Automated Formal Verification of Model Tranformations[⋆]

Dániel Varró and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521 Budapest, Magyar tudósok körútja 2.
`{varro,pataric}@mit.bme.hu`

Abstract. As the Model Driven Architecture (MDA) relies on complex and highly automated model transformations between arbitrary modeling languages, the quality of such transformations is of immense importance as it can easily become a bottleneck of a model-driven design process. Automation surely increases the quality of such transformations as errors manually implanted into transformation programs during implementation are eliminated; however, conceptual flaws in transformation design still remain undetected. In this paper, we present a meta-level and highly automated technique to formally verify by model checking that a model transformation from an arbitrary well-formed model instance of the source modeling language into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrate the feasibility of our approach on a complex mathematical model transformation from UML statecharts to Petri nets.

Keywords: model transformation, graph transformation, model checking, formal verification, MDA, UML statecharts, Petri nets.

1 Introduction

Nowadays, the Model Driven Architecture (MDA) of the Object Management Group (OMG) has become the dominating trend in software engineering. The core technology of MDA is the Unified Modeling Language (UML), which provides a standard way to build first a *platform independent model (PIM)* of the target system under design, which may be refined afterwards into several *platform specific models (PSMs)*. Finally, the *target application code* should be generated automatically by off-the-shelf UML CASE tools directly from PSMs.

While MDA puts the stress on a precise object-oriented modeling language (i.e., UML) as the core technology, it fails to sufficiently emphasize the importance of precise and highly automated *model transformations* for designing and implementing mappings from PIMs to PSMs, or PSMs to application code. The methodology (if there is any) behind existing code generators integrated into off-the-shelf UML CASE tools relies

[⋆] This work was partially carried out during the visit of the first author to the University of Paderborn (Germany), and it was supported by the SegraVis Research Network, the Hungarian Information and Communication Technologies and Applications Grant (IKTA 065/2000), and the Hungarian National Scientific Foundation Grant (OTKA 038027)

on textual programming language translations, which does not scale up for the needs of a UML based visual modeling environment. Moreover, PIM-to-PSM mappings are frequently hard wired into the UML tool thus it is almost impossible to be tailored to special requirements of applications.

In case of dependable and safety critical applications, further model transformations are necessitated to map UML models into various mathematical domains (like Petri nets, dataflow networks, transition systems, process algebras, etc.) to (i) define formal semantics for UML in a denotational way [2, 7, 8, 16], and/or (ii) carry out formal analysis of UML designs [4, 13].

In the current paper, we investigate the model transformation problem from a general perspective, i.e., to specify how to transform a well-formed instance of a source modeling language (which is typically UML in the context of MDA) into its equivalent in the target modeling language (which can be UML, a target programming language, or a mathematical modeling language).

Related work in model transformations Model transformation methodologies have been under extensive research recently. Existing model transformation approaches can be grouped into two main categories:

– *Relational approaches*: these approaches typically *declare a relationship* between objects (and links) of the source and target language. Such a specification typically based upon a metamodel with OCL constraints [1, 11, 17].
– *Operational approaches* these techniques *describe the process* of a model transformation from the source to the target language. Such a specification mainly combines metamodeling with (c) graph transformation [5, 8, 9, 14, 27], (d) triple graph grammars [22] (e) term rewriting rules [28], or (f) XSL transformations [6, 19].

Many of the previous approaches already tackle the problem of automating model transformations in order to provide a higher quality of transformation programs compared with manually written ad hoc transformation scripts.

Problem statement However, automation alone cannot protect against conceptual flaws implanted into the specification of a complicated model transformation. Consequently, a mathematical analysis carried out on the UML design after an automatic model transformation might yield false results, and these errors will directly appear in the target application code.

As a summary, it is crucial to realize that *model transformations themselves can also be erroneous* and thus becoming a quality bottleneck of MDA. Therefore, prior to analyzing the UML model of a target application, we have to prove that the model transformation itself is free of conceptual errors.

Correctness criteria of model transformations Unfortunately, due to their wide range of applications in the MDA environment, it is hard to establish a single notion of correctness for model transformations. The most elementary requirements of a model transformation are syntactic.

– The minimal requirement is to assure **syntactic correctness**, i.e., to guarantee that the generated model is a syntactically well–formed instance of the target language.

– An additional requirement (called **syntactic completeness**) is to completely cover the source language by transformation rules, i.e., to prove that there exists a corresponding element in the target model for each construct in the source language.

However, in order to assure a higher quality of model transformations, at least the following *semantic requirements* should also be addressed.

– **Termination:** The first thing we must also guarantee is that a model transformation will terminate. This is a very general, and modeling language independent semantic criterion for model transformations.
– **Uniqueness (Confluence, functionality):** As non-determinism is frequently used in the specification of model transformations (as in the case of graph transformation based approaches) we must also guarantee that the transformation yields a unique result. Again, this is a language independent criterion.
– **Semantic correctness (Dynamic consistency):** In theory, a straightforward correctness criterion would require to prove the semantic equivalence of source and target models. However, as model transformations may also define a *projection* from the source language to the target language (with deliberate loss of information), semantic equivalence between models cannot always be proved. Instead we define *correctness properties* (which are typically transformation specific) *that should be preserved by the transformation*.

Unfortunately, related work addressing these correctness criteria of model transformations is very limited. Syntactic correctness and completeness was attacked in [27] by planner algorithms, and in [10] by graph transformation. Recently in [15], sufficient conditions were set up that guarantee the termination and uniqueness of transformations based upon the static analysis technique of critical pair analysis [12].

However, no approaches exist to reason about the semantic correctness of model transformations. To be precise, the CSP based approach of [9] that aims to ensure dynamic consistency of UML models has the potential to be extended to reason about properties of transformations. However, defining manually the semantics of an arbitrary modeling language by mapping it into CSP is much more difficult and less intuitive than defining the operational semantics of the language by graph transformation.

Our contribution In this paper, we present a meta-level and highly automated framework (in Sec. 2) to formally verify by model checking that a model transformation from an arbitrary well-formed model instance of the source modeling language (specified by metamodeling and graph transformation techniques) into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrate the feasibility of our approach (in Sec. 3) on verifying a semantic property of a complex model transformation from UML statecharts to Petri nets.

2 Automated Formal Verification of Model Transformations

We present an automated technique to formally verify (based on the model checking approach of [24]) the correctness of the model transformation of a specific source model into its target equivalent with respect to semantic properties.

2.1 Conceptual overview

A conceptual overview of our approach is given in Fig. 1 for a model transformation from an fictitious modeling language A (which will be UML statecharts for our demonstrating example later on) to B (Petri nets as in our case).

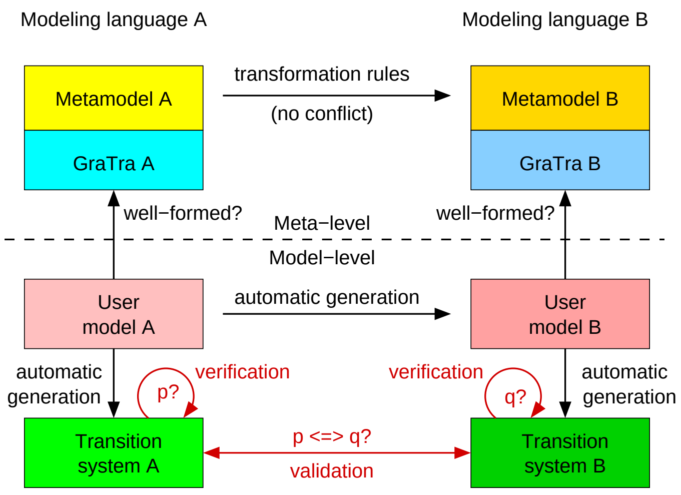


Fig. 1. Model level formal verification of transformations

1. **Specification of modeling languages**. As a prerequisite for the framework, each modeling language (both A and B) should be defined precisely using metamodeling and graph transformation techniques (see, for instance, [26] for further details).
2. **Specification of model transformations**. Moreover, the A2B model transformation should be specified by a set of (non-conflicting) graph transformation rules.
3. **Automated model generation**. For any specific (but arbitrary) well-formed model instance of the source language A, we derive the corresponding target model by automatically generated transformation programs (e.g., generated by VIATRA [5] as tool support).
4. **Generating transition systems**. As the underlying semantic domain, a behaviorally equivalent transition system is generated automatically for both the source and the target model on the basis of the transformation algorithm presented in [24] (and with a tool support reported in [21]).
5. **Select a semantic correctness property**. We select (one or more) semantic property p in the source language A which is structurally expressible as a graphical pattern composed of the elements of the source metamodel (and potentially, some temporal logic operators).
   Note that the formalization of these criteria for a specific model transformation is not at all straightforward. In many cases, we can reduce the question to a reachability problem or a safety property, but even in this case finding the appropriate

temporal logic formulae is non-trivial. More details on using graphical patterns to capture static well-formedness properties can be found, e.g., in [10].

6. **Model check the source model**. Transition system A is model-checked automatically (by existing model checker tools like SAL [3] or SPIN) to prove property p. This model checking process should succeed, otherwise (i) there are inconsistencies in the source model itself (a *verification* problem occurred), (ii) our informal requirements are not captured properly by property p (a *validation* problem occurred), or (iii) the formal semantics of the source language is inappropriate as a counter example is found which should hold according to our informal expectations (another *validation* problem).
7. **Transform and validate the property**. We transform the property p into a property q in the target language (manually, or using the same transformation program). As a potentially erroneous model transformation might transform incorrectly the property p in to property q, domain experts should validate that property q is really the target equivalent of property p or a strengthened variant.
8. **Model check the target model**. Finally, transition system B is model-checked against property q.
   – If the verification succeeds, then we conclude that the model transformation is correct with respect to the pair (p,q) of properties for the specific pairs of source and target models having semantics defined by a set of graph transformation rules.
   – Otherwise, property p is not preserved by the model transformation and debugging can be initiated based upon the error trace(s) retrieved by the model checker. As before, this debugging phase may fix problems in the model transformation or in the specification of the target language.

Note that at Step 2, we only require to use graph transformation rules to specify model transformations in order to use the automatic program generation facilities of VIATRA. Our verification technique is, in fact, independent of the model transformation approach (only requires to use metamodeling and graph transformation for specifying modeling languages), therefore it is simultaneously applicable to relational model transformation approaches as well.

Prior to presenting the verification case study of a model transformation, we briefly discuss the pros and contras of metamodel level and model level verification of model transformations.

2.2 Metamodel vs. model level verification of model transformations

In theory, it would be advisable to *prove that a model transformation preserves certain semantic properties for any well-formed model instance*, but this typically requires the use of sophisticated theorem proving techniques and tools with a huge verification cost. The reason for that relies in the fact that proving properties even in a highly automated theorem prover require a high-level of user guidance since the invariants derived directly from metamodels should be typically manually strengthened in order to construct the proof. In this sense, the effort (cost and time) related to the verification of a transformation would exceed the efforts of design and implementation which is acceptable only for very specific (safety-critical) applications.

However, the overall aim of model transformations is to provide a precise and automated framework for transforming concrete applications (i.e., UML models). Therefore, in practice, *it is sufficient to prove the correctness of a model transformation for any specific but arbitrary source model*. Thanks to existing model checker tools and the transformation presented in [24], the entire verification process can be highly automated. In fact, the selection of a pair (p,q) of corresponding semantic properties is the only part in our framework that requires user interaction and expertise.

Even if the a verification of a specific model transformation is practically infeasible due to state space explosion caused by the complexity of the target application, model checkers can act as highly automated debugging aids for model transformations supposing that relatively simply source benchmark models are available as test sets.

As a conclusion, from an industrial perspective, a highly automated debugging aid for model transformations (as provided by our model checking based approach) is (at least) as valuable as a user guided excessive formal verification of a transformation.

3 Case Study: From UML Statecharts to Petri Nets

We present (an extract of) a complex model transformation case study from UML statecharts to Petri nets (denoted as SC2PN) in order to demonstrate the feasibility of our verification technique for model transformations. The SC2PN transformation was originally design and implemented as part of an industrial project where UML statecharts are projected into Petri nets in order to carry out various kinds of formal analysis (e.g., functional correctness [18], performance analysis [13]) on UML designs (i.e., to formally analyze UML models but not model transformations). Due to severe page limitations, we can only provide an overview of the verification case study, the reader is referred to [25] for a more detailed discussion.

3.1 Defining modeling languages by model transformation systems

Prior to reasoning about this model transformation, both the source and target modeling languages (UML statecharts and Petri nets) have to be defined precisely. For that purpose, in [26] we proposed to use a combination of metamodeling and graph transformation techniques: the *static structure* of language is described by a corresponding *metamodel* clearly separating static and dynamic concepts of the language, while the *dynamic operational semantics* is specified by *graph transformation*.

Graph transformation (see [20] for theoretical foundations) provides a rule-based manipulation of graphs, which is conceptually similar to the well-known Chomsky grammar rules but using graph patterns instead of textual ones. Formally, a **graph transformation rule** (see e.g. addTokenR in Fig. 3 for demonstration) is a triple $Rule = (Lhs, Neg, Rhs)$, where Lhs is the left-hand side graph, Rhs is the right-hand side graph, while Neg is (an optional) negative application condition (grey areas in figures). Informally, Lhs and Neg of a rule defines the *precondition* while Rhs defines the *postcondition* for a rule application.

The **application** of a rule to a **model (graph)** M (e.g., a UML model of the user) alters the model by replacing the pattern defined by Lhs with the pattern of the Rhs. This is performed by

1. *finding a match* of the Lhs pattern in model M;
2. *checking the negative application conditions* Neg which prohibits the presence of certain model elements;
3. *removing* a part of the model M that can be mapped to the Lhs pattern but not the Rhs pattern yielding an intermediate model IM;
4. *adding* new elements to the intermediate model IM which exist in the Rhs but cannot be mapped to the Lhs yielding the derived model M'.

In our framework, graph transformation rules serve as elementary operations while the entire operational semantics of a language or a model transformation is defined by a **model transformation system** [27], where the allowed transformation sequences are constrained by *control flow graph* (CFG) applying a transformation rule in a specific *rule application mode* at each node. A rule can be executed (i) parallelly for all matches as in case *forall* mode; (ii) on a (non-deterministically selected) single matching as in case of *try* mode; or (iii) as long as applicable (in *loop* mode).

UML statecharts as the source modeling language As the formalization of UML statecharts (abbreviated as SC) by using this technique and a model checking case study were discussed in [23, 24], we only concentrate on the precise handling of the target language (i.e., Petri nets) in this paper. We only introduce below a simple UML model as running example and assume the reader's familiarity with UML and metamodels.

Example 1 (Voting). The simple UML design of Fig. 2) models a voting process which requires a consensus (i.e., unique decision) from the participants.

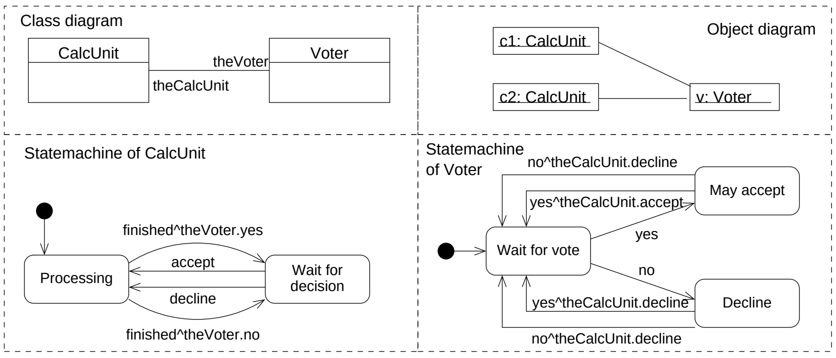


Fig. 2. UML model of a voter system

In the system, a specific task is carried out by multiple calculation units CalcUnit, and they send their local decision to the Voter in the form of a yes or no message. The voter may only accept the result of the calculation if all processing units voted for yes. After the final decision of the voter, all calculation units are notified by an accept or a decline message. In the concrete system, two calculation units are working on the desired task (see the object diagram in the upper right corner of Fig. 2), therefore the statechart of the voter is rather simplified in contrast to a parameterized case.

Petri nets as the target modeling language Petri nets (abbreviated as PN) are widely used means to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available tools. A precise metamodeling treatment of Petri nets was discussed in [26]. Now we briefly revisit the metamodel and the operational semantics of Petri nets in Fig. 3.

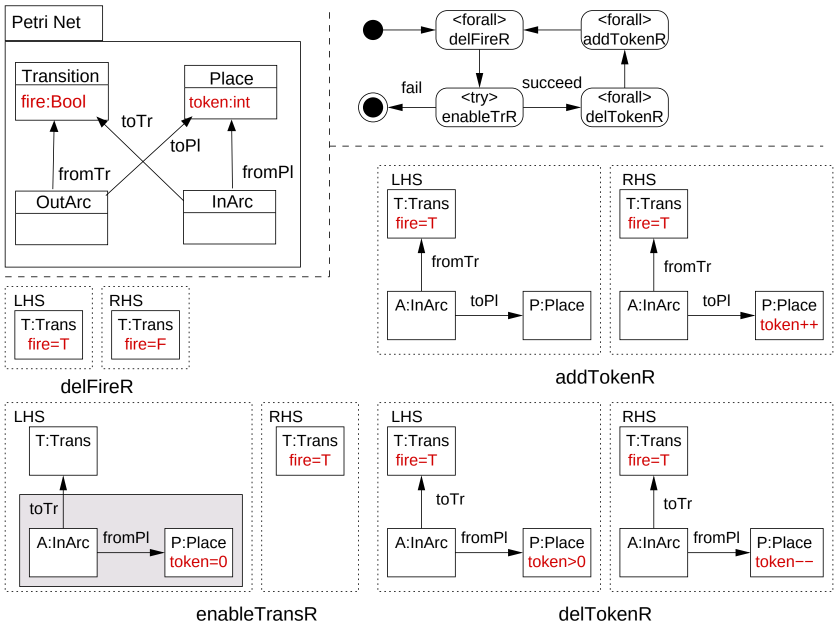


Fig. 3. Operational semantics of Petri nets by graph transformation

According to the metamodel (the Petri Net package in the upper left corner of Fig. 3), a simple Petri net consists of Places, Transitions, InArcs, and OutArcs as depicted by the corresponding classes. InArcs are leading from (incoming) places to transitions, and OutArcs are leading from transitions to (outgoing) places as shown by the associations. Additionally, each place contains an arbitrary (non-negative) number of tokens). Dynamic concepts, which can be manipulated by rules (i.e., attributes token, and fire) are printed in red.

The operational behavior of Petri net models are captured by the notion of *firing a transition* which is performed as follows.

1. First, fire attributes are set to false for each transition of the net by applying rule *delFireR* in *forall* mode.

2. A single enabled transition T (i.e., when all the places P with an incoming arc A to the transition contain at least one token token>0) is selected to be fired (by setting the fire attribute to true) when applying rule *enableTransR* in *try* mode.
3. When firing a transition, a token is removed (i.e., the counter token is decremented) from each incoming place by applying *delTokenR* in *forall* mode.
4. Then a token is added to each outgoing place of the firing transition (by incrementing the counter token) in a *forall* application of rule *addTokenR*.
5. When no transitions are enabled, the net is dead.

3.2 Defining the SC2PN model transformation

Modeling statecharts by Petri nets Each SC state is modeled with a respective place in the target PN model. A token in such a place denotes that the corresponding state is active, therefore, a single token is allowed on each level of the state hierarchy (forming token ring, or more formally, a *place invariant*). In addition, places are generated to model messages stored in event queues of a statemachine. However, the proper handling of event queues is out of the scope of the current paper, the reader is referred to [25].

Each SC step (i.e., a collection of SC transitions that can be fired in parallel) is projected into a PN transition. When such a transition is fired, (i) tokens are removed from source places (i.e., places generated for the source states of the step) and event queue places, and (ii) new tokens are generated for all the target places and receiver message queues. Therefore, input and output arcs of the transition should be generated in correspondence with this rule.

Example 2. In Fig. 4, we present a(n extract) of the Petri net equivalent of the voter's UML model (see Fig. 2). For improving legibility, only a single transition (leading from state may_accept to wait_for _vote and triggered by the yes event) is shown.

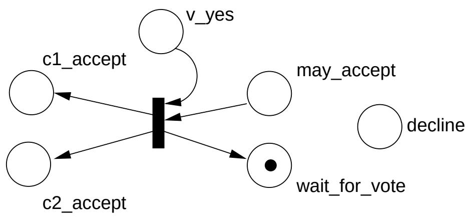


Fig. 4. The Petri net of the voter (extract)

The places of the voter subsystem are constituted of the states of the voter (such as wait_for_vote, may_accept, decline) and message queues for valid events (like yes). The initial state is marked by a token in wait_for_vote.

The depicted transition has two incoming arcs as well, one from its source state may_accept and one from the message queue of the triggering yes event. Meanwhile, this transition has multiple output places: one for the target state wait_for_vote, and one for each target event queue of the participants that receives the generated accept message.

Formalizing model transformations In [25], we formalize the SC2PN transformation (to handle a meaningful subset of UML statecharts) by model transformation systems consisting of more than 40 graph transformation rules. Feeding these high-level descriptions to VIATRA [5], (an XMI representation of) a transformation program is generated automatically, which would yield the target Petri net model (Fig. 4) as the output when supplying (the XMI representation of) the voter's UML model (Fig. 2) as the input.

Figure 5 gives a brief extract of transforming SC states into PN places. According to this pair of rules, each initial state (i.e., that is active initially) in the source SC model is transformed into a corresponding PN place containing a single token, while each non-initial state (i.e., that is passive initially) is projected into a PN place without a token.

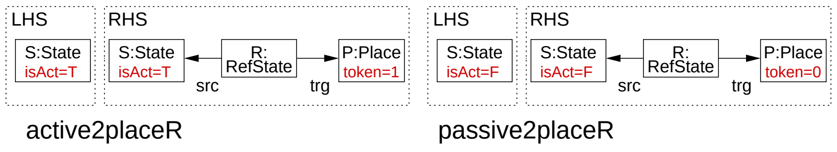


Fig. 5. Transforming SC states into PN places

It is worth noted that a model transformation rule in VIATRA is composed of elements of the source language (like State S in the rule), elements of the target language (like Place P), and reference elements (such as RefState R). Latter ones are also defined by a corresponding metamodel. Moreover, they provide bi-directional transformations for the *static parts* of the models, thus serving as a primary basis for back-annotating the results of a Petri net based analysis into the original UML design.

3.3 Verification of the SC2PN model transformation

For the SC2PN case study, Steps 1–3 in our verification framework have already been completed. Now, a transition system (TS) is generated automatically (according to [24]) for source and target models as an equivalent (model-level) representation of the operational semantics defined by graph transformation rules (on the meta-level).

Generating transition systems Transition systems (or state transition systems) are a common mathematical formalism that serves as the input specification of various model checker tools. They have certain commonalities with structured programming languages (like C or Pascal) as the system is evolving from a given *initial state* by executing non-deterministic if-then-else like *transitions* (or *guarded commands*) that manipulate *state variables*. In all practical cases, we must restrict the state variables to have finite domains, since model checkers typically traverse the entire state space of the system to decide whether a certain property is satisfied. For the current paper, we use the easy-to-read SAL [3] syntax for the concrete representation of transition systems.

Our generation technique (described in [24] also including feasibility studies from a verification point of view) enables model checking for graph transformation systems by automatically translating them into transitions systems. The main challenge in such a translation is two fold: (i) we have to "step down" automatically from the meta-level to the model-level when generating model-level transition systems from meta-level graph transformation systems, and (ii) a naive encoding of the graph representation of models would easily explode both the state space and the number of transitions in the transition system even for simple models. Therefore our technique applies the following sophisticated optimizations:

– Introducing state variables in TS only for dynamic concepts of a language.
– Including only dynamic parts of the initial model in the initial state of the TS.
– Collecting potential applications of a graph transformation rule by partially applying them on the static parts of the rule and generating a distinct transition (guarded command) for each of them that only contains dynamic parts as conditions in guards and assignments in actions.

In order to give an impression on the generated target transition system, we give below an extract from the SAL encoding of our Petri net model (of Fig. 4).

```
% Type declarations
placeID : TYPE = {wait_for_vote, may_accept, decline,
                  v_yes, c1_accept, c1_accept };
transID :  TYPE = {t, ...};
pn1 : MODULE =
BEGIN % declaring state variables
 GLOBAL token:   ARRAY placeID OF INTEGER
 GLOBAL fire:    ARRAY transID OF BOOLEAN
INITIALIZATION
 token[wait_for_vote] = 1; token[decline] = 0;
 token[may_accept] = 0; token[v_yes] = 0; ...
 fire[t] = FALSE; ...
TRANSITION
% generated for one potential matching of rule enableTransR
  fire[t] = FALSE AND
  NOT (token[wait_for_vote] = 0) AND
  NOT (token[v_yes] = 0) -->
    fire'[t] = TRUE; [] ...
END;
```

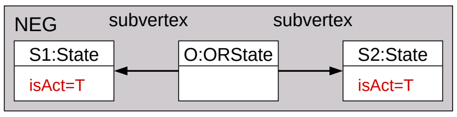
– The objects and variable domains are transformed into type (domain) declarations (see, e.g., the corresponding value for place decline in type placeID).
– State variable arrays are introduced only for attributes token and fire (the only dynamic concepts of Petri nets).
– Initialization is consistent with the initial marking of the Petri net (i.e., place wait_for_vote contains a token thus the corresponding variable token is initialized to 1).
– The guarded command generated from the potential application of rule *enableTransR* to the PN transition depicted Fig. 4 only checks the corresponding dynamic concepts (the fire attribute is false and there are tokens in both places wait_for_vote and v_yes) as the static preconditions of the rule are checked at compile time.

Formalizing the correctness property Now, a semantic criterion is defined for the verification process that should be preserved by the SC2PN model transformation. Note that the term "safety criterion" below refers to a class of temporal logic properties prohibiting the occurrence of an undesired situation (and not to the safety of the source UML design).

Definition 1 (Safety criterion for statecharts). *For all OR-states (non-concurrent composite states) in a UML statechart, only a single substate is allowed to be active at any time during execution.*

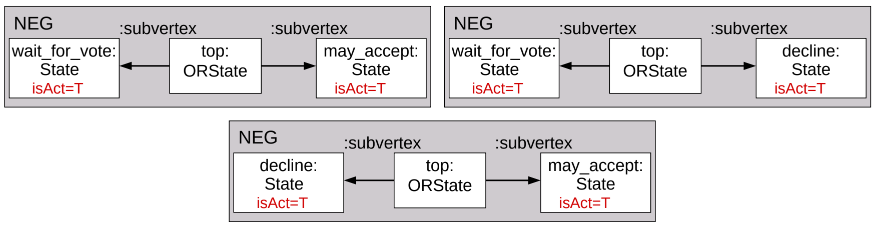
This informal requirement can be formalized by the following graphical invariant in the domain of UML statecharts (cf. Fig. 6 together with its equivalent logic formula). Informally speaking, it prohibits the simultaneous activeness of two distinct substates S1 and S2 of the same OR state C (i.e., non-concurrent composite state) .

Unfortunately, it is difficult to establish the same criterion on the meta level in the target language of Petri nets since the SC2PN transformation defines an abstraction in the sense that message queues of objects are also transformed into PN places (in addition to states). However, in order to model check a certain system, this meta-level correctness criterion can be re-introduced on the model level. Therefore, we first automatically instantiate (the static parts of) the criterion on the concrete SC model (as done during the transformation to transitions systems) to obtain the model level criterion of Fig. 7. Note that the different (model level) patterns denote conjunctions, therefore, none of the depicted situations are allowed to occur.



$$\nexists\, O : ORState, S1 : State, S2 : State : subvertex(A, S1) \wedge subvertex(A, S2) \wedge isAct(S1) \wedge isAct(S2) \wedge S1 \neq S2$$

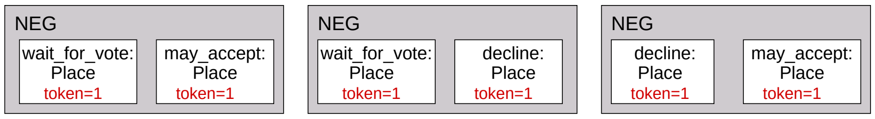
Fig. 6. A sample graphical safety criterion



$$\neg(subvertex(top, wait\_for\_vote) \wedge subvertex(top, may\_accept) \wedge isAct(wait\_for\_vote) \wedge isAct(may\_accept)) \wedge ...$$

Fig. 7. Model level safety criterion

Equivalent property in the target language This model level criterion is appropriate to be transformed into an equivalent criterion for the Petri net model. As the state hierarchy of statecharts is not structurally preserved in Petri nets (as Petri nets are flat) the equivalents of the OR states are not projected into Petri nets. Therefore, the corresponding property (shown in Fig. 8) contain only specific places having a token.



$$\neg(token(wait\_for\_vote) = 1 \wedge token(may\_accept) = 1) \wedge ...$$

Fig. 8. The Petri net equivalent of the model level safety criterion

At this point, we need to validate whether the equality ($= 1$) or inequality checks (≥ 1) are required in the property to be proved (i.e., what to do if there are multiple tokens in a single place). We may conclude that checking equality is also sufficient, however, checking the version with inequality definitely strengthens the property, therefore we can also decide to prove something stronger in the Petri net model.

Obviously, constructing the pair of properties to be proved for property preservation is non-trivial and requires a certain insight into the source and target languages and their transformation. Therefore the generation of a target property q from a source property p cannot always be automated.

Model checking the target model Given (i) a system model in the form of a transition system TS (with semantics defined as a Kripke structure), and (ii) a safety property ϕ, the *model checking problem* can be defined as to decide whether ϕ holds on all execution paths of the system (i.e., whether $TS \models \phi$).

Therefore, as the final step of our framework, the model checker is supplied with the transition system of the Petri net model and the textual representation of the property q. As the places derived from the states of the same OR-state form a place invariant (with a single token circulating around), the model checker easily verifies even the strengthened property.

As a conclusion for our case study, we may draw that the SC2PN model transformation preserved our sample correctness property for a specific source statechart model and its target Petri net equivalent. Additional correctness properties can be handled similarly. Unfortunately, for space considerations, we omitted the formal verification of property in the source SC model (Step 6), which could be performed identically to the handling of the target PN model.

4 Conclusions and Future Work

We presented a meta-level and highly automated technique to formally verify by model checking that a model transformation from an arbitrary well-formed model instance of a source modeling language into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrated the feasibility of our approach by verifying a semantic correctness property for a complex model transformation from UML statecharts to Petri nets.

Naturally, as based on model checking our technique has practical limitation imposed by the state explosion problem. Therefore, in the future, we aim to improve our automated encoding into transition systems to better exploit the built-in facilities of model checkers (like partial order reduction or symmetries) to allow the verification of larger scale model transformations. [1]

References

1. D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conf. on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 243–258. Springer, Dresden, Germany, 2002.
2. L. Baresi and M. Pezzè. On formalizing UML with High-Level Petri Nets. In G. Agha, F. De Cindio, and G. Rozenberg (eds.), *Concurrent Object-Oriented Programming and Petri Nets (Advances in Petri Nets)*, vol. 2001 of *LNCS*, pp. 271–300. Springer, 2001.
3. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway (ed.), *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pp. 187–196. 2000.
4. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.
5. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proc. ASE 2002: 17th IEEE Intern. Conf. on Automated Software Engineering*. Edinburgh, UK, 2002.
6. B. Demuth, H. Hussmann, and S. Obermaier. Experiments with XMI based transformations of software models. In *Workshop on Transformations in UML*. 2001.
7. H. Ehrig, R. Geisler, M. Grosse-Rhode, M. Klar, and S. Mann. On formal semantics and integration of object oriented modeling languages. In *EATCS*, vol. 70, pp. 77–81. The Formal Specification Column, 2000.
8. G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn (eds.), *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 272–286. Springer, 2001.
9. G. Engels, R. Heckel, J.-M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conf. on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 212–227. Springer, Dresden, Germany, 2002.
10. J. H. Hausmann, R. Heckel, and S. Sauer. Extended model relations with graphical consistency conditions. In *UML 2002 Workshop on Consistency Problems in UML-based Software Development*, pp. 61–74. Blekinge Institute of Technology, 2002. Research Report 2002:06.
11. J. H. Hausmann and S. Kent. Visualization of mappings ??? In *ACM ???*

[1] Remark to the reviewers: all previous papers of the authors are available at `http://www.inf.mit.bme.hu`.

12. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: First Intern. Conf. on Graph Transformation*, vol. 2505 of *LNCS*, pp. 161–176. Springer, Barcelona, Spain, 2002.
13. G. Huszerl and I. Majzik. Quantitative analysis of dependability critical systems based on UML statechart models. In *HASE 2000, Fifth IEEE International Symposium on High Assurance Systems Engineering*, pp. 83–92. 2000.
14. H. Juan de Lara, Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber (eds.), *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
15. J. M. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *Proc. VLFM'03: Intern. Conf. on Visual Languages and Formal Methods*. Submitted.
16. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, vol. 11(6):pp. 637–664, 1999.
17. D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Trans. on Software Engineering*, vol. 28(4):pp. 413–431, 2002.
18. A. Pataricza. Semi-decisions in the validation of dependable systems. In *Suppl. Proc. DSN 2001: The International IEEE Conference on Dependable Systems and Networks*, pp. 114–115. Göteborg, Sweden, 2001.
19. M. Peltier, J. Bézivina, and G. Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In J. Whittle et al. (eds.), *Workshop on Transformations in UML*, pp. 93–97. 2001.
20. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
21. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003: 6th Intern. Conf. on the Unified Modeling Language*. Submitted paper.
22. A. Schürr. Specification of graph translators with triple graph grammars. In . Tinhofer (ed.), *Proc. WG94: International Workshop on Graph-Theoretic Concepts in Computer Science*, no. 903 in LNCS, pp. 151–163. Springer, 1994.
23. D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: 1st Intern. Conf. on Graph Transformation*, vol. 2505 of *LNCS*, pp. 378–392. Springer, Barcelona, Spain, 2002.
24. D. Varró. Towards automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 2003. Submitted to the Special Issue on Graph Transformation and Visual Modelling Techniques.
25. D. Varró. *VIATRA: Visual Automated Model Transformations*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003. Submitted.
26. D. Varró and A. Pataricza. Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conf. on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 18–33. Springer, Dresden, Germany, 2002.
27. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.
28. J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth Intern. Conf. on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 227–242. Springer, Dresden, Germany, 2002.