Automated Formal Verification of Model Transformations Using the Invariants Mechanism

Boris Ulitin(✉), Eduard Babkin, Tatiana Babkina, and Arsenii Vizgunov

National Research University Higher School of Economics, Nizhny Novgorod, Russia
{bulitin, eababkin, tbabkina, anvizgunov}@hse.ru

Abstract. The article is devoted to the problem of automated formal verification of modeling artifacts during engineering of digital transformations. Automation significantly increases the quality of model transformations since many manual errors are eliminated. However, the formal checking the correctness of such automation remains an open question. One more problem is the dependence of the procedure for checking the correctness of transformations on the modeling languages of the source and target models. In the article we represent the solution, based on the formalism of invariant checking, that allows modelers to formally test the correctness of model transformation regardless of a modeling language.

Keywords: Model transformation · Graph transformation · Model checking · Formal verification · Invariants

1 Introduction

Engineering of digital transformations became a highly demanded and challenging topic both for practitioners and academy. New flexible forms of inter-organizational communication leverage design of dynamically bonded organizational bodies. Fractal organizations [1] or autonomous distributed organizations (DAO) [2] give bright examples of such new organizational forms.

Apart from economic and social benefits such new kinds of organizations draw attention to new problems of decision support [2]. In the context of fully digitalized information landscape importance of resolving the issue of semantic interoperability during decision procedures cannot be overestimated.

Typical procedures of situation analysis and decision making involve multiple kinds of models which mimic different aspects of an enterprise. For better sensemaking and comprehension of the situation multiple model-to-model (M2M) transformations are frequently applied. Resent results show that even in the context of a traditional enterprise available formal methods of model transformations still need improvements. For example, works [5, 7, 10] demonstrate the insufficient level of formal verification during model transformations.

That article aims at proposing a new approach to automated formal verification of model transformations, which includes a formal model and corresponding algorithms for verification. Application of our proposals should assist dynamically composed fractal organizations in the process of semantic integration and decision support. In comparison with other known results our approach brings several new contributions. At first, that proposed approach is automated and language-independent. Since we use the graph-oriented general approach and the corresponding transformation, it is possible to apply the algorithm of model transformation verification to any modeling (in general case, domain-specific) language. The second advantage is that proposed approach implements the mechanisms of invariants in the process of verification of model transformations, thereby formalizing it. Finally, the verification procedure supports the verification of direct, as well as bidirectional model transformations.

In the scope of our research we evaluated the models and methods proposed in the context of software engineering models transformations. The results of evaluation confirmed by the application of the proposed approach to the transformation from UML Statechart into Petri net models.

This article describes our approach and presents results as follows. In Sect. 2 we give main aspects of M2M transformations and define criteria for its verification. Section 3 contains the algorithm, applied for the automated verification of model transformation correctness in terms of the approach proposed. Section 4 is devoted to the application of proposed approach to the case of Statechart and Petri net models. We conclude the article with the analysis of the proposed approach and further research steps.

2 Background

2.1 Definition and Approaches to M2M Transformations

Before the analyzing the process of M2M transformations and its validation, the definition of them should be formulated. From the formal point of view, the basic concept of transformation definition is a production rule which looks like $p : L \to R$, where p is a *rule name*, L is a *left-hand side* of the rule, also called the pattern, and R is a *right-hand side* of the rule, which is called the replacement model (or the target model). Rules are applied to the starting model named the source model. From this point of view, the model transformation is a sequenced applying to the starting source model M_0 of finite set of rules $P = (p_1, p_2 \ldots p_n) : M_0 \xrightarrow{p1} M_1 \xrightarrow{p2} \ldots \xrightarrow{pn} M_n$.

Transformations can be classified as *horizontal* or *vertical* according to the direction. The horizontal transformation is the conversion, in which the source and target models belong to the same hierarchy level, for example, a conversion of model description from one notation to another. The vertical transformation converts the models which belong to different hierarchy levels, for example, mapping objects of the metamodel to domain model objects. In what follows, the authors concentrate on the horizontal transformations, implemented between different modeling languages.

Depending on the language on which the source and the target models are described, horizontal transformations can be divided into two types: *endogenous* and *exogenous*. An endogenous transformation is the transformation of the models, which are described on the same modeling language. An exogenous transformation is the transformation of models, which are described on different modeling languages [3].

Such separation is important since there are two main approaches to M2M transformation: *operational* and *declarative*. The former is based on rules or instructions that explicitly state how and when creating the elements of the target model from elements of the source one occurs. Such a specification mainly combines metamodeling with graph transformation [4, 5], triple graph grammars [6] or term rewriting rules [7].

Instead, in declarative approaches, some kind of visual or textual patterns describing the relations between the source and the target models are provided, from which operational mechanisms are derived e.g. to perform forward and backward transformations. These declarative patterns are complemented with additional information to express relations between attributes in source and target elements, as well as to constrain when a certain relation should hold. The Object Constraint Language (OCL) is frequently used for this purpose [8].

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.

However, automation alone cannot protect against conceptual flaws implanted into the specification of a complicated model transformation. Consequently, a formal analysis carried out on the source and the target models 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 [9]). Therefore, prior to analyzing the target model, we have to prove that the model transformation itself is free of conceptual errors.

Unfortunately, it is hard to establish a single notion of correctness for model transformations. In what follows we analyze existing criteria of M2M transformations correctness and choose the best ones for automated M2M transformation process.

2.2 Correctness Criteria of Model Transformations

When we talk about correctness of the model transformation, the properties are introduced, addressed by the verification of the model transformation. There are several works [5, 10] which introduce the problem of verification by defining the set of properties to be addressed. However, the contents of these proposals are mostly included and generalized in [11]. Based on this work, the following categories of properties can be identified: language-related and transformation-related properties.

Language-related properties refers to the computational nature of transformations and target properties of transformation languages. As introduced in [11], a transformation specification conforms to a transformation language which can possess properties on its own. In this context there are four properties of interest.

- ***Termination*** property, which guarantees the existence of a target model, i.e. that the transformation execution finishes for any well-formed transformation specification;

- ***Determinism (Confluence)*** property, which ensures uniqueness of the target model for a given source model and transformation specification;
- ***Typing*** property, which ensures the well-formedness of the transformation specification in terms of the transformation language chosen;
- ***Preservation of Execution Semantics*** (***Dynamic consistency***) property, which states that the transformation execution must behave as expected according to the definition of the transformation language semantics.

As you can see, these properties are mostly connected with the possibility of the transformation itself. From this point of view, they form the semantic requirements for the transformation correctness. In our case, during the analysis of the correctness of automatically generated transformations, the most interesting are termination, confluence and dynamic consistency properties. These criteria allow us to check not only the correctness of the constructed models and transformations between them, but also to make the procedure for such verification independent of the specific model description languages.

Along with this, syntactic criteria can also be entered. These criteria are represented by transformation-related properties, which can be separated into two categories.

- ***Syntactic correctness***, which includes ***Conformance and Model Typing*** and ***N-Ary Transformations Properties***, to guarantee that the generated model is a syntactically well–formed instance of the target language.
- ***Syntactic completeness***, to completely cover the source language by transformation rules, i.e., to prove that for each construct in the source language there is a corresponding element in the target model.

These properties are responsible for the correctness of the structures used to describe the source and the target models, as well as the transformations between them. Syntactic correctness and completeness were tackled in [5] by planner algorithms, and in [12] by graph transformations. Recently in [4], sufficient conditions were set up that guarantee the termination and uniqueness of transformations based upon the static analysis technique of critical pair analysis.

However, no approaches exist to reason about the semantic correctness of arbitrary model transformations, when transformation specific properties are aimed to be verified. In what follows, we describe the unified and highly automated approach, allowing developers 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.

In contrast to related solutions (such as [4]) this approach can be adapted to arbitrary modeling languages taken from any enterprise engineering (and/or even mathematical) domains on a very high level of abstraction. As a result, the same visual notation (based on metamodeling and graph transformation) is used to capture the semantics of modeling languages and model transformations between them. Finally, the approach can be automated, during providing the transformation from the source into the target mathematical domain and subsequent generation of the model checking description to verify the correctness of the model transformation.

3 Proposed Approach

After the definition of different approaches to organize the M2M transformations and choice of the criteria of such transformations correctness, an automated approach can be described to formally verify the model transformation correctness of a specific source model into its target equivalent with respect to semantic properties.

In order to organize the automated formal verification of the M2M transformations, we have to describe the source and target models in a well-formed, strict manner. Such requirement guarantees the possibility to formalize the checking properties of semantic correctness of the transformation and generate a transition system in an automatic way.

Assuming a conversion between the two modeling languages *A* and *B*, to organize the automated formal verification of the M2M transformations the following algorithm have to be applied. This algorithm is a generalized and supplemented version of the algorithms described in [8] and [12].

1. **Specification of modeling languages.** First of all, as it was mentioned, both modeling languages (*A* and *B*) should be defined precisely using metamodeling and graph transformation techniques. For this, the approach described in [13] can be applied.
2. **Specification of model transformations.** The M2M transformation from *A* into *B* can also be specified by a set of graph transformation rules. It logically follows from the opportunity to represent any model in graph-oriented manner as a pair (E, R) of *E* entities and *R* relations between them. The rationale for the effectiveness of such a decision is given in [13] and [14]. It is important to note, that such transition to graph transformations is used to apply the automatic program generation facilities without affecting the independence of the verification technique in general.
3. **Automated model generation.** During this step, for a certain (but an arbitrary) well-formed model instance of the source language A is derived the corresponding target model by automatically generated transformation programs (for example, VIATRA). The correctness of this automated generation step is proved in [10].
4. **Generating transition systems.** At this phase, 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 [10]. In our case we implement the transition system also as a set of OCL invariants, which state the correspondence between elements of the source and target models. Such definition of the transition system allows us to unify and automate the procedure of checking correctness of the model transformations and make it independent from the modeling language used. In more details the process of deriving OCL invariants from models' transformations is described in Sect. 3.2.
5. **Select a semantic correctness property.** After the definition of the transition system we should formulate the property of its correctness. For this goal one semantic property *p* (at a time) can be chosen in the source language *A* which is structurally expressible as a graphical pattern composed of the elements of the source metamodel. More details on using graphical patterns to capture static well-formedness properties can be found in [12].

6. **Model check the source model.** Since the property p of semantic correctness over the model A is chosen, we should check its appropriateness to the transition system derived. In this case, the discrepancy may be expressed in the following forms:
   a. Some inconsistencies detected in the source model itself (a verification problem occurred),
   b. Some informal requirements are not captured properly by property p (a validation problem occurred),
   c. The formal semantics of the source language is inappropriate as a counter example is found which should hold according to our informal expectations (also validation problem).
7. **Transform and validate the property.** Using the transformations system, the property p in the source language A is transformed into the property q in the target language B (manually, or using the same transformation program). Unfortunately, this vital validation step might not be fully automated, because it requires the participation of an expert, who proves that the property q is really the target equivalent of the property p or a strengthened variant.
8. **Model check the target model.** Finally, the transition system B is model-checked against the property q. The result of checking may be one of the following alternatives:
   a. 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.
   b. Otherwise, the property p is not preserved by the model transformation and debugging can be initiated based upon the error traces 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.

It is fair to note that the correctness of a model transformation can only be deduced if the transformation preserves every semantic correctness property used during the analysis. As a result, the procedure of verification of M2M transformation can be time-consuming. However, we leave the performance analysis beyond the scope of this study, addressing the works [5] and [12].

3.1 Defining Models Transformations Through Graphs and Invariants

Since the modeling language may represent an example of a domain-specific language (DSL), we assume to use a graph-oriented approach to the organization of model transformations, like described in the study [13]. Such approach is reasonable, since any model can be generalized as a set of interconnected entities in object-oriented manner with subsequent application of graph-transformations for its development or evolution.

Graph transformation (see [6] 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. From the formal point of view, any graph transformation rule is represented be a triple $Rule = (L, Neg, R)$, where L is the left-hand side graph, R is the right-hand side graph, *Neg* is (an optional) negative application condition.

In these conditions, the application of a rule to a model (graph) M results in replacing the pattern defined by L with the pattern of the R. From an algorithmic point of view, this means that we find a *match* of the L pattern in model M, check the negative application condition *Neg*, remove a part of model M, mapped to the pattern L and finally add new elements to the intermediate model *IM*, which exists in the R but cannot be mapped to the L yielding the derived model M.

Such mechanism is very close to the mechanism of finding inductive invariants, which describe the connection between components of two (or more) sets of objects and are denoted with inv_τ. For this type of invariants two types of specifications are defined, *next* and *inv*:

$$next_\tau.(p,q).F \triangleq [p \Rightarrow \mathcal{W}.F.q] \wedge inv_\tau.p.F \triangleq next_\tau.(p,p).F \wedge [\mathcal{J}.F \Rightarrow p]$$

Informally, $next_\tau.(p,q)$ means that whenever a transition is fired from a state that satisfies p, the resulting state satisfies q. Similarly, $inv_\tau.p$ specifies that p is true in any initial state and is preserved by every atomic transition. Therefore, by induction, p is true in every state. It should be noted that, since $[\mathcal{W}.F.q \Rightarrow q]$ because of possible stuttering $next_\tau.(p,q).F \Rightarrow [p \Rightarrow q]$.

According to the above definition, inductive invariant means, that there is a strong correspondence between elements of two sets of objects, which are connected during some relation (transformation). Such definition is very close to the relational approach for model transformation definition, when relationship between objects (and links) of the source and target language are declared. That results in the idea, that inductive invariant can be an effective mechanism for the definition of such model transformations and for the validation of the possibility of obtaining one model by transforming another.

In this case, 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, 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. From this point of view, the transition system consists of the operational invariants and can be associated with a subset $\mathcal{O}.F$ of $(\Sigma, \mathrm{F})^{w}$ of (in)finite sequences of states defined as follows.

An infinite computation $\sigma = \langle \sigma_0, \sigma_1, \ldots, \sigma_n, \ldots \rangle$ belongs to the set $\mathcal{O}.F$ if and only if

$$\begin{cases} \mathcal{J}.F.\sigma_0 \\ \forall i \in \mathbb{N} : \mathcal{W}.F.\{\sigma_{i+1}\}.\sigma_i \end{cases}$$

where $\{\sigma_{i+1}\}$ is the state predicate that evaluates to *true* for the state σ_{i+1} and to *false* for any other state.

Informally, $\mathcal{O}$ consists of those sequences of states that begin with an initial state that satisfies $\mathcal{J}$ (*Neg* negative application condition in our case) and in which each state has a successor in accordance with the transition $\mathcal{W}$ (*Rule* transformation rule in our case). The set $\mathcal{O}$ is nonempty because $\mathcal{J}$ is satisfiable and $\mathcal{W}$ includes stuttering steps.

Once the computations of a transition system are built, *next* and *inv* specifications are defined as expected:

$$next_{\mathcal{O}}.(p,q).F \triangleq \forall\sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p, \sigma_i \Rightarrow q, \sigma_{i+1}$$
$$inv_{\mathcal{O}}.p.F \triangleq \forall\sigma \in \mathcal{O}.F : \forall i \in \mathbb{N} : p, \sigma_i$$

Informally, $next_{\mathcal{O}}.(p,q)$ means that, in any computation of the system, any state that satisfies p is immediately followed by a state that satisfies q. Although computations include stuttering steps, $next_{\mathcal{O}}.(p,q)$ does *not* imply that $[p \Rightarrow q]$. In the same way, $inv_{\mathcal{O}}.p$ means that any state of any computation of a system satisfies p. Naturally, $next_{\mathcal{O}}$ and $inv_{\mathcal{O}}$ are related in a way similar to the relationship between $next_{\tau}$ and inv_{τ}, namely: $inv_{\mathcal{O}}.p.F \equiv next_{\mathcal{O}}.(p,q).F \wedge [\mathcal{J}.F \Rightarrow p]$.

According to these principles, we can conclude, that validation of the model transformation correctness can be fully described through invariant mechanisms. Such definition can allow us to automate the process of formal validation of the model transformation, reducing it to verifying the presence of invariants of both types among defined model (graph) transformations.

3.2 Deriving OCL Invariants from QVT Transformations

Since we describe the model transformation using the graph-oriented approach (see [13] for more details) in QVT transformation language, the procedure to derive the OCL invariants need be implemented. Using the principals of inductive invariants, we need to describe the correspondence between different components of the source and the target models.

This is fully consistent with the concept of the QVT transformation language. In this language, a bidirectional transformation consists of a set of relations between two models. There are two types of relations: top-level and non-toplevel. The execution of a transformation requires that all its top-level relations hold, whereas non-top-level ones only need to hold when invoked directly or transitively from another relation [15].

Each relation defines two domain patterns, one for each model, and a pair of optional when and where OCL predicates. These optional predicates define the link with other relations in the transformation: the when clause indicates the constraints under which the relation needs to hold and the where clause provides additional conditions, apart from the ones expressed by the relation itself, that must be satisfied by all model elements in the relation [15].

Among all nodes in a domain pattern, one is marked as a root element. Definition of root nodes is purely for the sake of clarity, that does not affect the semantics of the matching process. When referring to other relations in when or where clauses, parameters can be specified, and thus it is possible to pass bound variables from one relation to another. Note that the bound objects received as parameters are necessary preconditions to enforce the pattern. According to these principles, when we talk about deriving the transformation system from the QVT transformation, we need to identify the invariants of both levels, Top-relation as well as Non-top relation to provide the whole consistence between the source and the target models. In what follows, we use OCL constraints to define the specific invariants of both types.

Definition 1: Let p be a top-relation with domain patterns $S = \{root_s, s_1, \ldots, s_n\}$, and $T = \{root_t, t_1, \ldots, t_m\}$, and $T_{when} \subseteq T$ be the set of elements of T referenced in p's 'when' section. Then, the following top-relation invariant takes place for the $S \rightarrow T$:

$$\begin{gathered}
\textbf{\textit{context}}\ type(root_s)\ \textbf{\textit{inv}}\ p: \\
\left(\begin{array}{l} type(x_i) :: allInstances() \rightarrow forAll(x_i| \\ type(x_j) :: allInstances() \rightarrow forAll(x_j|\ldots \end{array}\right)\forall x_k \in (S\backslash\{root_s\}) \cup T_{when} \\
\textbf{\textit{if}}\ self.p - enabled(x_i, x_j, \ldots)\ \textbf{\textit{then}} \\
\left(\begin{array}{l} type(x_u) :: allInstances() \rightarrow exists(x_u| \\ type(x_v) :: allInstances() \rightarrow exists(x_v|\ldots \end{array}\right)\forall x_w \in T\backslash T_{when} \\
self.p - mapping(x_i, x_j, \ldots, x_u, x_v, \ldots)\ldots))\ \textbf{\textit{endif}}\ldots)) \\
\textbf{\textit{context}}\ type(root_s) :: p - enabled(x_i : type(x_i), \ldots) \\
\textbf{\textit{body}} : when\ and\ enabling\ conditions \\
\textbf{\textit{context}}\ type(root_s) :: p - mapping(x_i : type(x_i), \ldots) \\
\textbf{\textit{body}} : where\ and\ mapping\ conditions
\end{gathered}$$

Definition 2: Let p be a non-top relation with domain patterns $S = \{root_s, s_1, \ldots, s_n\}$, and $T = \{root_t, t_1, \ldots, t_m\}$, and $T_{when} \subseteq T$ be the set of elements of T referenced in p's 'when' section, and $P = \{a_1, \ldots, a_k\} \subseteq S \cup T$ the set of elements passed as parameters in the call to p from other relations.

Then, the following non-top relation invariant (a boolean operation) takes place for the $S \rightarrow T$:

$$\begin{gathered}
\textbf{\textit{context}}\ type(root_s) :: p(a_1 : type(a_1), \ldots, a_k : type(a_k)) \\
\left(\begin{array}{l} type(x_i) :: allInstances() \rightarrow forAll(x_i| \\ type(x_j) :: allInstances() \rightarrow forAll(x_j|\ldots \end{array}\right)\forall x_k \in (S\backslash\{root_s\}\backslash P) \cup T_{when} \\
\textbf{\textit{if}}\ self.p - enabled(x_i, x_j, \ldots)\ \textbf{\textit{then}} \\
\left(\begin{array}{l} type(x_u) :: allInstances() \rightarrow exists(x_u| \\ type(x_v) :: allInstances() \rightarrow exists(x_v|\ldots \end{array}\right)\forall x_w \in T\backslash T_{when}\backslash P \\
self.p - mapping(x_i, x_j, \ldots, a_1, \ldots, a_k x_u, x_v, \ldots)\ldots))\ \textbf{\textit{endif}}\ldots))
\end{gathered}$$

After such extraction of the invariants from the QVT transformations, the correctness of the model transformation can be applied for solving two problems: (1) verification of correctness properties of transformations, that is, finding defects in them and (2) validation of transformations, that is, identifying transformations whose definition does not match the designer intent.

With application of OCL invariants both problems can be solved using existing OCL verification and validation tools for the analysis of model transformations. With these inputs, verification tools provide means to automatically check the consistency of the transformation model without user intervention. Checking consistency allows the verification of the executability of the transformation and the use of all validation scenarios. Other properties checked automatically by OCL analysis tools (e.g. redundancy of an invariant) lead to the verification of other properties, chosen as a correctness property, described in the previous sections.

4 Example: Transformations of Statecharts and Petri Nets

In order to show the feasibility of our approach in practice, we show its application in the case of transformation between UML Statecharts and Petri Nets models.

To do this, we will go through all the steps of the proposed algorithm, applying it to the specified languages.

4.1 Analysis of the UML Statechart and Petri Net Modeling Languages

UML Statecharts as the Source Modeling Language

The formalization of UML Statecharts was in details described in [7], therefore here we describe only a simple UML model as running example. In our case we analyze the UML models of a voting process which requires a consensus from the participants.

In the system (Fig. 1), a specific task is handled by several calculation units CalcUnit, which send their local decision to the Voter buffer (where decision is ‘yes’ or ‘no’). The decision is considered accepted if all responses received by the Voter are ‘yes’. At the final stage, the Voter notify all calculation units about decision taken (where decision is ‘accept’ or ‘decline’). In what follows, we consider the system with two calculation units, that rather simplifies more general parameterized case.

Petri Nets as the Target Modeling Language

Petri Nets are often used to formally describe the dynamic semantics of the system. This popularity can be explained by the simplicity of this model, as well as the possibility of converting to other similar notations.

According to the metamodel of this modeling language (Fig. 2), 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) [9].

Dynamic concepts, which can be manipulated by rules (i.e., attributes tokens, and fires) are printed in bold italic. The operational behavior of Petri Net models is captured by the notion of firing a transition. Example of algorithm for transition between is described in [7] and consists of four stages, which are responsible for applying the rules, adding new and removing unmatched nodes.

4.2 Defining the Model Transformation

Modeling Statecharts by Petri Nets

In this section we describe main rules, used to translate Statecharts components into the Petri Net model.

First of all, each state in Statechart is translated into a corresponding place in the Petri Net model. Using tokens in the place, we can identify, that the original state for this place was an active state. From this point of view, one token is allowed on each level of the state hierarchy (forming a token ring, or formally, a place invariant).

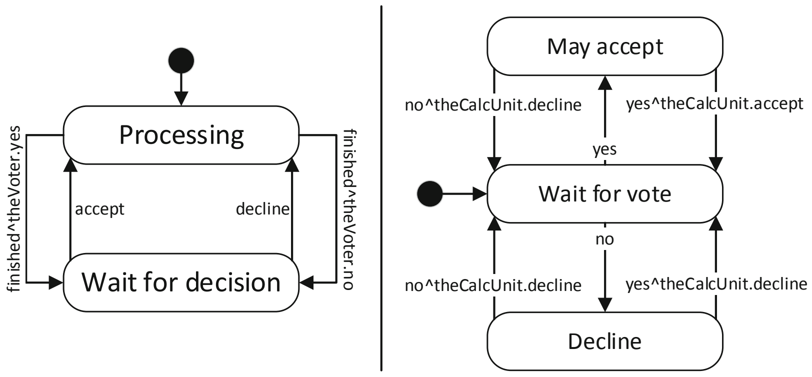

Fig. 1. Statemachines of CalcUnit (left) and Voter (right)

After the states, model messages from the event queues of a statemachine are translated into corresponding places. This stage of the transformation is not always straightforward, but we leave it out the scope of this study, assuming the approach described in [7].

Finally, each step in Statechart is transformed into a corresponding Petri Net transition. To do this, in fired transitions tokens are removed from source places (including event queue places) and new tokens are generated for all the target places and receiver message queues.

Applying previously provided rules to our Statechart, the following Petri Net can be derived (Fig. 3). In Fig. 3, for improving legibility, only a single transition (leading from state may_accept to wait_for_vote and triggered by the yes event) is shown. In this target model all original places of the Voter are replaced with the corresponding states and message queues for valid events, the initial state is marked by a token. Transitions have two incoming arcs as well, one from its source state.

Formalizing Model Transformations

In Sect. 3 we mentioned, that model transformations can be formalized using graph-oriented approach like [13]. Such formalization can be automated using special tools, for example VIATRA with a set of graph-transformation rules (in XMI or QVT) as its input.

The main element of such automation is the definition of inductive invariants, which establish correspondences between the elements of the source and target models. For example, in our case the following invariant for transforming Statechart states into Petri Net places can be derived (Fig. 4).

According to this pair of rules (Fig. 4), each initial state in the source model is transformed into a corresponding place containing a single token, while each non-initial state is projected into a place without a token.

Such triple structure of the invariants (and corresponding transformation rules) with reference elements allows us to organize the bidirectional transformations, that simplify the process of verification of model transformation correctness in the following stages.

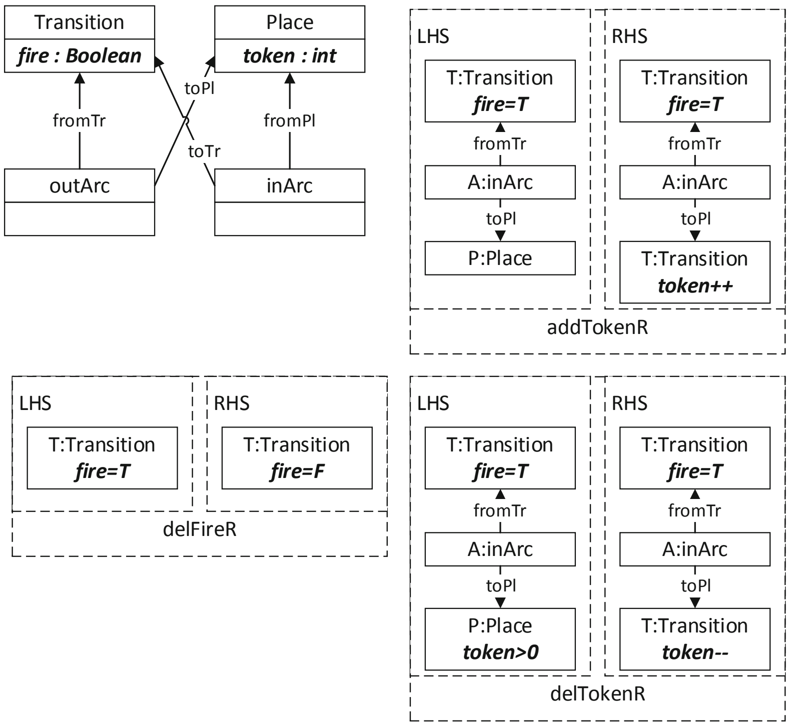

Fig. 2. A fragment of operational semantics of Petri Nets by graph transformation

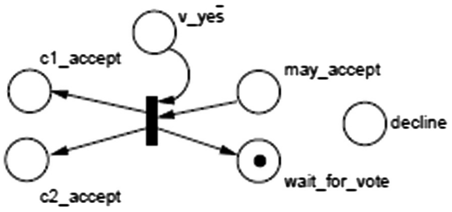

Fig. 3. A part of the Petri Net of the voter

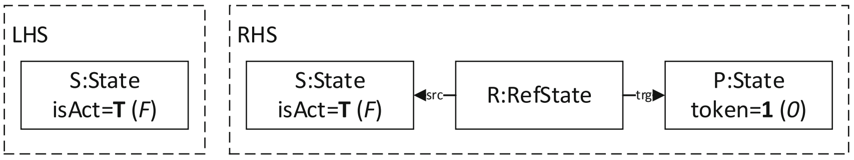

Fig. 4. Invariants for transforming active (*passive*) states into places

4.3 Verification of the Model Transformation

Generating Transition Systems

Transition systems are a common mathematical formalism that serves as the input specification of various model checker tools. In all practical cases, we have to 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 [15].

In what follows, we use the SAL syntax for the concrete representation of transition systems. This system is derived automatically from graph-oriented definition of model transformations between source and target model using the approach, described in [5].

Formalizing the Correctness Property

Finally, we define the property, which is used to validate both source and target models. The only prerequisite for such a property is the ability to formalize it, that allow a designer to automate the validation process.

In our case the following property will be used for Statechart model. For all OR-states (non-concurrent composite states), only a single substate is allowed to be active at any time during execution. From the formal point of view, this property can be formalized by the following invariant:

$$\nexists O : ORState, S_1 : State, S_2 : State : subvertex(O, S_1) \wedge subvertex(O, S_2) \wedge$$
$$isAct(S_1) \wedge isAct(S_2) \wedge S_1 \neq S_1$$

Informally, it prohibits the simultaneous activeness of two distinct substates S_1 and S_2 of the same *OR*-state.

Unfortunately, this criterion is difficult to check in terms of the target Petri Net model, since both states, and message queues of objects, are transformed into places. In order to resolve this contradictory, the property can be concretized for the model-level, using specific states from our model. As a result, the invariant above will be re-written for all possible states of the original Statechart (Fig. 1), for example for states *wait_for_vote* and *may_accept* the following invariant is generated:

$$\neg(subvertex(O, wait\_for\_vote) \wedge subvertex(O, may\_accept) \wedge$$
$$isAct(wait\_for\_vote) \wedge isAct(may\_accept) \wedge wait\_for\_vote \neq may\_accept)$$

After this concretization we have three different, non-conflicting correctness properties, which are appropriate to be transferred to the target Petri Net model. Since the state hierarchy of Statecharts is not structurally preserved in Petri Net, the equivalents of the OR states are not projected into the target model. As a result, the corresponding property (see Table 1) contain only specific pairs of places having a token.

Table 1. Table of state transitions Petri nets.

Place	Place
wait_for_vote (token = 1)	may_accept (token = 1)
wait_for_vote (token = 1)	decline (token = 1)
decline (token = 1)	may_accept (token = 1)

At this point, we need to validate whether the equality (one) or inequality checks (more than one) are required in the property to be proved. We may conclude that checking equality is also sufficient.

As we mentioned before, constructing the pair of properties to be proved for property preservation is not always straightforward and cannot always be automated, since it requires a certain insight into the source and target languages and their transformation.

Model Checking the Target Model
At the final stage of the algorithm, taking into account the definition of Transition system *TS* with semantics defined as a Kripke structure), a correctness property p, the model checking problem can be defined as to decide whether p holds on all execution paths of the system.

Therefore, at this stage, 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 model transformation preserved the defined above correctness property for a specific source Statechart model and its target Petri Net equivalent.

5 Conclusion

In current research we demonstrated the model-level, modeling language independent and highly automated approach to formally verify the model transformations between different (in the general case) modeling languages. Proposed approach is based on the idea of finding invariants between entities of such modeling languages with the subsequent checking of them with dynamic consistency properties. Such consistency is a key element in ensuring semantic interoperability in the process of digital transformations. The ability to describe an enterprise from different points of view through consistent modeling languages allows stakeholders to achieve a single vision for the enterprise as a whole. Our approach can be one of the means to achieve such coherence. To demonstrate the feasibility of the approach, the case with transition from UML Statechart model into Petri Net model was analyzed.

In comparison with existing approaches like [8] and [10], which also use the ideas of automated model generation with subsequent correctness property checking, our approach doesn't depend on the modelling language and property chosen. Such independency follows from deriving invariants as stable logical structures from the model transformation rules. As a result, the verification procedure reduces to a simple check of two sets of OCL constraints between themselves.

Among the limitations on the proposed approach the state explosion problem may be noted. To resolve this limitation, we propose to improve the automated encoding into transition systems to improve the technics and extend it on larger scale model transformations.

References

1. Nonaka, I., Kodama, M., Hirose, A., Kohlbacher, F.: Dynamic fractal organizations for promoting knowledge-based transformation – a new paradigm for organizational theory. Eur. Manag. J. **32**(1), 137–146 (2014)
2. Heavin, C., Power, D.J.: Challenges for digital transformation – towards a conceptual decision support guide for managers. J. Decis. Syst. **27**(1), 38–45 (2018)
3. Mens, T., Czarnecki, K., Gorp, P.V.: A taxonomy of model transformations. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
4. Degrandsart, S., Demeyer, S., Van den Bergh, J., Mens, T.: A transformation-based approach to context-aware modelling. Softw. Syst. Model. **13**(1), 191–208 (2014)
5. Bergmann, G., Ráth, I., Varró, G., Varró, D.: Change-driven model transformations. Softw. Syst. Model. **11**(3), 431–461 (2012)
6. Schürr, A.: Graph-transformation-driven correct-by-construction development of communication system topology adaptation algorithms. In: Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D., Seidl, C. (eds.) Modellierung. LNI, pp. 15–29. Gesellschaft für Informatik, Bonn (2018)
7. Rahim, L.A., Whittle, J.: A survey of approaches for verifying model transformations. Softw. Syst. Model. **14**(2), 1003–1028 (2015)
8. Akehurst, D., Kent, S.: A relational approach to defining transformations in a metamodel. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45800-X_20
9. Bondavalli, A., Dal Cin, M., Latella, D., Majzik, I., Pataricza, A., Savoia, G.: Dependability analysis in the early phases of UML based system design. Int. J. Comput. Syst. - Sci. Eng. **16**(5), 265–275 (2001)
10. Küster, J.M., Abd-El-Razik, M.: Validation of model transformations – first experiences using a white box approach. In: Kühne, T. (ed.) MODELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69489-2_24
11. Amrani, M., et al.: A tridimensional approach for studying the formal verification of model transformations. In: Verification and Validation of Model Transformations (VOLT) (2012)
12. Hausmann, J.H., Heckel, R., Sauer, S.: 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)
13. Ulitin, B., Babkin, E.: Ontology and DSL co-evolution using graph transformations methods. In: Johansson, B., Møller, C., Chaudhuri, A., Sudzina, F. (eds.) BIR 2017. LNBIP, vol. 295, pp. 233–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-64930-6_17
14. Ruffolo, M., Sidhu, I., Guadagno, L.: Semantic enterprise technologies. In: Proceedings of the First International Conference on Industrial Results of Semantic Technologies, vol. 293, pp. 70–84 (2007)
15. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. J. Softw. Syst. Model. **4**(4), 386–398 (2005)