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. When designing safety critical applications in UML, the system models are frequently projected into various mathematical domains (such as Petri nets, transition systems, process algebras, etc.) to carry out a formal analysis of the system under design by *automatic model transformations*. 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 model-level, modeling language independent 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, UML statecharts, Petri nets.

1 Introduction

For most computer controlled systems, especially dependable, real-time systems for critical applications, an effective design process requires an early validation of the concepts and architectural choices, without wasting time and resources to assess whether the system fulfills its requirements or needs some re-design.

The Unified Modeling Language (UML) together with domain specific profiles (e.g., the UML Profile for Schedulability, Performance and Time [16]) provides a standard and easy-to-understand visual way to capture both the requirements and the system model.

However, a standard modeling language does not alone guarantee the correctness of the design. In order to increase the level of confidence that can be put on a system mathematical tools (based on formal methods like Petri nets, dataflow networks, transition systems, process algebras, etc.) are used to assess the most important system parameters

* 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)

(such as functional correctness, timeliness, performability or dependability). Unfortunately, sophisticated verification tools (such as the SPIN model [11] checker) require a thorough knowledge of the underlying mathematics, and therefore special skills are needed for dependable IT system designers.

In order to bridge the huge abstraction gap, many approaches (e.g., [4, 7, 13, 25]) to automatically transform high-level UML based system models into low-level mathematical models, and then back-annotate the results of the formal analysis into the original UML model of the system in order to hide the underlying mathematics.

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) 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 is typically based upon a metamodel with OCL constraints [1, 15].
- *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 (a) graph transformation [5–8, 25], (b) triple graph grammars [20] or (c) term rewriting rules [26].

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 may become a quality bottleneck of a transformation based verification and validation framework (such as [4]). 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, 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 [25] by planner algorithms, and in [9] by graph transformation. Recently in [14], sufficient conditions were set up that guarantee the termination and uniqueness of transformations based upon the static analysis technique of critical pair analysis [10]. However, no approaches exist to reason about the semantic correctness of arbitrary model transformations, when transformation specific properties are aimed to be verified.

Our contribution In this paper, we present a model-level, modeling language independent and highly automated framework (in Sec. 2) to formally verify by model checking that a model transformation (specified by metamodeling and graph transformation techniques) 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 (in Sec. 3) on verifying a semantic property of a complex model transformation from UML statecharts to Petri nets.

The main benefit of our approach (in contrast to related solutions such as [8]) is that it can be adapted to arbitrary modeling languages taken from both software engineering and mathematical domains on a very high level of abstraction. More specifically, the transformation designers use the same visual notation (based on metamodeling and graph transformation) to capture the semantics of modeling languages and model transformations between them. Then our tools automatically (i) carry out the transformation from the source UML model into the target mathematical domain, and generate (ii) a model checking description to verify the correctness of the model transformation between the source and target model.

2 Automated Formal Verification of Model Transformations

We present an automated technique to formally verify (based on the model checking approach of [22]) 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, in our case).

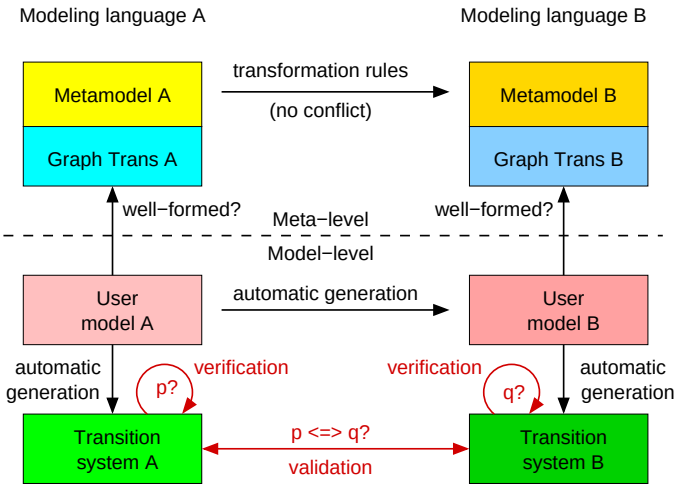


Fig. 1. Model level formal verification of transformations

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. We demonstrated in, for instance, [21,24] that many (we believe that all) languages in a realization of the MDA may have a semantics defined in this visual way, which is closely related to the UML philosophy.
2. **Specification of model transformations**. The A2B model transformation should be also specified by a set of (non-conflicting) graph transformation rules. The practical feasibility of such a solution has been demonstrated in many papers, see, e.g., [23] for an overview.
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). The correctness of this automated generation step is proved in [23].

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 provenly correct encoding presented in [22] (and with a tool support reported in [19]).
5. **Select a semantic correctness property**. We select one semantic property p (at a time) 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 [9].
6. **Model check the source model**. Transition system A is model-checked automatically (by existing model checker tools like SPIN [11] or SAL [3]) 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 into property q, domain experts should validate that property q is really the target equivalent of property p or a strengthened variant. Unfortunately, this validation step typically requires human expertise and might not be fully automated.
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.

Naturally, the correctness of a model transformation can only be deduced if the transformation preserves *every* semantic correctness property used in the analysis. Obviously, it requires several runs of the model checker, which can be time-consuming. Therefore, in [22], we assessed the expected run-time performance of our model checking based approach on a verification benchmark. In [2], the same technique was applied on architectural styles to check reachability properties. Both case studies demonstrated that our technique is applicable to non-trivial examples (of medium-size).

Furthermore, it is worth noting that the time related to the model transformation step, or to the automated generation of transition systems is still only a few percentage of the entire verification process in case of non-trivial models.

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

2.2 Meta-level vs. model level verification of model transformations

In theory, it would be advisable to *prove that a model transformation preserves certain predefined 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 lies 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 and critical applications.

However, the overall aim of model transformations is to provide a precise and automated framework for transforming the models of concrete applications (i.e., UML models). Therefore, in practice, *it is sufficient to prove the correctness of the model transformation from the source UML model of the system under design* against a set of properties defined by transformation engineers (while it is typically out of scope to demonstrate that the model transformation is correct for any source model). Thanks to existing model checker tools and the transformation presented in [22], such a model-level 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 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.

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 entire SC2PN transformation was originally designed and implemented as part of a Hungarian research project (IKTA 065/2000 – A framework for the modeling and analysis of dependable and safety critical systems) carried out in cooperation with industrial partners. Here UML statecharts are projected into Petri nets by this transformation in order to carry out (various kinds of) formal analysis such as functional correctness based on semi-decision methods of Petri nets [17].

The primary aim of the project was to formally verify UML models, but we also carried out the verification of the model transformation itself. Due to severe page limitations, we can only provide an overview of the verification case study, the reader is referred to [23] 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 [24] we proposed to use a combination of metamodeling and graph transformation techniques: the *static structure* of a 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 [18] 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) 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 define the *precondition* while Rhs defines the *post-condition* 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 (i) *finding a match* of the Lhs pattern in model M; (ii) *checking the negative application conditions* Neg which prohibits the presence of certain model elements; (iii) *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; (iv) *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** [25], where the allowed transformation sequences are constrained by a *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 [21, 22], 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.

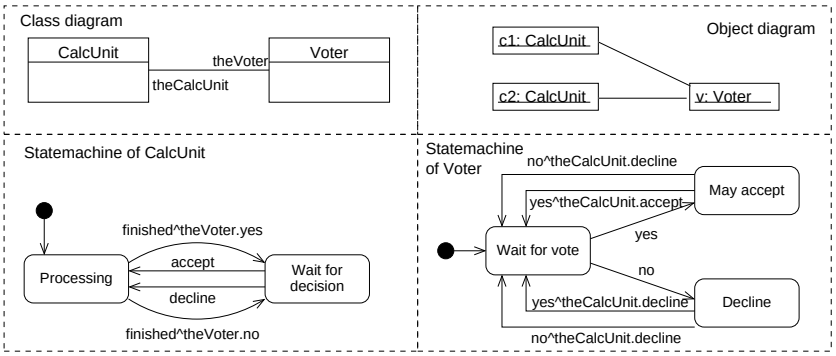


Fig. 2. UML model of a voter system

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 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 [24]. Now we briefly revisit the metamodel and the operational semantics of Petri nets in Fig. 3.

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.

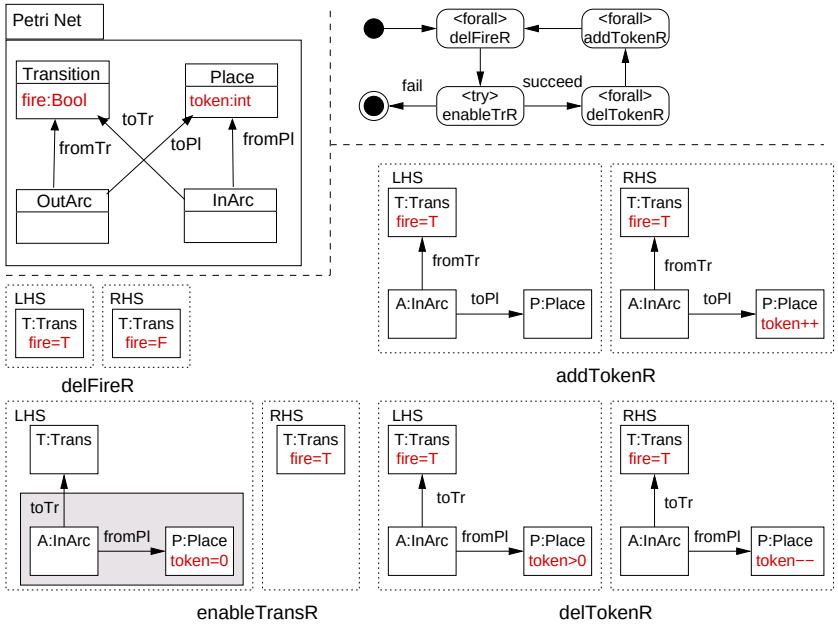


Fig. 3. Operational semantics of Petri nets by graph transformation

1. First, the fire attribute is 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 marks the corresponding state as active, therefore, a single token is allowed on each level of the state hierarchy (forming a 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 [23].

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.

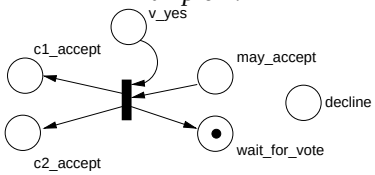


Fig. 4. The Petri net of the voter

In Fig. 4, we present an 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.

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 [23], 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.

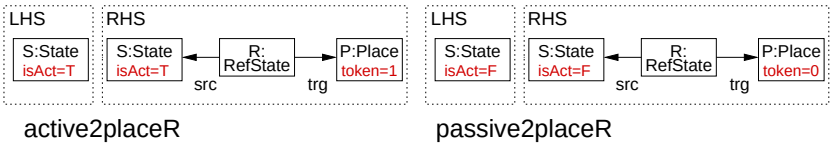


Fig. 5. Transforming SC states into PN places

It is worth noting 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). The latter ones are also defined by a corresponding metamodel. Moreover, they provide bi-directional transfor-

mations for the *static parts* of the models, thus serving as a 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 [22]) 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 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 syntax for the concrete representation of transition systems.

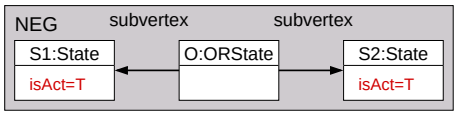
Our generation technique (described in [22] also including feasibility studies from a verification point of view) enables model checking for graph transformation systems by automatically translating them into transitions systems. The main challenge in such a translation is two fold: (i) we have to "step 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 the target transition system only for dynamic concepts of a language.
- Including only dynamic parts of the initial model in the initial state of the transition system.
- 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.

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.*

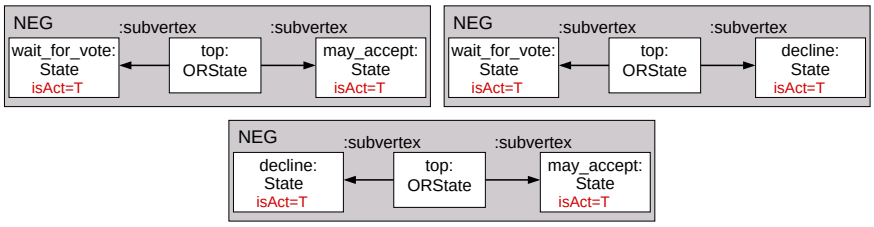
This informal requirement can be formalized by the following graphical invariant in the domain of UML statecharts (cf. Fig. 6 together with its equivalent logic formula). Informally speaking, it prohibits the simultaneous activeness of two distinct substates S1 and S2 of the same OR-state C (i.e., non-concurrent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



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

Fig. 6. A sample graphical safety criterion

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.



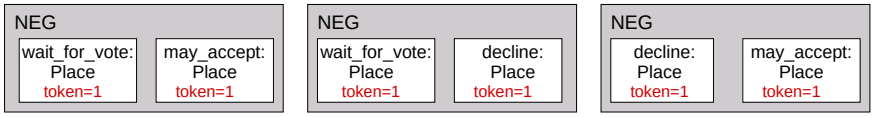
$$\neg(subvertex(top, wait\_for\_vote) \wedge subvertex(top, may\_accept) \wedge isAct(wait\_for\_vote) \wedge isAct(may\_accept)) \wedge ...$$

Fig. 7. Model level safety criterion

Note that our approach is not at all limited to verify only safety criteria. Further verification case studies (e.g., in [2, 22]) also covered reachability and liveness properties or deadlock freedom.

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.

At this point, we need to validate whether the equality (= 1) or inequality checks ($\geq$ 1) are required in the property to be proved (i.e., what to do if there are multiple tokens



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

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

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 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, 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 model-level, modeling language independent and highly automated technique to formally verify by model checking that a model transformation from a specific (but arbitrarily chosen) 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.

Further research should also aim at automating the transformation of semantic correctness properties. We think that our model transformation technique can be extended to handle this case as well. As a result, the same specification technique would be used for all transformations in our verification framework.

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 International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 243–258. Springer-Verlag, Dresden, Germany, 2002.
2. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles. In *Proc ESEC 2003: European Software Engineering Conference*. Helsinki, Finland. In press.
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 International Conference on Automated Software Engineering*, pp. 267–270. IEEE Press, Edinburgh, UK, 2002.
6. J. de Lara and H. 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.
7. 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.
8. 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 International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 212–227. Springer, Dresden, Germany, 2002.
9. 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.
10. 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 International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 161–176. Springer, Barcelona, Spain, 2002.
11. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23(5):pp. 279–295, 1997.
12. 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.

13. J.-M. Jézéquel, W.-M. Ho, A. L. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In R. J. Hall and E. Tyugu (eds.), *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
14. J. M. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *Proc. VLFM'03: International Conference on Visual Languages and Formal Methods*. Submitted.
15. D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Transactions on Software Engineering*, vol. 28(4):pp. 413–431, 2002.
16. Object Management Group. *UML Profile for Schedulability, Performance and Time*. `http://www.omg.org`.
17. 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.
18. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
19. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*. Accepted paper.
20. 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.
21. 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 International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 378–392. Springer-Verlag, Barcelona, Spain, 2002.
22. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 2003. Accepted to the Special Issue on Graph Transformation and Visual Modelling Techniques.
23. D. Varró. *Automated Model Transformations for the Verification and Validation of IT Systems*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003. Submitted.
24. D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modelling*, 2003 (1):pp. 1–24.
25. 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.
26. J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 227–242. Springer-Verlag, Dresden, Germany, 2002.