comput. Ind.*, 53(3):231–244, 2004.
44. Pieter Van Gorp. *Model-driven Development of Model Transformations.* PhD thesis, University of Antwerp, April 2008.
45. Pieter Van Gorp. Online demo: PN2SC in Java and GrGen. `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdiID=339`, February 2010.
46. Pieter Van Gorp, Anne Keller, and Dirk Janssens. Transformation language integration based on profiles and higher order transformations. pages 208–226, 2009.
47. Dániel Varró, Márk Asztalos, Dénes Bisztray, Artur Boronat, Duc-Hanh Dang, Rubino Geiß, Joel Greenyer, Pieter Gorp, Ole Kniemeyer, Anantha Narayanan, Edgars Rencis, and Erhard Weinell. Transformation of UML models to CSP: A case study for graph transformation tools. pages 540–565, 2008.
48. Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In *ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations*, pages 152–167, Berlin, Heidelberg, 2008. Springer-Verlag.