Noname manuscript No.
(will be inserted by the editor)

Automated Formal Verification of Visual Modeling Languages by Model Checking*

Dániel Varró

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521, Budapest, Magyar tudósok körútja 2.
e-mail: varro@mit.bme.hu

The date of receipt and acceptance will be inserted by the editor

Abstract Graph transformation has recently become more and more popular as a general, rule-based visual specification paradigm to formally capture (a) requirements or behavior of user models (on the model-level), and (b) the operational semantics of modeling languages (on the meta-level) as demonstrated by benchmark applications around the Unified Modeling Language (UML). The current paper focuses on the model checking-based automated formal verification of graph transformation systems used either on the model-level or meta-level. We present a general translation that inputs (i) a metamodel of an arbitrary visual modeling language, (ii) a set of graph transformation rules that defines a formal operational semantics for the language, and (iii) an arbitrary well-formed model instance of the language and generates a transitions system (TS) that serve as the underlying mathematical specification formalism of various model checker tools. The main theoretical benefit of our approach is an optimization technique that projects only the dynamic parts of the graph transformation system into the target transition system, which results in a drastical reduction in the state space. The main practical benefit is the use of existing back-end model checker tools, which directly provides formal verification facilities (without additional efforts required to implement an analysis tool) for many practical applications captured in a very high-level visual notation. The practical feasibility of the approach is demonstrated by modeling and analyzing the well-known verification benchmark of dining philosophers both on the model and meta-level.

Key words graph transformation – metamodeling – formal verification – model checking – model transformation

* This work was partially carried out during my visit to Computer Science Laboratory at SRI International (333 Ravenswood Ave., Menlo Park, CA, U.S.A.), and the University of Paderborn (Germany), and it was funded by the National Science Foundation Grant (CCR-00-86096), the SEGRAVIS Research Network, and Hungarian Scientific Grants FKFP 0193/1999, OTKA T038027, and IKTA 00065/2000.

1 Introduction

1.1 Visual modeling languages in system design

Best engineering practice for the past century clearly demonstrated that visual design methodologies take advantage over textual descriptions as they are more succinct and easily understood by humans. Describing the structure of a house or a car by words would result in several thousands of pages even for relatively small systems. When concerning computer controlled systems, visual modeling languages (VMLs) used, for instance, in control engineering or gate-level hardware design have also become dominant in their field. Moreover, the use of computer-aided design tools (like ArchiCAD, or Matlab/Simulink) gave an additional boost to productivity and quality as minor human slips could be pointed out automatically in many cases.

Unfortunately, the majority of *software systems* are not yet comparable to these traditional products of engineering in their quality and reliability despite the wide range of existing visual modeling techniques. The main reason is that (i) on the one hand, high-level VMLs frequently used in engineering practice are typically imprecise and ambiguous, and (ii) on the other hand VMLs with precise formal semantics are typically too low-level for direct use in software engineering. In this sense, a compromise is frequently needed between expressibility (or usability) and analyzability.

1.2 Towards automated verification of UML designs

The Unified Modeling Language (UML), which is the dominant object-oriented modeling language for the design process of IT systems, has undoubtedly become the most popular of all visual languages in the field of software engineering. However, despite its industrial success as being a unified and visual notation, the impreciseness of UML (i.e., the

lack of formal semantics) is still the major factor that hinders the general use of UML as a primary source language for (i) automated tools of formal verification and validation exploiting the results in the theory of formal methods, and (ii) automated code generators that would yield a provenly correct functional core of the target application.

In fact, due to the increasing complexity of IT systems (and modeling languages in the design process), conceptual human design errors will occur in any models of any high-level and even formal modeling paradigm. Thus, merely the *use* of formal specification techniques does not alone guarantee the functional correctness and consistency of the system under design. Therefore, automated formal verification tools (based on model checking, static analysis or theorem proving) are required to verify that the requirements are fulfilled by the system model.

- *Static analysis methods* are typically semi-decision techniques (see, e.g., [40]), i.e., their output is either yes/no or do not know. In this sense, they are computationally efficient but they cannot assure the overall correctness of the design.
- *Model checkers* (like, for instance, Murϕ [1], SAL [7], SPIN [31]) provide a highly automated decision technique for finite state systems. In case of a verification failure, they retrieve a counter example that shows an execution trace leading to an erroneous situation. However, model checkers suffer from the well-know *state space explosion* problem (pp. 172–175 in [41]), which frequently yields a practical limitation of ensuring the overall correctness of the design.
- *Theorem provers* (like PVS [16]) are not limited to finite state systems to establish the correctness of the design, but deductive verification is highly costly and time consuming as it requires significant user interaction for constructing a proof (pp. 208–210 in [41]). In this sense, deductive verification may become the bottleneck of a project. Moreover, deductive verification typically provides an "all-or-nothing" way to reason about the system, i.e., unsuccessful verification attempts do not provide meaningful insights into the system due to decidability issues. From the viewpoint of cost-efficiency, we can deduce that (current) theorem proving techniques do only scale up for verifying mission-critical parts of a system/algorithm.

Although, in theory, formal verification tools typically provide decision procedures to assess the correctness of the design, in practice, they may fail to assure the overall correctness of a complex system due to cost and time limitations. Even in such a case, formal methods can be used as highly automated debugging aids of software engineering applications. Since model checkers provide the highest automation rate among all these formal methods (where the correctness of a system specification is judged without user interaction), thus they are the primary target for such verification or debugging aids. As a result, specification errors can be detected in an a relatively early phase of the design process, prior to implementation, which frequently reduce the overall software development costs (especially in the implementation and testing phase) in addition to significantly improving the overall quality of the system.

As the input language of model checker tools is too low-level for a direct use, many transformations (e.g., [13, 34, 39]) have been developed recently to project behavioral UML models into the input languages of model checker tools (and preferably in the reverse direction as well) yielding a "push-button" method for UML designs (i.e., model checking is integrated into the UML CASE tool and thus it can be initiated by clicking on a menu item).

As currently UML (from version 2.0) is evolving into a family of modeling languages, the development of *many* of such automated transformations into model checker tools will be necessitated in the near future, which puts the stress on *metamodeling* and the *development of meta-level analysis techniques* in the UML modeling environment. By a meta-level analysis technique, we mean an analysis framework which also takes the modeling language as a parameter in addition to the instance model of the language.

1.3 Graph transformation and metamodeling

For many years, the abstract syntax of UML (and related profiles) has been defined visually by means of *metamodeling*. Metamodeling is a term for capturing the design of user models and modeling languages uniformly, in a single modeling framework. A straightforward representation of such models and languages can rely on the use of directed, typed, and attributed graphs as the underlying semantic domain.

In this sense, *graph transformation* [43] has recently become very popular as being a general, rule-based visual specification paradigm to formally capture (i) *requirements, constraints and behavior of UML-based system models* [10, 23], and (ii) the *operational semantics of modeling languages* based on metamodeling techniques [18, 22, 32, 47–49, 52]. Similar ideas are applied directly on formalizing transformations from UML into various semantic domains (Petri nets, SOS rules, dataflow nets, etc.) [24, 53].

Problem statement While graph transformation is very popular as a high-level and expressive specification formalism, the lack of proper techniques and, especially, tools for the formal *analysis* of such specifications aiming to decide whether a certain user requirement (such as the absence of deadlocks, safety and liveness properties) holds in the system model hinders the use of graph transformation systems (GTS) in an effective design process for systems and visual modeling languages.

While several conceptual approaches have been proposed recently as a formal analysis technique for GTSs, existing graph transformation tools only provide *simulation* capabilities to assess whether a certain requirement holds in the system model, which is frequently insufficient for verification purposes. Since developing an analysis tool from scratch

is very costly, one should possibly exploit existing model checker tools to carry out the verification of GTSs.

Related work Unfortunately, the formal verification of models (and modeling languages) defined by graph transformation systems has remained so far on a rather theoretical (conceptual) level.

- The theoretical basics of verifying open graph transformation systems by model checking techniques have already been studied thoroughly in, e.g., [27, 28] (and subsequent papers). The authors propose that graphs can be interpreted as states and rule applications as transitions in a transition system. Unfortunately, as we demonstrate in the paper, this direct encoding of graphs into a model checking problem is, unfortunately, infeasible in practice since model checkers will easily run out of space due to this verbose state representation.
- Ongoing research into the same direction (with an envisaged tool support) has recently been sketched in [42] aiming to extend earlier results on reasoning about allocation and deallocation problems [19].
- A recent framework [3, 4] aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique (based on foldings and unfoldings of a special class of Petri nets). This framework is able to handle infinite state systems by calculating a representative finite complete prefix. However, the class of GTSs they handle has certain drawbacks from a practical, model engineering point of view concerning intuitiveness and expressiveness. Probably the most severe of these limitations is that the removal of nodes is not allowed. In contrast, graph transformation rules in our approach are arbitrary rules following the single pushout (SPO) approach [21] (with straightforward extensions to the double pushout (DPO) approach [15]); however, the price we have to pay is that our graph transformation system has to be a priori bounded.
- Rule invariants have been proposed lately in [38] in analogy with the notions of transition invariants in Petri nets. The authors also transfer the traditional concepts of liveness, boundedness, etc. to GTSs. The main conceptual limitation from a verification point of view is that rule invariants in a GTS (like transition invariants in a Petri net) only provide a *semi-decision* technique. In other words, they detect *potential* cycles in the system, some of which might never occur on any execution paths. On the other hand, rule invariants can be computed efficiently only from the static structure of the GTS.

As a summary, none of the frameworks give direct suggestions on concrete implementation or tool support how to verify formal specifications given in the form of graph transformation systems by existing model checking tools.

1.4 Objectives

In the current paper, we extend our initial ideas already discussed in [50] and propose a *meta-level and optimized technique with benchmark examples* to verify graph transformation systems used as either a *model-level* or *meta-level* formal specification technique by existing model checkers.

- *Graph transformation on the model-level*. For any user model with structural descriptions in the form of traditional class and object diagrams and dynamic behavior captured by graph transformation systems (like FUJABA [36] or PROGRES [46] models as practical examples), we project it into a behaviorally equivalent transition system (TS).
- *Graph transformation on the meta/language level*. For any well-formed model of any high-level modeling language (with abstract syntax defined by metamodeling and operational semantics formalized by graph transformation), we generate a separate (metamodel and model-specific) transition system that faithfully represents the behavior of the model instance.

We present a *meta-level analysis technique* where only the semantics of a modeling language should be defined precisely when a new modeling language is constructed, and then the formal analysis can be carried out automatically (without designing individual mappings into analysis tools). In addition, the maintenance of meta-level analysis frameworks (like the one proposed in the current paper) is much easier since modifying the semantics of language on the GTS level is less erroneous then modifying a complex translation program.

The output TS is generated in two steps. First all potential applications of a graph transformation rule are collected into separate transitions by a Cartesian product construction. Then, in a second phase, an optimization is carried out which eliminates the static parts and dead transitions from the target TS to drastically reduce the state space.

In order to capture the translation problem at the right level of abstraction, moreover, to gain a certain level of independence of particular model checker tools, both the source and the target model of our mapping (and the mapping itself) will be defined in the form of abstract state machines (ASMs) [25] (i.e., both the source GTS and the target TS). ASMs will provide a uniform semantic representation to formalize and prove the correctness and completeness of our approach.

The input languages of concrete tools can be generated by further preprocessing step (which is rather syntactic and thus out of the scope of the current paper). Meanwhile, running examples on model checking specifications will always be given in the concrete tool format of the SAL framework [7] to provide guidelines on how to tailor our technique to existing tools.

The practical feasibility of our approach is demonstrated on the well-known example of the dining philosophers, which is a common benchmark for assessing the performance of verification tools. The dining philosophers' problem will be modeled and analyzed in different ways (with graph transformation appearing both on the model-level and meta-level) to prove deadlock freedom and safety properties.

As a summary, the main benefits of our approach are the following.

1. We present a *meta-level analysis techniques* which is parameterized by (the metamodel of) the modeling language.
2. We present an *optimization technique* that reduces the number of state variables and eliminates dead transitions to avoid state space explosion.
3. We build on *existing model checker tools* to speed up the verification process (and the work related to implement the verification tool itself).
4. We assess the *practical feasibility* of our approach on a *verification benchmark*.

Finally, we also explicitly summarize the limitations/ prerequisites of our approach (although the detailed explanations of these limitations will be provided later on in the paper).

1. We assume that the structure of a modeling language is defined by a metamodel (UML class diagram), while the dynamic behavior is captured operationally by graph transformation rules.
2. We assume that an initial instance model (UML object diagram) is also provided by the user.
3. We suppose the links (edges) are relations on objects (nodes) thus they do not have identities.
4. We assume that attributes of objects have finite domains.
5. We assume that an explicit upper bound is a priori known to each class in the metamodel (e.g., a class A is allowed to have at most 5 instances in an instance model).

Note that while our technique is applicable to modeling languages from arbitrary domains (if defined by means of metamodeling and graph transformation), in the paper, we will rather focus on applying it for software engineering purposes, as proving that an IT system will not collapse under within conditions is probably the most challenging task in our opinion.

The structure of the paper The rest of the paper is structured as follows. In Sec. 2, a short introduction is provided on model, metamodels and graph transformation. Section 3 introduces basic notions of transition systems and model checking. In Sec. 4, we provide a unified semantic representation of GTSs and TSs based on abstract state machines. In Sec. 5, we provide a detailed discussion on how to map graph transformation systems into transition systems. The correctness and completeness of this encoding is thereafter proved in Sec. 6. The feasibility of our approach is demonstrated on the verification case study of Sec. 7, while Sec. 8 concludes our paper. The paper contains two appendices, App. A gives a brief semi-formal overview on abstract state machines, while App. B contains SAL code extracts of the dining philosophers case studies.

2 Specifying Models and Modeling Languages

Initially, we semi-formally summarize the major concepts for defining models and modeling languages by a traditional combination of *metamodeling* (Sec. 2.1) and *graph transformation* (Sec. 2.2) techniques that will serve as the input for our transformation approach later on. As our method is planned to be applied both to models and modeling languages (and as the modeling terminology, i.e., MOF [37] and UML are very close to each other) the terms (i) class diagrams and metamodels, and (ii) object diagrams and (instance) models will be used interchangeably, even though they can be very different from an application point of view.

2.1 Models and metamodels

The *abstract syntax* of domain specific modeling languages is defined by a corresponding metamodel, which conforms to the best engineering practices in visual specification techniques. Frequently, models (denoted as M in the sequel) and metamodels (referred as MM) are represented internally as typed, attributed and directed graphs with nodes for classes (objects) and directed edges for each navigable association (link) end together with the corresponding typing homomorphisms [14]. Attributes (slots) associated to classes (objects) can be interpreted as attributes (or slots) on graph nodes. For the current paper, *we restrict our models* in such a way *that only one link of a certain type may lead between two objects* (thus a link is a binary relation on objects).

As a conclusion, we use the following "generalized" notation of models and graphs.

- For the **metamodel elements**, we use the terms *classes*, *associations*, and *attributes* (also instead of the notions of node, edge and attribute types) in their traditional sense.
- For **model elements** (elements of a user model on the model level), the terminology of *objects*, *links*, and *slots* is used.
- Finally, for **rule elements** (contents of a graph transformation rule) we use the terms *nodes*, *edges*, and *slots*.

The distinction between model and rule elements is irrelevant from a modeling point of view (both are on the model-level); however, it follows the traditional notation.

In addition to these modeling elements, the traditional notion of inheritance is also interpreted for classes (but not for association and attributes as, for instance, in [52]). Thus an instance object of a subclass is an instance of the superclass as well, moreover, attributes of the superclass are also accessible in the subclass.

At this very initial point, we distinguish between *static and dynamic model (graph) elements* (following the guidelines of [49]). By dynamic model elements we mean elements that can be altered, (updated, removed or added) during the execution of models, which can be easily collected by analyzing the structure of the behavioral specification (graph transformation rules in our case). For a notational guidance, dynamic elements will appear in red in models and metamodels (with additional dashed lines in case of links and printed in italics for text).

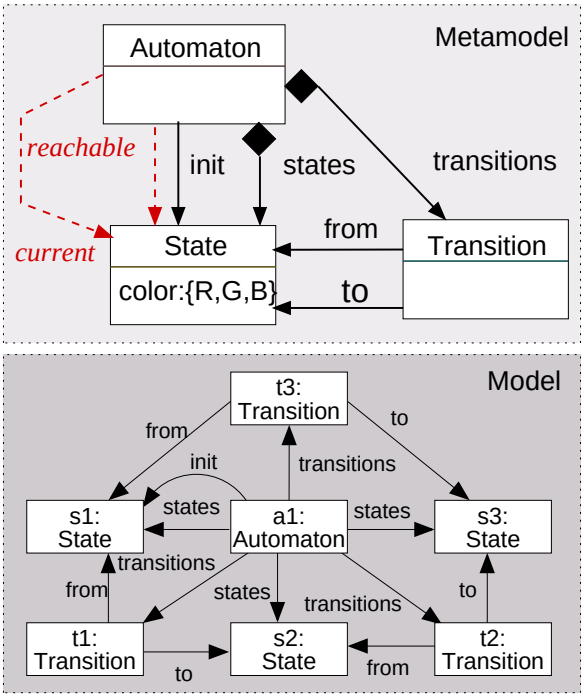

Fig. 1 A metamodel and model of finite automata

Example 1 A sample metamodel and a simple model of finite automata are depicted in Figure 1.

According to the metamodel, a well–formed instance of a finite *automaton* is composed of *states* and *transitions*. A transition is leading from its *from* state to its *to* state. The initial states of the automaton are marked with *init*, the active states are marked with *current*, while the reachable states starting from the initial states are modeled by *reachable* edges.

A sample automaton a1 consisting of three states (s1, s2, s3) and three transitions between them t1 (leading from s1 to s2), t2 (leading from s2 to s3), and t3 (leading from s2 to s3) is also depicted. We can notice that the initial state of a1 is s1.

In order to concentrate on dynamic behavior of models (and languages), we assume for the rest of the paper that our models and metamodels are all well formed (thus well-formedness checks are not included in the algorithms presented in the paper).

2.2 Graph transformation rules

For the current paper, we suppose that the *dynamic operational semantics* of a modeling language is specified by graph transformation rules. In the sequel, we define graph transformation rules and their application to a user model in an operational way which is functionally equivalent with the category-theoretical single pushout approach (SPO) [21] but requires less mathematical preparations.

Definition 1 *A **graph transformation rule** is a 5-tuple* $R = (Lhs, Neg, Rhs, Cond, Assgn)$*, where* Lhs *is the left-hand side graph,* Rhs *is the right-hand side graph, while* Neg *denote the (optional) negative application condition graph(s). Each node in* Lhs *and* Neg *may contain additional conditions* $Cond$ *for attributes/slots, while each node in* Rhs *may have attribute assignments* $Assgn$ *associated to them.*

Definition 2 *The **application of a rule** to a model* M *(i.e., a well-formed instance of its metamodel* MM*) rewrites the model by replacing the pattern defined by* Lhs *(restricted by prohibited subgraphs of* Neg *and attribute conditions* $Cond$*) with the pattern of the* Rhs*. This is performed as follows.*

1. *Find a matching*, i.e., a homomorphic (possibly not isomorphic) image of the Lhs graph in the model M (by graph pattern matching).
2. *Check the negative application conditions* (Neg) which prohibit the presence of certain submodels (objects and links) in M,
3. *Checking attribute conditions.* When a node (object) is matched, all its associated attribute conditions are checked and if any of them is violated then the current matching is discarded.
4. *Remove*. A part of the model M that can be mapped to the Lhs but not the Rhs graph is then removed (yielding the context model). When the image of a node (i.e., an object) is to be removed, all possible *dangling links (edges)* have to be removed as well in order to follow the SPO approach.
5. *Glue*. Rhs and the context model are glued together (by adding new objects and links) to obtain the derived model M'.
6. *Update attributes.* Finally, attribute updates are performed in accordance with $Assgn$.

For an informal, and more procedural interpretation of graph transformation, let us assign a variable to each node in a graph transformation rule. During the pattern matching phase, each node in the Lhs is instantiated with a type-conformant object taken from the instance model. Moreover, we have to check that if there is an edge leading from node A to node B, a corresponding link should be existent in the instance model leading from the image of A to the image of B (where the term “image” refers to the instantiated object). The deletion phase may remove objects and links already identified by the instantiation of variables. Finally, the variables of nodes appearing only in the Rhs are instantiated during the creation of the corresponding objects (and links).

Example 2 A pair of rules describing how the reachability problem on finite automata can be formulated by graph rewriting rules is depicted in Figure 2.

For instance, rule *initR* is structured as follows.

– The Lhs graph consists of two nodes (objects) (A1 of type Automaton, and S1 of type State) and an edge (link) of type initial.

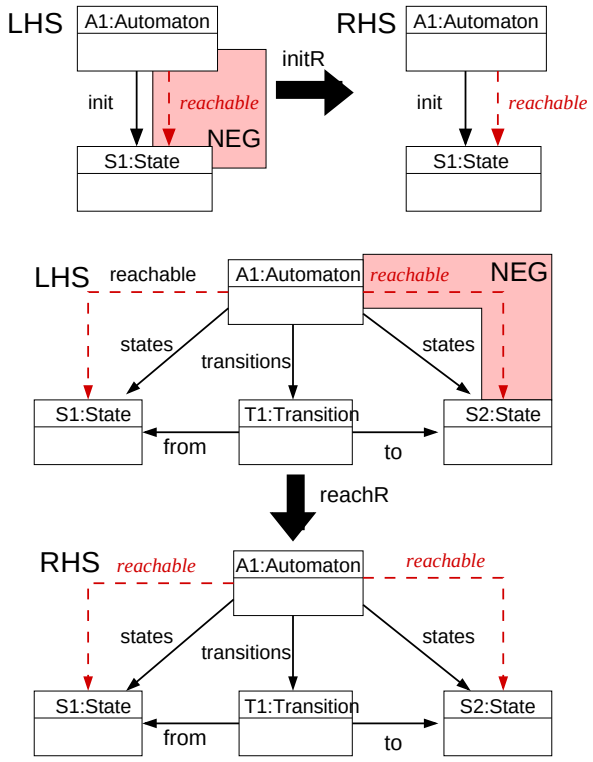

Fig. 2 Calculating reachable states by graph transformation

– The Neg graph consists of the same two nodes (A1 and S1) and an edge of type reachable. Negative application condition graphs are denoted by shaded areas in figures labeled with the NEG keyword.
– The Rhs graph consists of the same two nodes (A1 and S1) and (a new copy of) an edge of type reachable.
– The rule does not contain attribute conditions or assignments.

Rule *initR* in Fig. 2 states that all states of the automaton marked as initial are reachable (if the state has not been marked previously). When applying this rule to the finite automaton model of Fig. 1, a new reachable link is created leading from a1 to s1.

Rule *reachR* expresses that if a reachable state S1 of the automaton is connected by a transition T1 to such a state S2 that is not reachable yet then S2 should also become reachable as a result of the rule application. Note that without the negative application condition, the transformation would generate more than a single reachable link between an automaton and a state, which would contradict our assumption that only a single link of a certain type is allowed between two objects.

3 Model Checking Transition Systems

3.1 Transition systems

Transition systems (TS) are a common mathematical formalism that serves as the input specification of various model checker tools. They have certain commonalities (in many cases on the concrete language level as well) with structured programming languages (like C or Pascal) as the system/program is evolving by executing non-deterministic if-then-else like rules 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 upcoming definitions, we use a combination of [44] pp. 71–75 in [41] and as a reference.

Definition 3 *Formally, a **transition system** is a 4-tuple* $TS = (V, Dom, T, Init)$ *where*

1. $Dom = \{D_1, \ldots, D_m\}$ *is a set of finite **domains***
2. $V = \{v_1, \ldots, v_k\}$ *is the set of **state variables** taking their values from a corresponding domain. The domain of a variable is denoted as* $dom(v_j) = D_j$ *or shortly as* $v_j : D_j$ *where* $D_j \in Dom$*;*
3. $T = \{\tau_1, \ldots, \tau_n\}$ *is the set of **transitions (guarded commands)** which is of the form* $p \longrightarrow v'_1 := e_1, \ldots, v'_n := e_n$ *where the* p *is a boolean guard condition, and an action (or assignment)* $v'_j := e_j$ *specifies an update for state variable* v_j*;*
4. $Init$ *is a(n unquantified first order) predicate defining the **initial state**.*

Similarly to the majority of model checker tools, we suppose that state variables can be stored in state variable arrays $p_1, \ldots, p_m$ ranging on (sets of) object identifiers, and they can be referred to as $p_j[i] = v_i$. In this case, an n-dimensional state variable array $p[i_1] \ldots [i_n]$ has n *index* domains $ID_j \in Dom$ (for which $i_j \in ID_j$), and a *value* domain $VD \in Dom$. The intended meaning is that each state variable $v_{i_1,\ldots,i_n} = p[i_1] \ldots [i_n]$ (i.e., a location in the state variable array) takes its value from the value domain VD. Naturally, all index and value domains are required to be a priori finite. For instance, we will declare a one-dimensional state variable array color later in Example 3 with an index domain StateID = {s1, s2, s3} and value domain ColorType = {R, G, B}.

Note that transition systems are a kind of an abstract syntax of a *specification language*, which can be used to *generate* a state space (formally, a Kripke structure) describing the system.

Definition 4 *A **Kripke structure** $KS = (\Sigma, N, I, \sigma)$ is a four tuple where (i) Σ is the set of **states** (induced by all possible evaluations of state variables), i.e., $\Sigma = dom(v_1) \times \ldots \times dom(v_k)$; (ii) $N \subseteq \Sigma \times \Sigma$ is the **transition relation** defined as $N = \bigcup_{i=1}^{n} Act_{\tau_i}$ where Act_{τ_i} is a relation induced by the guarded commands of the TS, i.e., $Act_{\tau_i}(V, V') = p_i \wedge \bigwedge_{i=1}^{k} v'_{i,j} = e_{i,j}$; (iii) $I \subseteq \Sigma$ is the set of **initial states**; and (iv) $\sigma : \Sigma \to 2^{AP}$ is a **labeling function** mapping each state to a subset of atomic propositions AP (e.g., atomic equations) that are valid in the given state.*

Intuitively, a transition (guarded command) of the form $p \longrightarrow v'_1 := e_1, \ldots, v'_n := e_n$ can be executed in any state satisfying condition p. Thus condition p is called the *enabledness (guard) condition* of the transition τ and denoted as en_τ. We say that τ is *enabled* in a state s, if its condition p is satisfied in s (denoted as $s \models^{KS} p$ or simply $s \models p$). The effect of executing τ is that (i) all expressions $e_1, \ldots, e_n$ are calculated first based upon s, and then these new values are assigned to state variables $v_1, \ldots, v_n$. An *execution path* of a Kripke structure is an infinite sequence of states $s_0, s_1, s_2, \ldots$ starting from one of the initial states ($s_0 \models Init$) and progresses from one state to another by non-deterministically selecting and firing (enabled) transitions of the system.

The requirements (or properties to be verified) for models specified by a Kripke structure are frequently captured by some temporal logic formulae. However, since only safety properties and deadlock freedom are being proved in the current paper, we define these concepts without the use of temporal logic operations. As our technique focuses on the transformation of the *system specification* (which is independent of expressing requirements), we suppose that the requirements can be expressed by some formalism understood by the target model checker.

- **Safety properties**. A safety property ϕ_S ([33]) is an *invariant* (a boolean expression composed of atomic predicates) that must hold in each state of the system. Whenever a state is reached during the traversal of the state space where this property is violated, model checking can terminate immediately with an error message.
- **Deadlock freedom**. A system is in a deadlock (pp. 72–73 in [41]), if no transitions are enabled at the specific state. Here the corresponding deadlock property ϕ_D could be derived by combining the guards of transitions.

Now a simplified definition of the traditional model checking problem (for handling safety and deadlock properties that avoids the explicit use of temporal logic formulae) is as follows.

Definition 5 (Model checking problem) *Given (i) a system model in the form of a transition system TS (inducing a Kripke structure KS), and (ii) a* safety *property ϕ, then the model checking problem can be defined as to decide whether ϕ holds on each execution path of the system (i.e., whether $s_i \models \phi$ for all s_i on the execution path). Moreover, the system should be free of* deadlocks, *i.e.,* $\forall i : \exists \tau \in T : s_i \models en_\tau$.

After the mathematical definitions, we overview the concepts of a specific model checker tool that will provide the notation for examples on transition systems, since the language itself is very close to the mathematical definition.

3.2 SAL: Symbolic Analysis Laboratory

The SAL (Symbolic Analysis Laboratory) [7] framework aims at combining different tools for abstraction, program analysis, theorem proving, and model checking for the evaluation of system properties. The SAL architecture can be interpreted as a "tool–bus" where a collection of tools interact *through* the common intermediate language of transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate layer and the analysis results are fed back to this intermediate level.

- In the SAL intermediate language, the unit of specification is a context, which contains declaration of types, constants, transition system modules, and assertions. A basic SAL module is a state transition system where the state consists of *input*, *output*, *local*, and *global* variables, which refer to different access modes.
- A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition (or assignment) is of the form $x = expression$ or $x' = expression$, where x' refers to the new value of variable x in a transition. A guarded command is of the form $g \longrightarrow S$, where g is a boolean guard and S is a list of definitions of the form $x' = expression$. In addition to that, we may also define (auxiliary) functions that always yield a deterministic result.
- SAL modules can be composed (i) *synchronously*, so that $M_1||M_2$ is a module that takes M_1 and M_2 transitions in a lockstep, or (ii) *asynchronously*, when $M_1[]M_2$ is a module that takes an interleaving of M_1 and M_2 transitions.

Example 3 The SAL example below defines two *domains* (StateID and ColorType) and a one-dimensional *state variable array* color mapping (elements of) StateID to ColorType. For *initialization*, we assign the values R, G, and B to the locations s1, s2, s3 of the array, respectively. The single *transition* (guarded command) states that values R and G can be swapped at locations s1 and s2 (respectively) of array color.

```
% Domains
 StateID   : TYPE = {s1, s2, s3};
 ColorType : TYPE = {R, G, B};
% State variables
 GLOBAL color:  ARRAY StateID OF ColorType
% Initialization predicate
INITIALIZATION
 color[s1] = "R";
 color[s2] = "G";
 color[s3] = "B";
% Guarded commands
TRANSITION
 color[s1] = "R" AND color[s2] = "G" -->
   color'[s1] = "G";
   color'[s2] = "R"
```

In the paper, we will use the SAL specification language for code level examples when describing graph transformation systems as traditional state transition systems despite the fact that the SAL framework is not yet available for public use. However, as SAL is aimed to provide a general front-end

to *many* individual model checkers, we can achieve a high level of independence from concrete tools in exchange.

4 A Unifying Semantic Framework by Abstract State Machines

It is relatively common in various graph transformation tools (such as PROGRES [46] or VIATRA [48]) to formally represent models and metamodels as algebraic terms and graph transformation as manipulation of such terms. In the paper, we use Gurevich's abstract state machines (ASMs) [25] to provide a uniform semantic framework both for GTSs and TSs. ASMs treat the static structure of a language as terms over an arbitrary algebra and the dynamic behavior is captured by rules which may update certain functions thus allowing the algebra to evolve. The primary aim of using ASMs in the current paper is to ease the proof of correctness and completeness of our approach in Sec. 6.

Since the rules of ASMs are rather intuitive and algorithms specified by them are very close to traditional programming languages, we hope that the encoding technique presented in the paper using the ASM formalism is understandable as it is. However, for those who are particularly interested in the mathematical details but they are not familiar with ASMs, a brief introduction is given in App. A.

4.1 An ASM encoding of modeling languages and instance models

Now we discuss informally how the graph representation of models can be encoded into an algebraic representation that constitutes an ASM state.

The *vocabulary* (signature) of an ASM representation of a modeling language is driven by the metamodel. We define

- a *unary predicate* (boolean function symbol) for each class (such as *State*, *Transition*, and *Automaton* in our running example);
- a *binary predicate* for each association (e.g., *from*, *to*, *reachable*, etc.);
- a *unary function symbol* of a specific domain for each attribute (see the *color* attribute of states); and
- traditional *boolean* ($\top$, $\bot$) and *arithmetic function symbols* (such as +, $\leq$, $undef$) if required.

Instance models are encoded as ASM states (denoted as $\mathfrak{A}^{gr}$). The *superuniverse* $|\mathfrak{A}^{gr}|$ is composed of the set of unique object identifiers, the abstracted domains of attributes (i.e., some enumeration domains), and a reserve for fresh object identifiers (which are not used in the initial model). The *interpretation* of these function symbols is as follows.

- A *unary (class) predicate* cls is interpreted as $\top$ at location n_i (denoted as $cls^{\mathfrak{A}^{gr}}(n_i) := \top$), if the object identified by n_i of type cls is initially active (or existing) in the instance model. Otherwise the predicate is interpreted as false by definition: $cls^{\mathfrak{A}^{gr}}(n_i) := \bot$.
- For a *binary (association) predicate*, the interpretation is $asc^{\mathfrak{A}^{gr}}(n_i, n_j) := \top$ if the link of type asc between objects n_i and n_j is active at the beginning. Otherwise, the predicate is interpreted as false by definition: $asc^{\mathfrak{A}^{gr}}(n_i, n_j) = \bot$.
- For a *unary (attribute) function*, the interpretation is $att^{\mathfrak{A}^{gr}}(n_i) := val$, if n_i is an active object, and the slot of type att is equal to val initially, moreover, val is taken from the proper domain. Otherwise, by definition, the predicate is interpreted as undefined: $att^{\mathfrak{A}^{gr}}(n_i) = undef$.

Thereafter, a state in the ASM representation of a(n instance) model is defined by the *current evaluation of ground predicates* (denoted as $[\![p]\!]^{\mathfrak{A}^{gr}}$), i.e., predicates without free variables. (therefore the evaluation is independent of the environment ζ).

For a notational shorthand, we will frequently write $true$ instead of $\top^{\mathfrak{A}^{gr}}$, and $false$ instead of $\bot^{\mathfrak{A}^{gr}}$. Moreover, we also abbreviate $[\![p(n_i)]\!]^{\mathfrak{A}^{gr}} = true$ and $[\![p(n_i)]\!]^{\mathfrak{A}^{gr}} = false$ as $[\![p(n_i)]\!]^{\mathfrak{A}^{gr}}$ and $[\![\neg p(n_i)]\!]^{\mathfrak{A}^{gr}}$, respectively, which corresponds to the traditional intuitive representation of predicates.

Note that if $parent$ is a superclass of $child$ in the inheritance hierarchy (denoted as $parent \leftarrow child$) of the metamodel then the truth value of $child$ implies the truth value of $parent$ for all locations x in any state $\mathfrak{A}^{gr}$ (formally, $\forall x, \mathfrak{A}^{gr} : [\![child(x)]\!]^{\mathfrak{A}^{gr}} \rightarrow [\![parent(x)]\!]^{\mathfrak{A}^{gr}}$).

Example 4 The vocabulary Σ_{fa} induced by the metamodel of finite automata (Fig. 1) consists of automaton, state, etc. (for classes), states, reachable, current, etc. (for associations), color (for attributes), and R, G, B for constant symbols. The superuniverse $|\mathfrak{A}^{fa}|$ includes (at least) the following identifiers: $\{a1, s1, s2, s3, t1, t2, t3, R, G, B\}$.

The ASM state $\mathfrak{A}^{fa}$ of the instance model of Fig. 1 can be defined as $[\![\mathsf{automaton}(a1)]\!]^{\mathfrak{A}^{fa}}$, $[\![\mathsf{state}(s1)]\!]^{\mathfrak{A}^{fa}}$, $[\![\mathsf{states}(a1, s1)]\!]^{\mathfrak{A}^{fa}}$, $[\![\mathsf{init}(a1, s1)]\!]^{\mathfrak{A}^{fa}}$, $[\![\neg\mathsf{reachable}(a1, s1)]\!]^{\mathfrak{A}^{fa}}$, etc.

4.2 An ASM encoding of graph transformation rules

Now we informally discuss how graph transformation (following the SPO approach) can be formalized as ASM rules (for a precise formal treatment of encoding graph transformation rules as ASMs, see [51], for instance).

The ASM representation of a graph transformation (GT) rule (formalized in a ASM code pattern in Alg. 1) must appropriately handle (i) the *pattern matching* phase, (ii) the *deletion* of elements according to the difference of the LHS and the RHS, (iii) the removal of *dangling links*, (iv) the *creation* of elements according to the difference of the RHS and LHS, and (v) the *inheritance* in the class hierarchy.

1. *Pattern matching and checking negative conditions.* Let us first assign a variable x for each node in the rule. The *precondition* of the rule is constructed from (i) the LHS graph by instantiating variables of the LHS using

Algorithm 1 ASM representation of GT rules

Notation:
$\overline{X}_{lhs}$: variables related to nodes appearing in the Lhs;
$\overline{X}_{neg}$: variables appearing only in the Neg but not in the Lhs (i.e., related to nodes in $Neg \setminus Lhs$);
$\overline{X}_{pre}$: variables appearing in Lhs or Neg, i.e., $\overline{X}_{lhs} \cup \overline{X}_{neg}$;
$\overline{X}_{del}$: variables related to nodes in $LHS \setminus RHS$ (where $\overline{X}_{del} \subseteq \overline{X}_{lhs}$);
$\overline{X}_{add}$: variables related to nodes in $RHS \setminus LHS$;
$\overline{X}_{all}$: variables appearing in Lhs or RHS, i.e., $\overline{X}_{lhs} \cup \overline{X}_{add}$.

cls: a predicate derived from a class
asc: a predicate derived from an association
att: a predicate derived from an attribute
$parent \leftarrow child$: $parent$ is a superclass of $child$
$undef$: the undefined value
val: a value from a corresponding value domain

rule r_{spo} =
1: /* Instantiate variables that satisfy the precondition (i.e., both LHS and NEG) */
2: **choose** $\overline{X}_{lhs}$ **with** $\phi_{lhs}(\overline{X}_{lhs}) \wedge \neg(\exists \overline{X}_{neg} : \phi_{neg}(\overline{X}_{pre}))$ **do**
3: /* Deleting objects or links or slots */
4: $cls(x) := \bot$ or $asc(x,y) := \bot$ or $att(x) := undef$;
5: **if** $cls(x)$ is an object location to be falsified **then**
6: /* Removing dangling links */
7: **forall** z **with** $asc(x,z) = \top$ or $asc(z,x) = \top$ **do**
8: $asc(x,z) := \bot$ or $asc(z,x) := \bot$
9: **end for**
10: /* A similar treatment of slots belonging to the falsified object should come here */
11: /* Removing from the inheritance hierarchy */
12: **forall** cls_1 **with** $cls \leftarrow cls_1$ or $cls_1 \leftarrow cls$ **do**
13: $cls_1(x) := \bot$
14: **end for**
15: **end if**
16: **create** $\overline{X}_{add}$ **do**
17: **let** $x_1 \in \overline{X}_{add}, y_1, z_1 \in \overline{X}_{all}$ **in**
18: /* Creating objects or links or slots */
19: $cls(x_1) := \top$ or $asc(y_1, z_1) := \top$ or $att(y_1) := val$;
20: **if** $cls(x)$ is an object location set to true **then**
21: /* Modifying the inheritance hierarchy for all supertypes $parent$ of cls at location x */
22: **forall** $parent$ **with** $parent \leftarrow cls$ **do**
23: $parent(x) := \top$
24: **end for**
25: **end if**
26: **end create**
27: **end choose**

the non-deterministic *choose* construct, and (ii) the negative application condition graphs by creating a formula that prescribes the non-existence of variable assignments that satisfies the negative condition (Line 2). In this sense, a *matching pattern* of the GT rule is defined by extending the environment ζ by variable assignments for each variable in $\overline{X}_{lhs}$ (formally, $\zeta\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}; x_i \in \overline{X}_{lhs}$).

2. *Deletion of elements.* Then we perform the deletion of model elements by falsifying all the corresponding locations according to the $LHS \setminus RHS$ of the GT rule (Line 4).
3. *Dangling links.* When removing an object x of type cls (Line 5), a special care is required to appropriately handle potential dangling edges. For that reason, all locations of links (and slots) that are related to the object to be removed are falsified as well (Lines 7–9).
4. *Creation of elements.* In case of creation, the proper locations are set to true (Line 19). When creating an object of type p, we use the *create* construct (Line 16), which obtains a fresh identifier from the reserve.
5. *Handling of inheritance.* In case of removing an object of type cls Moreover, all locations in superclasses and subclasses of cls should be falsified as well (Lines 12–14). When creating a new object of type cls, all locations in superclasses (but not subclasses!) of p should be set to true as well (Lines 22–24) in order to faithfully preserve the inheritance hierarchy.

Note that Alg. 1 is not a real algorithm but only a scheme for the structure of the ASM rule that implements the GT rule. Concrete ASM algorithms will be presented in the upcoming example.

Example 5 The ASM encoding of rule *initR* (of Fig. 2) derived according to Alg. 1 is as follows.

rule initR_{spo} =
choose A_1, S_1 **with** $\mathsf{automaton}(A_1) \wedge \mathsf{state}(S_1) \wedge \mathsf{init}(A_1, S_1) \wedge \neg\mathsf{reachable}(A_1, S_1)$ **do**
$\mathsf{reachable}(A_1, S_1) := \top$
end choose

This example can be read as follows: in the next state, the reachable relation should become true for a (non-deterministic) assignment for variables A_1 and S_1 whenever the automaton function holds at A_1, state holds at S_1 and init holds at A_1, S_1, but $\mathsf{reachable}(A_1, S_1)$ is false in the current state.

Example 6 To demonstrate how to handle dangling links, the ASM representation of a rule that removes a transition of a finite automaton is the following.

rule $\mathrm{delTransR}_{spo}$ =
choose T_1 **with** $\mathsf{transition}(T_1)$ **do**
$\mathsf{transition}(T_1) := \bot$
forall X **with** $\mathsf{transitions}(X, T_1)$ **do**
$\mathsf{transitions}(X, T_1) := \bot$
end for
forall X **with** $\mathsf{from}(T_1, X)$ **do**
$\mathsf{from}(T_1, X) := \bot$
end for
forall X **with** $\mathsf{to}(T_1, X)$ **do**
$\mathsf{to}(T_1, X) := \bot$
end for
end choose

It is worth noting that only slight modifications are required in Alg. 1 to follow the DPO approach [15] instead

of SPO. First of all, instead of implicitly removing dangling edges (Lines 5–9), we should create a *dangling condition* formula ϕ_{dang} (by extending Line 2) to forbid the application of the rule in such a case. Moreover, the *identification condition* can be expressed by another formula ϕ_{iden} (in Line 2), which prescribes that we must not assign the same value during pattern matching to variables of objects to be removed.

4.3 An ASM encoding of transition systems

Since abstract state machines are generalizations of transition systems, the ASM encoding of transition systems is rather straightforward.

State variables, Domains, Initialization Since state variables are arranged into state variable arrays, we may assign a function symbol p for each state variable array $p[x]$. As a special case, a nullary dynamic function symbol in an ASM may represent a single state variable in a TS.

In TSs, each index and value domains D_i should be a priori bounded. Therefore we may assign sorts to function parameters and return values to express the fact that each parameter should be taken from the corresponding domain. As a result, for a TS with an n dimensional state variable array p with corresponding index domains ID_i and value domain VD, in the ASM representation we have $p(x_1 : ID_1, \ldots, x_n : ID_n) : VD$.

Thereafter the initialization predicate simply defines the initial state $\mathfrak{A}^{ts}_{init}$ in the ASM representation of the TS. Since, in general, the initial state of a TS may be non-deterministic, the ASM representation of the initial state may depend on the environment (in other terms, it may contain free variables).

Transitions (guarded commands) The most crucial restriction of TSs (in contrast to ASMs) is that non-determinism is only allowed to select from a set of enabled transitions, but if someone already selected a single transition $g \longrightarrow p_1[v_1] := e_1, \ldots, p_n[v_n] := e_n$, it should be deterministic. Therefore the guard g and the expressions e_i must not contain free variables, and the use of the *choose* construct is also prohibited in the ASM representation of a TS. Moreover (as a practical restriction), the majority of model checkers does not support quantified formulae in guards. Finally, Table 1 summarizes the ASM and SAL representation of a transition.

ASM	SAL
rule r_{ts} =	
if g **then**	`g -->`
$p_1(v_1) := e_1$	`p1'[v1] = e1;`
...	...
$p_n(v_n) := e_n$	`pn'[vn] = en;`
end if	

Table 1 A transition in a TS (ASM and SAL representation)

5 From Graph Transformation Systems to Transition Systems

In the current section, we provide a meta-level approach to map graph transformation systems into transition systems in order to verify properties of user models by model checking tools.

In other words, we propose a translation that inputs (i) the *metamodel* of a visual modeling language (or class diagram, on the model level), (ii) its operational semantics (dynamic behavior) in the form of a *graph transformation system*, and (iii) a concrete, well-formed *model instance* of the language (object model), and generates a *transition system* as the output.

Note that since we use model checkers, we do not reason about the properties of the language itself (as done by theorem provers). However, we can automatically prove certain correctness properties (like safety and deadlock freedom in our case study) for a well-formed specific but arbitrary instance model of the language.

It is essential to be pointed out that in practical cases, the user is only interested in the correctness of his or her model and not the correctness of the entire modeling language. Moreover, proving the correctness of a property for all valid model instances is often impossible.

5.1 A conceptual overview of the encoding

As demonstrated previously by [27, 28], graph transformation systems can be interpreted as a TS where the state space is constituted from attributed graphs created by elementary graph transformation steps (see Fig. 3 for an overview).

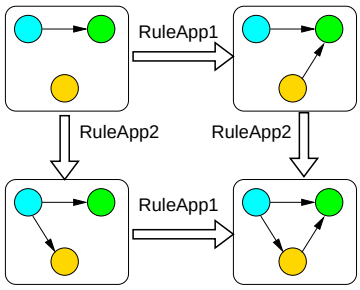

Fig. 3 The state space of graph transformation systems

This state space has a special structure: while the graph representation of a user model is typically finite (for instance, infinite UML models are somewhat rare), attributes may result in potentially infinite state representations (e.g., in case of integers or reals). As current model checking tools can only traverse state spaces induced by state variables of finite domain, variables having infinite domains should be abstracted to boolean domains before model checking, for instance, by a technique called *predicate abstraction* [44].

Thereafter, models having attributes of finite domain (either originally or after predicate abstraction) will form the

states of the TS, and they will be encoded as predicates over node identifiers. Applying a graph transformation rule for a single match will be represented as a *transition in the TS*.

The major challenge in such an encoding is that while graph transformation is a meta-level specification technique, transitions in a TS are defined on the model level. As a consequence, *a single graph transformation rule* is encoded into *several transitions in the TS*; moreover, *the same graph transformation rule may yield different enabled transitions* (even during the same execution) when applied to different instance graphs.

5.2 A naive encoding of graph transformation systems into transition systems

First we present a naive encoding of GTSs into TSs, which easily demonstrates how to derive a set of transitions for a single graph transformation rule, but which is inefficient from a verification point of view.

Mapping graphs into state variable arrays Since we introduced the same semantic representation for the graphs and state variable arrays, we can naturally map (i) each class into a one-dimensional boolean state variable array, (ii) each association into a two-dimensional boolean state variable array, and (iii) each attribute into a one-dimensional state variable array with enumeration range. However, since TSs are not as expressive as GTSs, the following additional assumptions are required.

- In order to define the corresponding index domains for these state variable arrays, we have to assume that *there exists an a priori upper bound for the number of objects in the model for each class*. In this respect, we suppose that when a new object is created it is only activated from the bounded "pool" of currently passive objects (deletion means passivation, naturally), and the same applies to the interpretation of links. This pool of initially non-existent objects is called the *reserve*.
- Concerning the value domains of state variable arrays, *attributes (slots) of infinite type have been abstracted into some representative finite domain*, which is carried out by the user (potentially with tool support) using predicate abstraction, for instance.

The main limitation imposed by these restrictions is that we cannot handle graph transformation systems with potentially infinite state space. In other terms, we carry out *bounded model checking* [12] , which is only able to efficiently traverse the state space up to a certain depth provided as a parameter. In our case, this parameter is the list of upper bounds for each class[1]. This list of upper bounds serves as an additional input to our translation (because an algorithmic "guess" for the upper bounds has computational complexity problems).

As a result of this encoding, we can trivially establish a mapping $\mathcal{F}_1$ from the ASM states $\mathfrak{A}^{gr}$ of a GTS to the ASM state $\mathfrak{A}^{nv}$ of a naive TS since the same function symbols are used with the same interpretation. Formally, $\mathfrak{A}^{nv} = \mathcal{F}_1(\mathfrak{A}^{gr})$ with (i) $\mathcal{F}_1(p) := p$ for all function symbol p, and (ii) $[\![p(x) = v]\!]^{\mathfrak{A}^{nv}} := [\![p(x) = v]\!]^{\mathfrak{A}^{gr}}$ for all locations of a p.

In this sense, the initialization predicate of the corresponding TS is thus defined by the initial instance model of the GTS.

Example 7 The naive SAL encoding of our finite automaton model (Fig. 1) would include the following lines.

```
% Domains
AutID : TYPE = {a1};
StateID : TYPE = {s1, s2, s3};
fa1 : MODULE =
% State variable arrays
BEGIN
 GLOBAL reachable: ARRAY AutID OF
                   ARRAY StateID OF BOOLEAN
 GLOBAL states:    ARRAY AutID OF
                   ARRAY StateID OF BOOLEAN
% Initialization predicate
INITIALIZATION
  states[a1][s1] = TRUE;
  states[a1][s2] = TRUE;
  states[a1][s3] = TRUE;
  reachable[a1][s1] = FALSE;
  reachable[a1][s2] = FALSE;
  reachable[a1][s3] = FALSE;
```

Mapping graph transformation steps to transitions The main task in encoding transformation steps (potential applications of graph transformation rules) into transitions of TSs is to simulate all the different behaviors imposed by the graph pattern matching process in a low-level structure. As graph transformation is a meta-level specification technique, *a single graph transformation rule will be encoded into several transitions*. In fact, all potential application of a rule (occurrences of a pattern) have to be collected at compile time and then enumerated explicitly as different guarded commands.

The explanation for this "unfolding" of GT rules stems from the fact that the pattern matching process of a GT rule is non-deterministic while such non-determinism is not allowed in the guard of a TS transition. Naturally, the selection of the next (enabled) rule to apply is still non-deterministic in both cases.

The number of potential transitions (according to a naive first estimation) is determined by the *complexity of the LHS* of a rule (i.e., the number of nodes), and *the size of the model* (i.e., the cardinality of domains of state variable arrays). We have to instantiate the variables of the LHS in all possible combinations by a Cartesian product construction to enumerate all potential matchings of the rule. Thus a node in the LHS is tried to be matched to each object in the model having a conformant type.

[1] The list of upper bounds is only required for dynamic classes, since we can easily calculate the upper bound for a static class depending on the model instance

Note that the Cartesian product of variable instantiation should also enumerate all possible combinations of object identifiers taken from the reserve (i.e., for objects that are not present in the initial model). However, as we assume to have an a priori upper bound for the model, it is no longer a problem.

The process of generating a single transition can be divided into three phases, namely, the generation of (i) the positive guard corresponding to the LHS, (ii) the negative guard corresponding to NEG, and (iii) the action part.

- *Guard of the LHS.* For a specific potential matching (i.e., combination of variable instantiations) of the LHS, the guard of the LHS is derived straightforwardly from the ground LHS formula obtained after substituting the values to variables.
- *Guard of the negative condition.* In order to ensure that *none* of the potential instantiations of the variables of the NEG graph can be satisfied for the given potential matching, we have to generate a conjunction of negative clauses, where each clause is a possible instantiation of the variables in NEG (alternatively speaking, a potential matching of the NEG graph). The guard of the transition generated for the negative application conditions is obtained as before from the ground NEG formulae.
- *Actions.* Prior to generating the actions that update certain locations in state variable arrays, we have to instantiate the variables of objects created by the GT rule application. In the paper, we abstract from this problem by assuming that there exists a special procedure *nextIdFromReserve()*, which retrieves the next location from the reserve at compile time. When all the variables in LHS and RHS are instantiated, we may simply copy the updates (in the ASM representation) of the GT rule.

The process of generating transitions in the TS from the ASM representation of a graph transformation rule is formally defined in a pseudo ASM code in Alg. 2.

1. First we collect all variables $\overline{X}_{lhs}$ in the LHS and $\overline{X}_{pre}$ in the NEG graph of the rule (Line 1).
2. Then we process one by one all elements $\langle a_1, \ldots, a_n \rangle$ in the Cartesian product of variable domains of $\overline{X}_{lhs}$.
   (a) In Line 3, we instantiate the variables $x_1, \ldots, x_n$ with $\langle a_1, \ldots, a_n \rangle$ to constitute a potential matching of the rule
   (b) In Line 4, we substitute them into the LHS formula $\phi_{lhs}(\overline{X}_{lhs})$ to obtain the first part (g_{lhs}) of the TS guard.
   (c) In Line 5, we proceed similarly to create the conjunction of the negative condition NEG formula(e) $\phi_{neg_i}(\overline{X}_{neg})$ to obtain g_{neg}. This conjunction expresses that no assignments for variables $\overline{X}_{pre}$ may satisfy any of the negative condition formulae $\phi_{neg_i}(\overline{X}_{pre})$.
   (d) We instantiate (in Lines 6–8) all variables in $\overline{X}_{add}$ with elements taken from the reserve of the corresponding domains.

Algorithm 2 A naive generation of transitions in a TS

Notation:
$\overline{X}_{lhs}$: variables related to nodes appearing in the Lhs;
$\overline{X}_{pre}$: variables appearing in Lhs or Neg, i.e., $\overline{X}_{lhs} \cup \overline{X}_{neg}$;
ζ: variable assignment
$x \mapsto a$: value a is now assigned to variable x
$dom(x)$: domain (or sort) of variable x

ϕ_{lhs}: the formula derived from the LHS of a rule
ϕ_{neg}: the formula derived from the Neg graph of a rule
p_i: a predicate in the LHS formula
$p_{i,j}$: a predicate in the NEG formula
$nextIdFromReserve()$: a special function to handle symmetries when creating object; called at compile time
$\tau \equiv g \longrightarrow act$: a transition in the target TS with guard g and action act

fun *naive_transitions* =
1: **let** $\overline{X}_{lhs} \equiv \{x_1, \ldots, x_n\}$, $\overline{X}_{pre} \equiv \{y_1, \ldots, y_m\}$,
2: **forall** $\langle a_1, \ldots, a_n \rangle \in dom(x_1) \times \ldots \times dom(x_n)$ **do**
3: $\zeta_{lhs} := \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$
4: $g_{lhs} := \phi_{lhs}(a_1, \ldots, a_n) \equiv \bigwedge_i p_i$
5: $g_{neg} := \bigwedge_i \neg\phi_{neg_i}(b_1, \ldots, b_m)$ where $\langle b_1, \ldots, b_m \rangle \in dom(y_1) \times \ldots \times dom(y_n)$ and $\phi_{neg_i} = \bigwedge_j p_{i,j}$
6: **forall** z_i **with** $z_i \in \overline{X}_{add}$ **do**
7: $z_i \mapsto$ *nextIdFromReserve*$(dom(z_i))$;
8: **end for**
9: **forall** updates $p_k(x_k) := v_k$ in rule r_{spo} **do**
10: $act :=$ **par**$_k$ $(p_k[x_k] := v_k)$
11: **end for**
12: $\tau := (g_{lhs} \wedge g_{neg}) \longrightarrow act$
13: **generate** τ
14: **end for**

   (e) We define (in Lines 9–11) a TS action act as the parallel composition of assignments (as defined by Alg. 1 to add and remove graph elements).
   (f) Finally (in Lines 12–13), we generate a TS transition τ composed g_{lhs} and g_{neg} as guards and act as action.

Example 8 In case of rule *initR* (of Fig. 2), the corresponding SAL specification in a naive encoding is as follows.

```
TRANSITION % guarded commands for initR
% first potential match
 automaton(a1) AND init(a1,s1) AND state(s1)
 AND NOT (reachable[a1][s1]) -->% guard
    reachable'[a1][s1] = TRUE;  % assignment
[] % asynchronous composition
% second potential match
 automaton(a1) AND init(a1,s2) AND state(s2)
 AND NOT (reachable[a1][s2]) -->
    reachable'[a1][s2] = TRUE;
[]
% third potential match
 automaton(a1) AND init(a1,s3) AND state(s3)
 AND NOT (reachable[a1][s3]) -->
    reachable'[a1][s3] = TRUE;
```

Handling symmetries in object creation Probably, one of the most crucial design decisions one has to make is con-

cerned with the creation (activation) of objects in order to handle symmetries (or isomorphism) in an appropriate way. The problem relies in the fact that graph transformation only prescribes to introduce new objects (with fresh identifiers), while in a TS, we have to specify precisely which is *the* identifier for the new object.

A naive approach may simple generate all the different ways how a new object can be created (by testing whether an object having a certain identifier is active or not); however, such a solution would result in an unacceptable amount of transitions considering isomorphic (or symmetric) cases differently.

Our proposal (that was implemented in [45]) is to maintain a counter for each class of objects pointing to the next free object identifier. In this sense, a total order is defined on identifiers (starting, for instance, with initially active object identifiers), and when a new object is to be activated, the location indexed by the next identifier (according to the ordering relation) will be set to true.

The drawback of the solution is that each object is allowed to be activated only once in its life-time (a cyclic activation and passivation is thus not allowed for the same identifier), which might require a larger input model to be considered. Since allowing to create each object once is close to the object-oriented philosophy, this limitation is not crucial in our opinion.

5.3 Optimizations in transition systems

When the current approach was applied for encoding and verifying UML statecharts formalized by graph transformation systems (following [49]), we revealed that the previous encoding consumes an unacceptable amount of space when model checking even small applications. For instance, the encoding of an automaton having 20 states and 20 transitions requires more than 500 boolean state variables, which is typically far too many to be handled by state-of-the-art model checking tools (resulting in a state space having 2^{500} states).

The problem originates from the fact that in the previous naive approach, both state variables and transitions were introduced "verbosely" for the static parts of a model as well.

Eliminating static state variables Supposing that the structure of a finite automaton remains unchanged during the lifetime of the model, our translation needs to *create state variables only for dynamic elements* (such as current or reachable links in the metamodel of finite automata), while *static parts* can be omitted by compile-time preprocessing.

Example 9 The optimized SAL encoding of our finite automaton model (Fig. 1) would include the following lines. Note that lines of code in Example 7 that are not part of Example 9 (such as the `states` state variable array) are eliminated as the result of our optimization.

```
AutID : TYPE = {a1};
StateID : TYPE = {s1, s2, s3};
```

```
fa1 : MODULE =
BEGIN
 GLOBAL reachable: ARRAY AutID OF
                   ARRAY StateID OF BOOLEAN
INITIALIZATION
  reachable[a1][s1] = FALSE;
  reachable[a1][s2] = FALSE;
  reachable[a1][s3] = FALSE;
```

Naturally, as a specific transition may never be applied during the execution of a specific model, such an optimized encoding may still introduce state variables for model elements that are never changed. However, the only possibility to eliminate such unreachable model parts requires at compile time the use of some sophisticated static analysis techniques on graph transformation rules (which triggers further research).

Eliminating dead transitions Our naive approach may easily generate redundant transitions with guards that can never be satisfied, although each one is investigated and tested at every step over and over again causing an unacceptable decline in performance. For instance, the finite automaton model in our running example (Fig. 1) has a single initial state `s1`, therefore the transitions generated in Example 8 for `a1-s2` and `a1-s3` pairs are superfluous as *the guards will constantly be false due to the static structure of the model*.

To eliminate such an overhead, in Alg. 3, we eliminate transitions with guards that can never be satisfied by further preprocessing the TS representation of a graph transformation system.

The positive conditions $g_{lhs} = \bigwedge_i p_i$ (generated in accordance with the LHS of the GT rule) are processed as follows (Lines 4–12).

- If a (positive) literal p_i is constantly evaluated to false (i.e., it refers to some static parts which is not present in the initial model), then the corresponding transition τ is eliminated (Lines 5–6).
- If a literal p_i is constantly evaluated to true (i.e., it refers to some static parts which is present in the initial model), then the guard of τ is truncated by removing p_i (Lines 7–8).
- If the truth value of a literal p_i may vary (as it is dynamic) then it is kept as it is in the guard of τ (Lines 9–10).

The negative conditions $g_{neg} = \bigwedge_i \neg\phi_i \equiv \bigwedge_i(\neg(\bigwedge_j p_{i,j})$ (generated in accordance with the NEG condition graph(s) of the GT rule) are processed follows (Lines 13–24).

- If for all j literals $p_{i,j}$ are constantly evaluated to true, then a matching pattern of the NEG part of the rule is found which always prevents rule application thus we remove the entire transition τ (Lines 14–15).
- If a literal $p_{i,j}$ is constantly evaluated to false (Lines 18–19), then the current conjunct ϕ_i of the negative condition can be removed from the guard. Since the application of the GT rule may still be prevented by another conjunct ϕ_k thus we do not remove the entire negative condition in this step.

Algorithm 3 Eliminating dead transitions in a TS

Notation:
$\tau \equiv g_{lhs} \wedge g_{neg} \longrightarrow act$: a transition in the naive TS
p_i: a predicate in the LHS formula
$p_{i,j}$: a predicate in the NEG formula
$\neg\phi_i$: a (negated) conjunction of atomic predicates in NEG
$[\![p_i]\!]^{\mathfrak{A}^{init}}$: the truth value of p_i in the initial state $\mathfrak{A}^{init}$

fun *eliminate_dead_transitions* =
1: **forall** $\tau \equiv (g_{lhs} \wedge g_{neg} \longrightarrow Act)$ generated by *naive_transitions* **do**
2: **let** $g_{lhs} = \bigwedge_i p_i$
3: **let** $g_{neg} = \bigwedge_i \neg\phi_i \equiv \bigwedge_i(\neg(\bigwedge_j p_{i,j})$
4: **forall** $p_i \in g_{lhs}$ **do**
5: **if** p_i is static and $[\![p_i]\!]^{\mathfrak{A}_{init}} = false$ **then**
6: eliminate τ
7: **else if** p_i is static and $[\![p_i]\!]^{\mathfrak{A}_{init}} = true$ **then**
8: remove p_i from g_{lhs}
9: **else if** p_i is dynamic **then**
10: leave p_i in g_{lhs} as it is
11: **end if**
12: **end for**
13: **forall** $(\neg\phi_i) \in g_{neg}$ **do**
14: **if** $\forall j : p_{i,j}$ is static and $[\![p_{i,j}]\!]^{\mathfrak{A}_{init}} = true$ (thus $[\![\neg\phi_i]\!]^{\mathfrak{A}_{init}} = false$) **then**
15: eliminate τ
16: **end if**
17: **forall** $p_{i,j} \in \phi_i$ **do**
18: **if** $p_{i,j}$ is static and $[\![p_{i,j}]\!]^{\mathfrak{A}_{init}} = false$ **then**
19: remove ϕ_i from g_{neg}
20: **else if** $p_{i,j}$ is dynamic **then**
21: leave $p_{i,j}$ in ϕ_i as it is
22: **end if**
23: **end for**
24: **end for**
25: **end for**

– If the truth value of a literal p_i may vary then it is kept as it is in the negative guard g_{neg} of τ (Lines 20–21).

Note that since only dynamic elements can be modified therefore no further preprocessing is required for the actions of a guarded command. In other terms, a state variable array appearing in an action is guaranteed to be dynamic.

Example 10 After this preprocessing step, we expect to have the following SAL specification for the transitions of our sample automaton model (in Fig. 1) formalized by graph transformation rules of Fig. 2.

```
% guarded commands for initR and reachableR
TRANSITION
  NOT reachable[a1][s1]  -->
    reachable'[a1][s1] = TRUE; []% s1 is init
  reachable[a1][s1] AND
  NOT reachable [a1][s2] -->
    reachable'[a1][s2] = TRUE; []% s1 -> s2
  reachable[a1][s1] AND
  NOT reachable [a1][s3] -->
    reachable'[a1][s3] = TRUE; []% s1 -> s3
  reachable[a1][s2] AND
  NOT reachable [a1][s3] -->
    reachable'[a1][s3] = TRUE; % s2 -> s3
END;
```

Starting from the naive encoding of *initR* of Example 8, on the one hand, we truncate the guard for the first potential match by eliminating literals automaton$(a1)$, states$(a1, s1)$, state$(s1)$, etc. as they constantly evaluate to true. On the other hand, we eliminate the other two transitions since, init$(a1, s2)$ and init$(a1, s3)$ is constant false due to the static structure of the model.

Filtering properties Since only dynamic elements appear in the optimized version of a TS generated from a graph transformation system, the (safety or deadlock) property to be verified should also be modified by removing the static parts. If we assume that a property does not contain (neither quantified nor unquantified) variables only specific locations (i.e., $p(x)$ is not allowed if x is a variable only $p(v_i)$ where v_i is a specific value), then we can proceed in a similar way as done when truncating/eliminating guards.

Example 11 Consider the following property of finite automata, which expresses that state $s1$ of automaton $a1$ is not reachable.

```
NOT (automaton[a1]  AND state[s1] AND
     states[a1][s1] AND reachable[a1][s1])
```

As the static parts are satisfied in the initial model, and they are not modified during the evolution of the model, the filtered property (containing only dynamic elements) is as follows.

```
NOT (reachable[a1][s1])
```

6 Proof of Operational Equivalence

In the current section, we show on the ASM level that there is bisimulation between the original graph transformation system and the generated transition system, moreover, there is a one-to-one mapping between the states of the corresponding ASMs. We recommend that the reader should skip the current section if not interested in the formal correctness and completeness proof of our approach and continue with the case study of Sec. 7.

For the sake of simplicity, we split the equivalence proof into two.

– First we prove the equivalence of the ASM of a graph transformation system (ASM^{gr}) and the ASM of the naive encoding (ASM^{nv}) to show that our Cartesian product construction is appropriate (for unfolding graph transformation rules).
– Then we show that if we abstract from static parts of ASM^{nv} to obtain the optimized version of the target TS (ASM^{opt}), this equivalence still holds after eliminating certain (dead) transitions.

For the proofs, we use the traditional refinement scheme [9] given in the form of the commuting diagram of Fig. 4 (which is tailored to our current proof). For a given machine ASM^{Abs} which is refined to a machine ASM^{Ref}, we define a partial abstraction functions $\mathcal{F}$ to serve as a *proof maps*. The abstraction function $\mathcal{F}$ maps certain refined states $\mathfrak{A}^{Ref}$ of ASM^{Ref} to abstract states $\mathfrak{A}^{Abs} = \mathcal{F}(\mathfrak{A}^{Ref})$ of ASM^{Abs} and certain sequences R of Ref-rules to sequences $\mathcal{F}(R)$ of abstract ASM^{Abs}-rules. Since our proof is split into two, we will establish two proof maps $\mathcal{F}_1$ and $\mathcal{F}_2$ for each individual step.

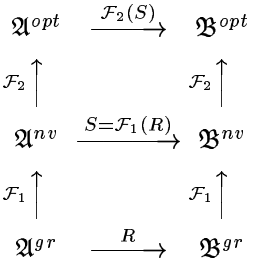

Fig. 4 ASM abstraction/refinement scheme

In order to establish the desired *operational equivalence* of the two machines, before proving the commutativity of the diagram, one can (and first of all has to) define the appropriate notions of *correctness* (i.e., each concrete computation implements an abstract computation) and/or *completeness* (i.e., each abstract computation is implemented by a concrete computation) between refined runs $(\mathfrak{B}; S)$ and abstract runs $(\mathfrak{A}; R)$. This definition is in terms of the locations (the "observables") one wants to compare in the related states of the two machines. The observables could be, for example, the operations the user sees in the abstract machine, which are implemented through the refinement step.

6.1 Correctness and completeness of the naive encoding

For the first proof, we show that the ASM representation ASM^{gr} of a graph transformation system is equivalent (in a certain sense) with the ASM representation ASM^{nv} of a TS generated according to the naive encoding (Alg. 2).

For a notational shorthand, we write $\mathfrak{A} = \mathfrak{B}$ if all locations are identical in states $\mathfrak{A}$ and $\mathfrak{B}$, i.e., for all function symbol p: $[\![p(x) = v]\!]^{\mathfrak{A}} \iff [\![p(x) = v]\!]^{\mathfrak{B}}$.

Proposition 1 (Equivalence of initial states) *The initial states $\mathfrak{A}^{gr}_{init}$ of ASM^{gr} and $\mathfrak{A}^{nv}_{init}$ of ASM^{nv} are equivalent with respect to $\mathcal{F}_1$. Formally, $\forall \mathfrak{A}^{gr}_{init} \exists \mathfrak{A}^{nv}_{init} : \mathfrak{A}^{nv}_{init} = \mathcal{F}_1(\mathfrak{A}^{gr}_{init})$, and $\forall \mathfrak{A}^{nv}_{init} \exists \mathfrak{A}^{gr}_{init} : \mathfrak{A}^{nv}_{init} = \mathcal{F}_1(\mathfrak{A}^{gr}_{init})$.*

Proof A trivial consequence of the construction in Sec. 5.2.

Proposition 2 *We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of ASM^{gr} and ASM^{nv} (provided that* nextIdFromReserve *is implemented correctly). Formally,*

1. ***Completeness / Forward simulation.***
   $\forall \mathfrak{A}^{gr} \forall \mathfrak{A}^{nv} \forall \mathfrak{B}^{gr} : \mathfrak{A}^{nv} = \mathcal{F}_1(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{gr} = next_R(\mathfrak{A}^{gr}) \rightarrow \exists \mathfrak{B}^{nv} : \mathfrak{B}^{nv} = next_{\mathcal{F}_1(R)}(\mathfrak{A}^{nv}) \wedge \mathfrak{B}^{nv} = \mathcal{F}_1(\mathfrak{B}^{gr})$.
2. ***Correctness / Backward simulation.***
   $\forall \mathfrak{A}^{gr} \forall \mathfrak{A}^{nv} \forall \mathfrak{B}^{nv} : \mathfrak{A}^{nv} = \mathcal{F}_1(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{nv} = next_{\mathcal{F}_1(R)}(\mathfrak{A}^{nv}) \rightarrow \exists \mathfrak{B}^{gr} : \mathfrak{B}^{gr} = next_R(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{nv} = \mathcal{F}_1(\mathfrak{B}^{gr})$.

Proof (Sketch) Each direction is proved separately.

Completeness Let r^{gr} be an enabled rule in ASM^{gr}, and R^{nv} be the set of transitions in ASM^{nv} generated from r^{gr} by Alg. 2.

Let $\zeta_{lhs} = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ denote a variable assignment for a successful matching of the rule r^{gr} (i.e., which respects the negative conditions as well). Since $\langle a_1, \ldots, a_n \rangle$ is an element from $dom(x_1) \times \ldots \times dom(x_n)$, thus we executed the loop of Line 2 in Alg. 2 to obtain a transition $\tau \in R^{nv}$ that is enabled exactly for the matching ζ_{lhs}.

Due to the encoding (Lines 9–11 in Alg. 2), the updates of τ are identical with the updates of r^{gr} provided that the same locations are accessed. Since ζ_{lhs} is already defined, this only leaves us to prove the non-deterministic creation of new objects (see Line 16 of Alg. 1) is correctly implemented by Lines 6–8 of Alg. 2, which is assumed here for *nextIdFromReserve*.

Since $\mathfrak{A}^{nv} = \mathcal{F}_1(\mathfrak{A}^{gr})$ by assumption, and the update set is identical, this finishes the proof of completeness.

Correctness Now let r^{nv} be an enabled transition (or, to be precise, rule) in ASM^{nv}, and we have to show that it corresponds to a successful application of a rule r^{gr} in ASM^{gr} for some matching ζ_{lhs}.

Since r^{nv} is enabled therefore all locations $\langle a_1, \ldots, a_n \rangle$ appearing in its guard g_{lhs} should be true (moreover, the negative guard g_{neg} should be satisfied as well). As $\mathfrak{A}^{nv} = \mathcal{F}_1(\mathfrak{A}^{gr})$ holds by assumption, therefore the truth values of the same locations in ASM^{gr} are identical.

Due to the fact that r^{nv} was generated by Alg. 2, there exists a rule r^{gr} in ASM^{gr} from which it was derived. However, notice that the variable assignment $\zeta_{lhs} := \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ satisfies $\phi_{lhs}(x_1, \ldots, x_n)$ (while ϕ_{neg} is also respected), therefore, rule r^{gr} is enabled as well. Moreover, since g^{nv} was generated from r^{gr} the updates are identical as well (as we already proved for completeness).

Applying the rule on the same matching with the same update set implies that $\mathfrak{B}^{nv} = \mathcal{F}_1(next_{r^{gr}}(\mathfrak{A}^{gr}))$. □

6.2 Correctness and completeness of the optimized encoding

For the second part of the proof, we show that the ASM representation ASM^{nv} of a TS is equivalent (in a certain sense) with the ASM representation ASM^{opt} of a TS generated from ASM^{nv} according to the optimization method by eliminating dead transitions in Alg. 3.

Proposition 3 (Equivalence of initial states) *The initial states $\mathfrak{A}^{nv}_{init}$ of ASM^{nv} and $\mathfrak{A}^{opt}_{init}$ of ASM^{opt} are equivalent with respect to $\mathcal{F}_2$. Formally, $\forall \mathfrak{A}^{nv}_{init} \exists \mathfrak{A}^{opt}_{init} : \mathfrak{A}^{opt}_{init} = \mathcal{F}_2(\mathfrak{A}^{nv}_{init})$, and $\forall \mathfrak{A}^{opt}_{init} \exists \mathfrak{A}^{nv}_{init} : \mathfrak{A}^{opt}_{init} = \mathcal{F}_2(\mathfrak{A}^{nv}_{init})$.*

Proof (Sketch) Locations derived from dynamic elements are identical in $\mathfrak{A}^{nv}_{init}$ and $\mathfrak{A}^{opt}_{init}$ due to the construction of Sec. 5.3, while static elements of $\mathfrak{A}^{nv}_{init}$ are not projected into $\mathfrak{A}^{opt}_{init}$.

Proposition 4 *We can establish a bisimulation between the steps/runs and a bidirectional mapping between states of ASM^{nv} and ASM^{opt}. Formally,*

1. ***Completeness / Forward simulation.***
   $\forall \mathfrak{A}^{nv} \forall \mathfrak{A}^{opt} \forall \mathfrak{B}^{nv} : \mathfrak{A}^{opt} = \mathcal{F}_2(\mathfrak{A}^{nv}) \wedge \mathfrak{B}^{nv} = next_R(\mathfrak{A}^{nv}) \rightarrow \exists \mathfrak{B}^{opt} : \mathfrak{B}^{opt} = next_{\mathcal{F}_2(R)}(\mathfrak{A}^{opt}) \wedge \mathfrak{B}^{opt} = \mathcal{F}_2(\mathfrak{B}^{nv})$.
2. ***Correctness / Backward simulation.***
   $\forall \mathfrak{A}^{nv} \forall \mathfrak{A}^{opt} \forall \mathfrak{B}^{opt} : \mathfrak{A}^{opt} = \mathcal{F}_2(\mathfrak{A}^{nv}) \wedge \mathfrak{B}^{opt} = next_{\mathcal{F}_2(R)}(\mathfrak{A}^{opt}) \rightarrow \exists \mathfrak{B}^{nv} : \mathfrak{B}^{nv} = next_R(\mathfrak{A}^{nv}) \wedge \mathfrak{B}^{opt} = \mathcal{F}_2(\mathfrak{B}^{nv})$.

Proof (Sketch) Each direction is proved separately (using the notation of Alg. 3).

Completeness Let $\tau^{nv} = g^{nv}_{lhs} \wedge g^{nv}_{neg} \longrightarrow act^{nv}$ be an *enabled* transition (rule) in ASM^{nv}. Let $\tau^{opt} = g^{opt}_{lhs} \wedge g^{opt}_{neg} \longrightarrow act^{opt}$ be a transition in ASM^{opt} generated from τ^{nv} as a result of Alg. 3. We first show that τ^{opt} is enabled.

Since τ^{nv} is enabled, therefore all (positive) literals p_i in g^{nv}_{lhs} accessing locations in ASM^{nv} are evaluated to true in state $\mathfrak{A}^{nv}$ ($\forall i : [\![p_i]\!]^{\mathfrak{A}^{nv}} = true$). As a consequence, we can state the following.

- If p_i is static, then p_i does not appear in the guard g^{opt}_{lhs} of τ^{opt}.
- If p_i is dynamic, then p_i appears in the guard g^{opt}_{lhs}, but $[\![p_i]\!]^{\mathfrak{A}^{opt}} = true$.

Furthermore, as τ^{nv} is enabled, none of the (negated) clauses $\neg\phi_i$ in g^{nv}_{neg} are violated in state $\mathfrak{A}^{nv}$ (i.e., $\forall i : [\![\neg\phi_i]\!]^{\mathfrak{A}^{nv}} = true$). Therefore, for all negated clause ϕ_i there exists a literal $p_{i,j}$ for which $[\![p_{i,j}]\!]^{\mathfrak{A}^{nv}} = false$.

- If $p_{i,j}$ is static then ϕ_i does not appear in g^{opt}_{neg}.
- If $p_{i,j}$ is dynamic, then it appears in the guard g^{opt}_{neg}, but $[\![p_{i,j}]\!]^{\mathfrak{A}^{opt}} = false$ implying that $[\![\neg\phi_i]\!]^{\mathfrak{A}^{opt}} = true$.

Therefore τ^{opt} is enabled whenever τ^{nv} is enabled. As act^{nv} is identical with act^{opt}, the same locations are updated in both cases. Hence $\mathfrak{B}^{opt} = next_{\mathcal{F}_2(R)}(\mathfrak{A}^{opt}) = \mathcal{F}_2(\mathfrak{B}^{nv})$, which finishes the proof of completeness.

Correctness Let $\tau^{opt} = g^{opt}_{lhs} \wedge g^{opt}_{neg} \longrightarrow act^{opt}$ be an *enabled* transition in ASM^{opt} generated from some transition $\tau^{nv} = g^{nv}_{lhs} \wedge g^{nv}_{neg} \longrightarrow act^{nv}$ in ASM^{nv} as a result of Alg. 3. Again as $act^{opt} = act^{nv}$, we only have to show that τ^{nv} is enabled whenever τ^{opt} is enabled.

Since τ^{opt} is enabled, therefore all (positive) literals p_i in g^{opt}_{lhs} accessing locations in ASM^{opt} are evaluated to true in state $\mathfrak{A}^{opt}$ (i.e., $\forall i : [\![p_i]\!]^{\mathfrak{A}^{opt}} = true$). Let us now suppose by contradiction that τ^{nv} is not enabled because there exists a literal p_j for which $[\![p_j]\!]^{\mathfrak{A}^{nv}} = false$ in the original (naive) transition system.

- If p_j is static, then τ^{nv} would be eliminated (Lines 5–6 in Alg. 3), which contradicts the fact that τ^{opt} is generated from τ^{nv}.
- If p_j is dynamic, then p_j appears in the guard g^{opt}_{lhs}, but $[\![p_j]\!]^{\mathfrak{A}^{opt}} = true$ and $[\![p_j]\!]^{\mathfrak{A}^{nv}} = false$, which is a contradiction as well.

Furthermore, as τ^{opt} is enabled, none of the (negated) clauses $\neg\phi^{opt}_i$ in g^{opt}_{neg} are violated in state $\mathfrak{A}^{opt}$ (i.e., $\forall i : [\![\neg\phi^{opt}_i]\!]^{\mathfrak{A}^{opt}} = true$). Therefore, for all ϕ^{opt}_i there exists a literal p_{i,j_0} for which $[\![p_{i,j_0}]\!]^{\mathfrak{A}^{opt}} = false$.

Let us now suppose by contradiction that τ^{nv} is not enabled because there exists a clause $\phi^{nv}_{i_0}$ in the original (naive) transition system τ^{nv} for which $[\![\neg\phi^{nv}_{i_0}]\!]^{\mathfrak{A}^{nv}} = false$. This may happen only if $\forall j : [\![p_{i_0,j}]\!]^{\mathfrak{A}^{nv}} = true$ where $\phi^{nv}_{i_0} \equiv \bigwedge_j p_{i_0,j}$.

- Now if all $p_{i_0,j}$ is static, then τ^{nv} would be eliminated (Lines 14–15 in Alg. 3), which contradicts the fact that τ^{opt} is generated from τ^{nv}.
- If there is a dynamic literal $p_{i_0,j}$ in $\phi^{nv}_{i_0}$, then this litaral also appears in the guard g^{opt}_{neg} in the corresponding clause $\phi^{opt}_{i_0}$. This causes a contradiction, since there exists a literal p_{i_0,j_0} for which $[\![p_{i_0,j_0}]\!]^{\mathfrak{A}^{opt}} = false$ but $[\![p_{i_0,j_0}]\!]^{\mathfrak{A}^{nv}} = true$.

As a result, we have $\mathfrak{B}^{opt} = \mathcal{F}_2(\mathfrak{B}^{nv})$ where $\mathfrak{B}^{nv} = next_R(\mathfrak{A}^{nv})$, which completes the proof of correctness.

As a consequence of all the propositions, we established the following main theorem for the correctness and completeness of our encoding.

Theorem 1 *The initial states $\mathfrak{A}^{gr}_{init}$ of ASM^{gr} and $\mathfrak{A}^{opt}_{init}$ of ASM^{opt} are equivalent with respect to $\mathcal{F}$ where $\mathcal{F} = \mathcal{F}_2 \circ \mathcal{F}_1$. Formally, $\forall \mathfrak{A}^{gr}_{init} \exists \mathfrak{A}^{opt}_{init} : \mathfrak{A}^{opt}_{init} = \mathcal{F}(\mathfrak{A}^{gr}_{init})$, and $\forall \mathfrak{A}^{opt}_{init} \exists \mathfrak{A}^{gr}_{init} : \mathfrak{A}^{opt}_{init} = \mathcal{F}(\mathfrak{A}^{gr}_{init})$.*

Moreover, we can establish a bisimulation between the steps/runs and a bidirectional mapping between states of ASM^{gr} and ASM^{opt} with the same $\mathcal{F}$. Formally,

1. ***Completeness / Forward simulation.***
   $\forall \mathfrak{A}^{gr} \forall \mathfrak{A}^{opt} \forall \mathfrak{B}^{gr} : \mathfrak{A}^{opt} = \mathcal{F}(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{gr} = next_R(\mathfrak{A}^{gr}) \rightarrow \exists \mathfrak{B}^{opt} : \mathfrak{B}^{opt} = next_{\mathcal{F}(R)}(\mathfrak{A}^{opt}) \wedge \mathfrak{B}^{opt} = \mathcal{F}(\mathfrak{B}^{gr})$.
2. ***Correctness / Backward simulation.***
   $\forall \mathfrak{A}^{gr} \forall \mathfrak{A}^{opt} \forall \mathfrak{B}^{opt} : \mathfrak{A}^{opt} = \mathcal{F}(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{opt} = next_{\mathcal{F}(R)}(\mathfrak{A}^{opt}) \rightarrow \exists \mathfrak{B}^{gr} : \mathfrak{B}^{gr} = next_R(\mathfrak{A}^{gr}) \wedge \mathfrak{B}^{opt} = \mathcal{F}(\mathfrak{B}^{gr})$.

7 Dining Philosophers: A Case Study for Modeling and Verification

In the current section, our model checking framework is evaluated by a case study on the well-known problem of dining philosophers. Even though the problem itself is relatively simple from a modeling point of view, it frequently serves as a benchmark to assess the performance of verification tools.

For the case study, we will use only a single model checker tool, but the problem will be modeled and encoded into transition systems in different ways.

- First the dynamic behavior of dining philosophers will be captured by graph transformation rules, thus our transformation technique will be applied on the model-level.
- Then, the behavior of philosophers is formalized by UML statecharts and projected into transition systems according to a set of graph transformation rules providing formal semantics for statecharts on the meta-level.
- Finally, the UML statecharts description is projected directly (i.e., without the use of graph transformation) into transition system following the approach of [34].

By that case study, we try to assess the succinctness of the target transition systems (concerning the state space) in each case by increasing the number of philosophers, thus the different modeling formalisms will be judged from a *verification* point of view.

The problem of dining philosophers In the dining philosophers' problem, n philosophers are sitting around a table and thinking. From time to time, when they get hungry, they initiate an eating process by grabbing first a left fork and then a right fork. Unfortunately, there is a single fork between two philosophers (which is hence a shared resource) therefore they might need to wait for the forks to become available. When a philosopher manages to get both the left and the right fork, he (or she) starts eating immediately. As soon as the philosopher has finished eating he places back both forks and goes back to thinking.

From a verification point of view, our aim is to show that (started from a state where each philosopher is thinking and all forks are on the table) the system of dining philosophers will not reach a deadlock (deadlock freedom property), moreover, no forks are held at the same time by multiple philosophers (a safety criterion requiring mutual exclusion on forks).

7.1 Case 1: Graph transformation on the model level

The class diagram in Fig. 5 captures the static structure of dining philosophers with two static classes (Phil and Fork) with two static associations left and right (identifying the left and right fork of a philosopher), a dynamic association hold (for expressing that a philosopher holds a certain fork), and the dynamic attribute status of philosophers, which may take the value from the enumeration think, hungry, hasL (has only left fork), hasR (has only right fork) and eat.

For the presentation in the current paper (but not for verification), we work with three philosophers in all three cases sitting around in a way depicted in the object diagram of Fig. 5.

The behavior of dining philosophers is captured in this case by the set of graph transformation rules shown in Fig. 6.

- Rule *getHungryR* simply sets the status attribute of a thinking philosopher to hungry.

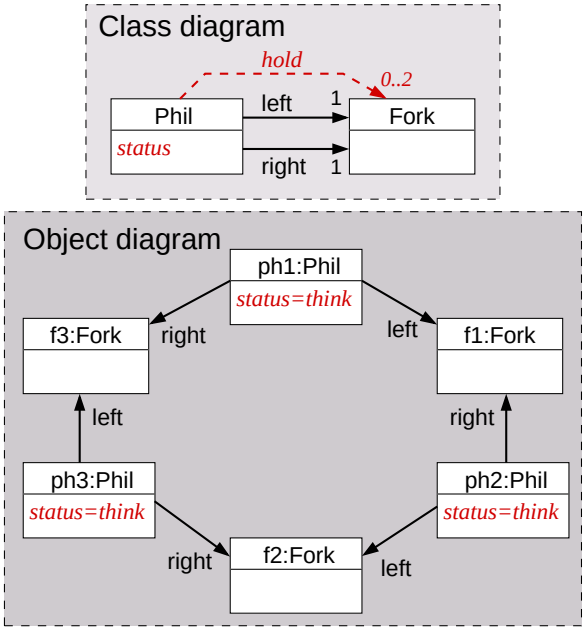

Fig. 5 Dining philosophers: class and object diagram

- A hungry philosopher may get his left fork by applying *getLeftForkR*, which requires that the left fork is not yet held as being the right fork of his/her right neighbor.
- A philosopher already having the left fork in hand may apply for the right fork by executing rule *getRightForkR* (which is conceptually similar to *getLeftFork*) and starts eating if succeeded.
- An eating philosopher will release his or her left fork at some point by applying rule *finishEatingR* getting into the status hasR in the meantime.
- Finally, if a philosopher only has the right fork in hand, then the fork can be safely released, and the philosopher goes back to thinking (see rule *releaseRightForkR*).

The corresponding transition system derived according to our encoding of Sec. 5 is listed in Appendix B.1.

Naturally, the verification process automatically detects that the system may get into a deadlock, if each philosopher only manages to grab his or her left fork and thus waits for the right fork forever. However, our safety property requiring that no fork is held by two philosopher at any time (formally, $\forall$ f: Fork, ph1, ph2: Phil: G (ph1 $\neq$ ph2 $\rightarrow \neg$ (hold[ph1][f] $\wedge$ hold[ph2][f]))) is verified.

The two traditional ways to avoid the deadlock problem for dining philosophers are depicted in Fig. 7.

- An additional graph transformation rule (*releaseLeftForkR*) can be introduced to move the system from a possibly deadlock situation by prescribing that if a philosopher already having a left fork cannot get the right fork as well then the left fork is put back onto the table and the philosopher goes back to hungry status.
- Alternatively, rules *getLeftForkR* and *getRightForkR* can be merged into a single rule *getBothForksR*, when both

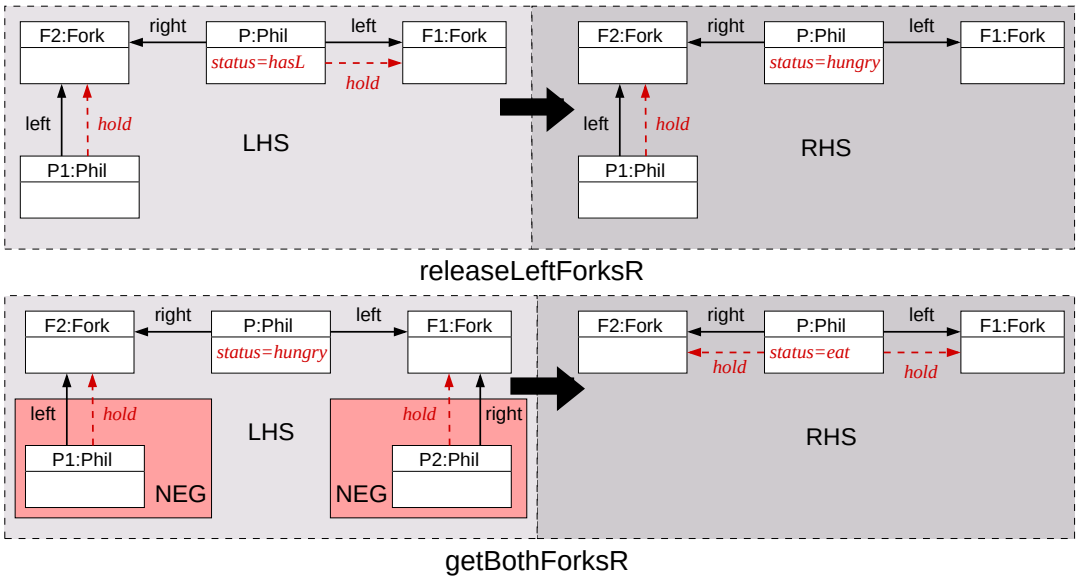

Fig. 7 Two ways to avoid deadlock for dining philosophers

left and right forks are grabbed at once thus preventing the philosopher to wait for a single fork.

The verification of the transition systems generated from these two versions proved that no deadlock situation is possible (while the safety criterion is still satisfied); however, there were be crucial differences in performance.

In a traditional graph transformation system, the first solution would typically supercede the other in performance as the application of a more complex rule takes much more time than formalizing the problem with *a set of* rules of smaller complexity (by complexity we mean the number of nodes in the Lhs and Neg graphs since the performance of pattern matching is the critical phase in rule application).

However, after all the preprocessing (collecting potential matchings) performed at compile time, it will turn out that having a small number of relatively complex rules yield a better performance for verification than a larger number of rules with relatively low complexity (see Table 2 for a comparison later in Sec. 7.3).

7.2 Case 2: Graph transformation on the meta-level

In our second case study, graph transformation is applied on the meta-level: the dining philosophers' problem is captured by means of UML Statecharts (see 8) on the model-level but the formal semantics of statecharts is formalized by graph transformations rules (as depicted in Fig. 10 and 11).

We applied the following restrictions on the traditional UML statecharts to achieve a solution that is comparable to the one obtained from model-level graph transformation rules.

– All the actions are considered to be send actions, and each transition may only contain a single send action as the effect.
– Send actions are addressed by role names appearing in the class diagram (as a statechart is a class-level specification mechanism).
– Guards may only contain a single ISIN(role:state) statement querying whether another statemachine accessible via the specific role stays currently in a certain state.

Behavior of philosophers expressed by UML Statecharts Therefore, a thinking philosopher may get into a hungry state at any time. After that, he checks whether his left fork is in the free state, and if so, sends an acquire (acq) message to the left fork and moves himself to hasLeft state. Next, the same procedure is done for acquiring the right fork getting the philosopher into the eating state. Finally, after eating, the forks are released one by one, by sending a rel message to the left fork and the right fork, respectively.

The statemachine of a fork simply contains two states (free and held) stating whether the fork is held by a philosopher or it is situated on the table. Transitions between them are triggered by the acq and rel messages, respectively.

Semantics of UML Statecharts by graph transformation To provide formal semantics for UML Statecharts by graph transformation systems, we have built on [49] where an extended hierarchical automaton formalism was used as the underlying structural representation for statecharts. However, for the current case study, the state hierarchy was flattened by collecting all state configurations and transitions that can be fired simultaneously at a preprocessing phase (as done in [34]), thus obtaining a flat finite automaton formalism (see the metamodel in Fig. 9) communicating with message passing to each other as a simplification.

– The static parts of the metamodel are the following:

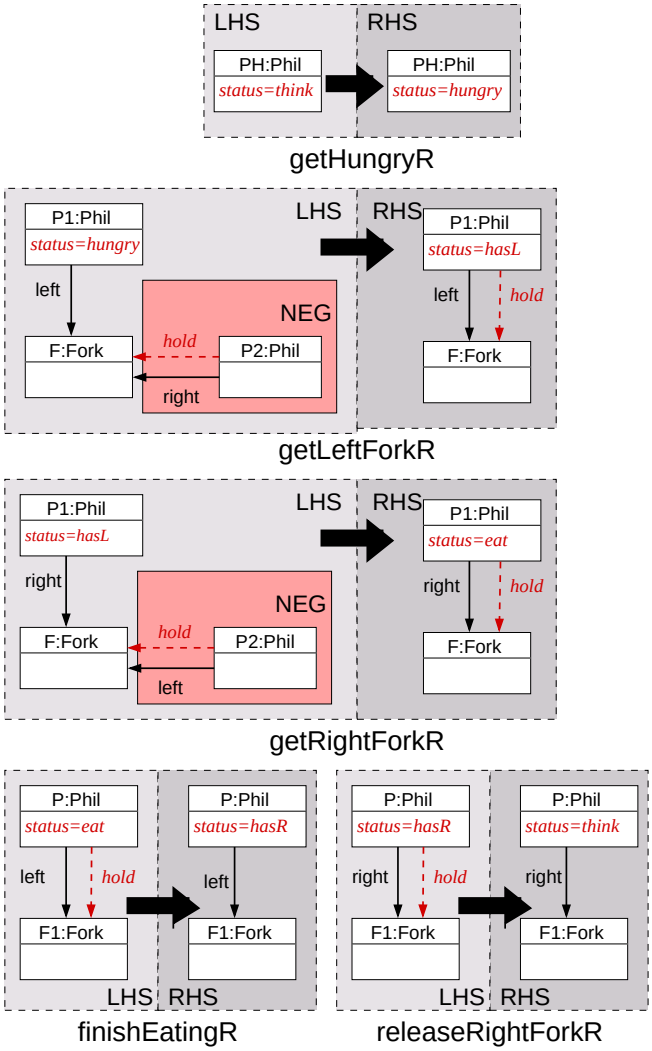

Fig. 6 Graph transformation rules for dining philosophers

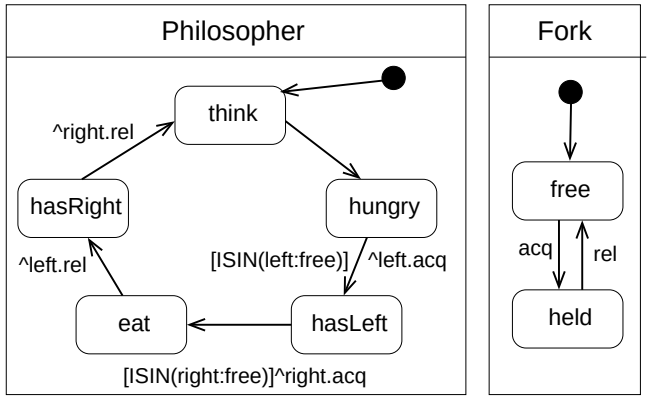

Fig. 8 Dining philosophers defined by UML Statecharts

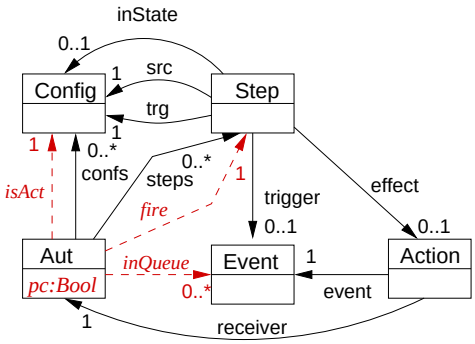

Fig. 9 A metamodel of flattened UML statecharts

- The automaton Aut consists of state configurations (Config) and steps (Step, which are the collections of transitions that can be fired at a time).
- Each step has a source src and a target trg configuration, an optional inState link to the configuration of another automaton (ISIN statement), a trigger Event, and an Action as effect.
- An Action contains a link to an Event to be sent and a receiver automaton.

– Dynamic parts of the metamodel are constituted as follows:
  - Each automaton may have an isAct link pointing to a configuration indicating that the automaton is currently in the certain configuration,
  - An automaton also has an inQueue link indicating whether a specific event is in the event queue of the automaton (event queues are modeled as sets, thus only one instance of the event may be stored in the queue).
  - The fire link leading to a Step states that the automaton is currently firing all the transitions in Step.
  - Finally, pc is a simple program counter indicating which phase of the statechart semantics is being executed for the moment by the virtual statemachine.

The graph transformation rules in Fig. 10 handle state (configuration) changes of a statemachine identifying four different cases split on the basis of existence of inState links and trigger events.

– In each case, the isAct link of the automaton A1 is rewritten from configuration S1 to configuration S2, supposing that the two configurations are connected by a step T1. Meanwhile, a fire link is added leading to T1 (since the step T1 is being fired currently), and the pc attribute of the automaton is updated to addQR to indicated that before processing the next event, the automaton must send the associated actions (to provide a synchronized behavior of the machine).
– In each four cases, there are additional conditions that are required for executing the current step of the statemachine (automaton).
  - Rule *fireNoEvtWithInStR* handles the case when that no trigger events are associated to the step (see the negative condition), however, the configuration identified by the inState link must be active in the corresponding automaton.

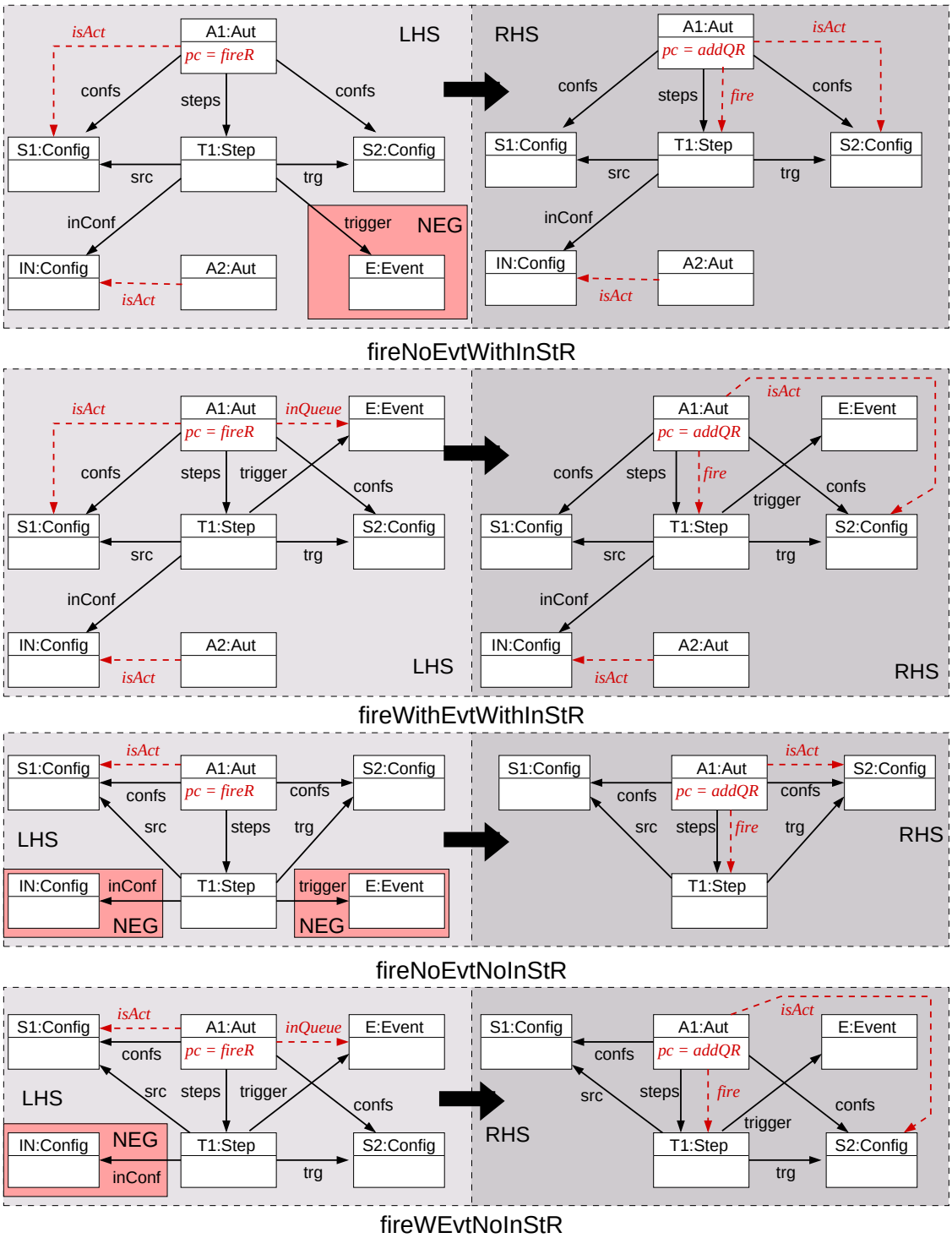

Fig. 10 Firing a transition (set) in a statechart automata

– Rule *fireWithEvtWithInStR* requires the trigger event of the step to be in the event queue of the automaton (the inQueue link), moreover, the configuration identified by the inState link must be active in the corresponding automaton. As an additional result, the event is removed from the event queue.
– Rule *fireNoEvtNoInStR* forbids the existence of both a trigger event and an inState link in the context of the step.
– Rule *fireWithEvtNoInStR*, finally, prescribes the non-existence of an inState link but requires that the trigger event of the step to be in the event queue of the automaton (inQueue link). As an additional result this

time as well, the event is removed from the event queue.

Message sending is modeled by graph transformation rules of Fig. 11.

- In case of rule *addQueueWithActR* there is a send action associated to the step, thus the event of the action should be placed in the event queue of the receiver automaton as a result of rule application. Note that the rule is only applicable if a certain step has already been selected to firing (see the fire link and the addQR value of the program counter).
- In case of rule *addQueueNoActR* there are no actions related to the firing step thus the rule simply concludes that the firing of the transition has terminated (thus, in both cases, the fire link is removed and the program counter is updated to the fireR).

In fact, this statechart semantics is probably oversimplified (not fully realistic) when compared to the original UML semantics, but the formalism had to be kept simple in order to provide some comparison with the other case studies from a verification point of view. For more precise ways of formalizing UML statecharts by graph transformation systems the reader is referred to [32, 49].

As graph transformation techniques were applied on the meta-level, the derived transition system (enlisted in Appendix B.2) is totally different in this case, since the state variable arrays are defined by the (meta)classes of the metamodel (in Fig. 9), and not by the classes of the model (Fig. 5).

Even though for meta-level encodings the verification managed to terminate only for few number of philosophers (as shown in Table 2), the verification process automatically detected a non-trivial error in the our graph transformation semantics of UML statecharts which leads our system into a state where our safety criterion is violated.

The problem originates from the fact that testing whether the fork is not held by anyone, and actually acquiring the fork is not an atomic operation. Moreover, the system may evolve between the sending of an acq message and the processing of the same message by the receiver since UML assumes a distributed environment for the execution of statemachines. Therefore, two philosophers may find the same fork to be free at one step and sending only the acquire message in the other. As a result, both philosophers move to a state when they think that they hold the specific fork, which is a contradiction with our safety requirement.

We can now conclude that (i) either the graph transformation rules presented as semantics for UML statecharts (in Fig. 10 and 11) are not appropriate, as the well-known system of dining philosophers behaves differently than what was expected (a *validation problem*), or (ii) if we accept that the transformation rules precisely capture the semantics of UML statecharts, then the UML specification of dining philosophers is not correct (a *verification problem*).

Model-level encoding of UML Statecharts In order to provide comparison with an existing approach for verifying UML statecharts, we adapted the SPIN encoding of UML statemachines described in [34] for other target model checkers. The main idea in the approach is to maintain only a single state variable for the state configuration information of each object and a state variable for its event queue in addition. Therefore, when a step is fired, the state changes and the message sendings are performed in a single and atomic operation (however, the synchronization of sending and receiving acquire messages had to be solved on the UML model level). The model-level (direct) encoding of UML statecharts representation of dining philosophers is listed in Appendix B.3.

7.3 Assessment of verification results

The three different kinds of specifications were executed by a model checker with different number of philosophers. Since the SAL model checker is not publicly available yet, for the concrete verification runs we used the Murϕ system to increase the comparability of results.

For our test experiments (summarized in Table 2), for each verification run at most 100 Megabytes of system memory was allocated to store the state space, and Murϕ was running on a 550 MHz Pentium III machine.

Table 2 Verification runs for the dining philosophers problem

Test case	N	States	Fired tr.	Time	Result
GT (err)	5	235	405	0.12	dead
GT (ok1)	3	75	201	0.15	ok
GT (ok1)	5	1363	6095	0.18	ok
GT (ok1)	9	439103	3535515	122.99	ok
GT (ok2)	3	20	48	0.10	ok
GT (ok2)	5	152	620	0.10	ok
GT (ok2)	12	172928	1695360	103.12	ok
GT/SC (err)	3	931	2387	0.15	unsafe
GT/SC (err)	5	6287	20519	0.49	unsafe
GT/SC (ok)	2	68084	236902	6.59	ok
SC (err)	3	743	1915	0.98	dead
SC (err)	5	128439	577570	12.52	dead
SC (ok)	3	7057	29289	1.43	ok
SC (ok)	4	138001	764936	16.24	dead

There were seven different specifications tested with an increasing number of philosophers. For each specification, the last line of the test cases contain the maximum number of philosophers for which the verification terminated within the given resources.

The columns of the table contain, respectively, (1) the identifier of the specification, (2) the number of philosophers, (3) the number of states traversed, (4) the number of transitions during verification, (5) an average execution time (in seconds), and (6) the result of the verification (where '*unsafe*' means that the safety criterion was violated, '*dead*' refers to the fact that a deadlock was detected, while '*ok*' means that no errors were found).

The different specifications are encoded as follows:

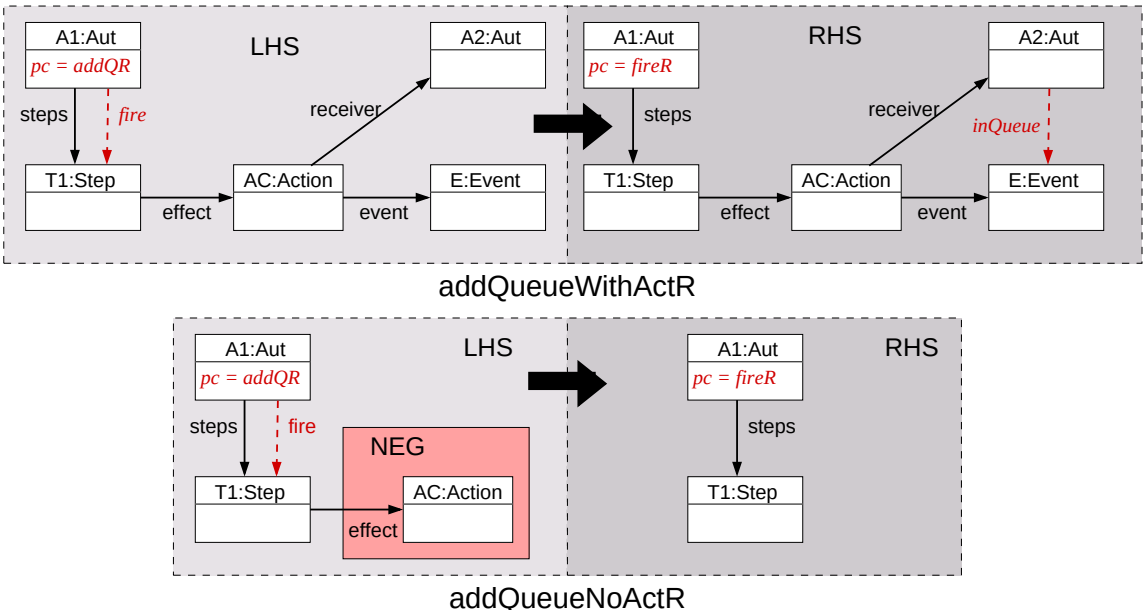

Fig. 11 Adding a send event to the target queue

- **GT (err)**: the model-level encoding of the dining philosophers problem using the original set of graph transformation rules of Fig. 6;
- **GT (ok1)**: the model-level encoding of the dining philosophers problem using a corrected set of graph transformation rules with rule *releaseLeftForkR* of Fig. 7;
- **GT (ok2)**: the model-level encoding of the dining philosophers problem using a corrected set of rules including rule *getBothForksR* of Fig. 7;
- **GT/SC (err)**: the meta-level encoding of the dining philosophers problem captured by the UML statecharts of Fig. 8 (with statechart semantics defined by graph transformation rules of Fig. 10 and 11);
- **GT/SC (ok)**: the same approach as before but corrected by certain changes on the statechart (not discussed in the paper in details);
- **SC (err)**: the direct (model-level) encoding of UML statecharts to Murϕ following the guidelines of [34]. This test set is identical to the case captured by statechart of Fig. 8.
- **SC (ok)**: the same as before but an additional transition was introduced leading back to hungry state from hasLeft to get out from a possibly deadlock situation (following the principles of rule *releaseLeftForkR* on the statechart model).

From these verification results, the following conclusions can be drawn.

Remark 1 Transition systems derived from graph transformation systems used as *model-level* specifications by Alg. 2 and 3 show very good performance in verification (comparable to manual encoding of the problem as a transition system).

We can also derive that using our encoding on the meta-level has certain practical limitations on the size of the model to be verified.

Remark 2 The verification of transition systems derived from graph transformation systems used as a *meta-level* specification technique would typically work for small models of the modeling language in question. However, meaningful specification flaws (either in the model or in the formal semantics) can frequently be detected even on such relatively small models.

Typically, when the semantics of a new modeling language is created, it is first tested on very small models, thus a large percentage of weaknesses could possibly be detected by our technique.

Moreover, our recent investigations show that SPIN provides better facilities to handle the interleaving of transitions (based on partial orderedness and the explicit use of queues for communication between processes) than Murϕ did in our case study.

Remark 3 Graph transformation systems with a small number of complex rules typically behaves better for verification than a graph transformation system with a larger number of relatively simple rules formalizing the same problem.

This statement is a result of the our compile time preprocessing step that collects all potential matches thus the target transition system used for verification does not have to execute complex queries on the model. Therefore, (unlike the case of running a simulation of a system in a traditional graph transformation tool!) a smaller set of complex rules would cause fewer interleavings of transitions resulting in better run-time performance.

Additional case studies These conclusions drawn from the dining philosophers problem was also supported by additional simple verification case studies that have been carried

out within the SAL environment by encoding UML statecharts into SAL specifications. In fact, the need for the optimizations described in Sec. 5 were triggered by unsuccessful preliminary verification attempts. The automatic transformation into SAL specifications (for this specific modeling language, namely, UML) was carried out within the VIATRA environment [53].

In another case study of our approach, we captured the operational semantics of Petri nets by graph transformation systems and translated them into the specification language of the Murϕ model checker. In case of bounded Petri nets (where the number of tokens in Petri nets has an *a priori* upper bound), our approach was directly applicable. We also exploited the use of predicate abstraction by abstracting away from the concrete number of tokens, which resulted in a semi-decision procedure for proving liveness and safety properties of Petri nets.

Recently, our technique was applied to carry out formal analysis of architectural styles [5, 6] in an early phase of design. For these investigations, we proved reachability properties (i.e., to decide whether certain target states are reachable or not from a given initial state) of a heavily dynamic GTS (with several dynamic classes and associations).

8 Conclusions and Future Work

In the paper, we presented an approach that is simultaneously applicable for encoding (i) well-formed models of arbitrary modeling languages with semantics defined by metamodeling techniques (abstract syntax) and graph transformation rules (operational semantics) and (ii) graph transformation systems (in the original model-level sense) for describing the dynamic behavior of user models into transition systems.

As a result, we are able to verify semantic properties (like safety, deadlock freedom, etc.) of any specific well-formed model instance of the language or the user model itself by model checking tools having a specification language based on transition systems. This way, traditional correctness results from the theory of graph transformation (like confluence [15]), which provide solutions for proving very general properties of a specific problem can be complemented by our technique to reason about problem-specific properties by existing model checker tools.

The feasibility of our approach was demonstrated on a well-known verification benchmark, i.e., by verifying deadlock freedom and a safety property for the dining philosophers' problem captured by graph transformation both on the model-level and the meta-level. However, note that our technique also allows to investigate liveness or reachability properties as demonstrated by additional case studies.

As a verification rule of thumb for graph transformation systems, we established the "few complex is better than many simple" principle when concerning the complexity (of the left-hand side and negative condition graphs) of rules.

After having carried out several benchmark experiments in different domains (with partially automated translations), we are currently building a tool [45] that is capable of automatically translating models of arbitrary visual modeling languages defined by metamodeling and graph transformation (and thus model-level specifications based on graph grammars as well) into the corresponding Promela specifications (which is the input language of the SPIN model checker).

However, further research is required to overcome the state explosion problem we experienced in *meta-level* encodings of models (and, naturally, in model-level encodings of complex IT systems). Our case study has clearly revealed that the bottleneck of the verification process is not merely the *number* of transitions generated by our algorithm, but rather the *interleaving* of different transitions (i.e., larger steps in the evolution of the model yield a smaller number of intermediate states).

The main problem originates from the fact that traditional confluency results of the graph transformation theory cannot be exploited during model checking. Since in many cases, the behavior of the system is required to be investigated only on a single path instead of all possible interleavings of rule applications due to an appropriate theorem or analysis technique (like, e.g., critical pair analysis [11, 29]). Unfortunately, existing model checkers typically do not provide any facilities for the user to control the model checking process in such a way. For instance, in meta-level encodings, certain intermediate states of the graph transformation system (like executing a step and sending the action in case of our statecharts semantics) should be transparent for the model checker (without considering interleavings of transitions for them).

As a consequence of this reduced controllability, control structures appearing in many existing graph transformation tools (like, for instance, in case of PROGRES [46] or VIATRA [53]) can only be encoded as data (thus in the state variables as was done in our Petri net case study), which rather increases the number of interleavings instead of drastically decreasing them.

A different line of research aims at adapting our encoding to verification tools using labeled transition systems (LTS) as the underlying mathematical formalism (where information is related to transitions in contrast to Kripke structures where it is stored in states) in order to investigate process algebras (like CSP [30], or CCS [35]). Here the practical goal is to consistency analysis of behavioral UML diagrams extended with graph transformation rules in the style of [24].

Further challenges involve the formal analysis of graph transformation systems with time [26] based upon model checkers like Kronos [17], which typically use timed automata [2] as their specification language. As a potential result, we aim at investigating real-time systems specified in a high-level visual notation.

Acknowledgment

I would like to thank to András Pataricza (Budapest Univ. of Technology), John Rushby and many of his colleagues (at SRI International) for their encouragement and support. I am

also very much grateful to Luciano Baresi (Politecnico Milano) for reading the earlier versions of the paper and suggesting valuable improvements. I also very much appreciate the valuable comments of the anonymous reviewers of the paper.

References

1. *The Murϕ Model Checker*. `http://verify.stanford.edu/dill/murphi.html`.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, vol. 126:pp. 183–235, 1994.
3. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K. G. Larsen and M. Nielsen (eds.), *CONCUR 2001 - Concurrency Theory, 12th International Conference*, vol. 2154 of *LNCS*, pp. 381–395. Springer, Aalborg, Denmark, 2001.
4. P. Baldan and B. König. Approximating the behaviour of 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. 14–29. Springer, Barcelona, Spain, 2002.
5. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles. In P. Inverardi and J. Paakki (eds.), *Proc ESEC 2003: 9th European Software Engineering Conference*, pp. 68–77. ACM Press, Helsinki, Finland, 2003.
6. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles based on graph transformation. In I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau (eds.), *The 6th ICSE Workshop on Component Based Software Engineering: Automated Reasoning and Prediction*, pp. 67–72. Carnegie Mellon University, USA, and Monash University, Australia, Portland, Oregon, USA, 2003.
7. 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.
8. E. Börger and J.Schmid. Composition and submachine concepts for sequential asms. In P. Clote and H. Schwichtenberg (eds.), *Computer Science Logic (Gurevich Festschrift). Proc. 14th International Workshop CSL*, no. 1862 in LNCS, pp. 41–60. 2000.
9. E. Börger and R. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
10. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans, S. Kent, and B. Selic (eds.), *Proc. 2000 - Third International Conference on The Unified Modeling Language. Advancing the Standard.*, vol. 1939 of *LNCS*, pp. 294–308. Springer, York, UK, 2000.
11. P. Bottoni, A. Schürr, and G. Taentzer. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. Tech. rep., University of Rome, 2000.
12. E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, vol. 19(1):pp. 7–34, 2001.
13. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Tech. Rep. CSE-TR-423-00, 2000.
14. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, vol. 26(3/4):pp. 241–265, 1996.
15. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *In [43]*, chap. Algebraic Approaches to Graph Transformation — Part I: Basic Concepts and Double Pushout Approach, pp. 163–245. World Scientific, 1997.
16. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, 1995.
17. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III: Verification and Control*, vol. 1066, pp. 208–219. Springer, Rutgers University, New Brunswick, NJ, USA, 1995.
18. J. de Lara and H. Vangheluwe. Computer aided multi-paradigm modelling to process Petri nets and statecharts. 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. 239–253. Springer-Verlag, Barcelona, Spain, 2002.
19. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, vol. 223 of *IFIP Conference Proceedings*, pp. 435–447. Kluwer Academic Publishers, 2002.
20. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools. World Scientific, 1999.
21. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [43]*, chap. Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pp. 247–312. World Scientific, 1997.
22. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard*, vol. 1939 of *LNCS*, pp. 323–337. Springer, 2000.
23. 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.
24. 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.
25. Y. Gurevich. *Specification and Validation Methods*, chap. Evolving Algebras 1993: Lipari Guide. Oxford University Press, 1995.
26. S. Gyapay, R. Heckel, and D. Varró. Graph transformation with time: Causality and logical clocks. 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. 120–134. Springer-Verlag, Barcelona, Spain, 2002.
27. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proc. FASE: Fundamental*

Approaches to Software Engineering, vol. 1382 of *LNCS*, pp. 138–153. Springer, 1998.
28. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Integrating the specification techniques of graph transformation and temporal logic. In *Proc. Mathematical Foundations of Computer Science (MFCS'97), Bratislava*, vol. 1295 of *LNCS*, pp. 219–228. Springer, 1997.
29. 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.
30. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
31. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23(5):pp. 279–295, 1997.
32. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn (eds.), *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 241–256. Springer, 2001.
33. L. Lamport. What good is temporal logic. In R. E. A. Mason (ed.), *Proc. of the IFIP Congress*, pp. 657–668. North Holland, 1983.
34. 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.
35. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1995.
36. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, Limerick, Ireland, 2000.
37. Object Management Group. *Meta Object Facility Version 1.3*, 1999. `http://www.omg.org`.
38. J. Padberg and B. J. Enders. Rule invariants in graph transformation systems for analyzing safety-critical systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: Firs International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 334–350. Springer, Barcelona, Spain, 2002.
39. I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In R. J. Hall and E. Tyugu (eds.), *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
40. 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.
41. D. Peled. *Software Reliability Methods*. Springer, 2001.
42. A. Rensink. Model checking graph grammars. In M. Leuschel, S. Gruner, and S. Lo Presti (eds.), *Proc. of the 3rd Workshop on Automated Verification of Critical Systems (AVOCS 2003)*, Technical Report DSSE–TR–03–2, pp. 150–160. University of Southampton, 2003.
43. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
44. H. Saïdi. Model checking guided abstraction and analysis. In J. Palsberg (ed.), *Seventh International Static Analysis Symposium (SAS'00)*, vol. 1824 of *LNCS*, pp. 377–339. Springer-Verlag, Santa Barbara, CA, 2000. `http://www.sdl.sri.com/papers/saidi_sas00/`.
45. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In P. Stevens, J. Whittle, and G. Booch (eds.), *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, vol. 2863 of *LNCS*, pp. 92–95. Springer, San Francisco, CA, USA, 2003.
46. A. Schürr, A. J. Winter, and A. Zündorf. *In [20]*, chap. The PROGRES Approach: Language and Environment, pp. 487–550. World Scientific, 1999.
47. J. Sprinkle and G. Karsai. Defining a basis for metamodel driven model migration. In *Proceedings of 9th Annual IEEE Internation Conference and Workshop on the Engineering of Computer-Based Systems, Lund, Sweden*. 2002.
48. D. Varró. Automatic program generation for and by model transformation systems. In H.-J. Kreowski and P. Knirsch (eds.), *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pp. 161–173. Grenoble, France, 2002.
49. 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.
50. D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas (eds.), *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, vol. 72 of *ENTCS*, pp. 57–70. Elsevier, Barcelona, Spain, 2002.
51. D. Varró. *Automated Model Transformations for the Analysis of IT Systems*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003. Submitted.
52. 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 International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 18–33. Springer-Verlag, Dresden, Germany, 2002.
53. 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.

Dániel Varró was graduated at the Budapest University of Technology and Economics, and currently, he is an assistant lecturer at the Department of Measurement and Information Systems. His main research interest is to design automated and provenly correct model transformations within and between visual modeling languages with a special focus on UML. He was a former visiting researcher at SRI International and at the University of Paderborn.

A An introduction to abstract state machines

The current section provides a brief, semi-formal overview on abstract state machnies. For a more formal introduction of ASMs, the reader is referred to [8].

The main structural element of an ASM is a *state* $\mathfrak{A}$ (i.e., an arbitrary algebra) over a vocabulary (or signature) Σ consisting of a non-empty set denoted as the *superuniverse* and the *interpretations* of function symbols.

– The superuniverse $|\mathfrak{A}|$ is often divided into (i) subdomains $D_1, \ldots, D_m$, where all $D_i \subseteq |\mathfrak{A}|$ and (ii) a *reserve* Res, which is a set of elements that are currently not in use (i.e., reserved for entities to be created later).
– The interpretation of an ASM in a given state $\mathfrak{A}$ depends on the environment, i.e., the interpretation ζ that assigns a value to each free variable. We use the standard interpretation of terms $[\![t]\!]^{\mathfrak{A}}_{\zeta}$ and formulae $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta}$ in state $\mathfrak{A}$ under variable interpretation ζ (but we often suppress mentioning the underlying interpretation of variables for the sake of simplicity).
– Each *function symbol* has a *name* and an *arity*, and informally, they can be regarded as *n*-dimensional arrays in a transition system, however, with potentially infinite index and value domains (if the superuniverse itself is infinite).

Example 12 The basic ASM concepts are illustrated on Boolean algebras.

– The vocabulary Σ_{Bool} of Boolean algebras contains two nullary function symbols (constants) 0 and 1, a unary function symbols '–', and two binary function symbols '+' and '*'.
– We may define a state $\mathfrak{A}$ for the vocabulary Σ_{Bool} as follows. The superuniverse $|\mathfrak{A}|$ of the state $\mathfrak{A}$ is the set '{0,1}'. The functions are interpreted as follow, where a and b are either '0' or '1'.

$0^{\mathfrak{A}}$	:=	0	(zero)
$1^{\mathfrak{A}}$	:=	1	(one)
$-^{\mathfrak{A}} a$	:=	$1 - a$	(negation)
$a +^{\mathfrak{A}} b$	:=	$max(a, b)$	(logical OR)
$a *^{\mathfrak{A}} b$	:=	$min(a, b)$	(logical AND)

– The following are terms of the vocabulary Σ_{Bool}: $+(v_0, v_1)$, $+(1, *(v_7, 0))$. They are usually written as $v_0 + v_1$ and $1 + (v_7 * 0)$.
– Consider the state $\mathfrak{A}$ for Σ_{Bool}. Let ζ be a variable assignment with $\zeta(v_0) = 0$, $\zeta(v_1) = 1$ and $\zeta(v_2) = 1$. Then we have: $[\![(v_0 + v_1) * 2]\!]^{\mathfrak{A}}_{\zeta} = 1$.

Rules that define the dynamic behavior of ASMs are built up from *function updates* and built-in constructs for *skip*, *if-then-else*, *let*, *forall*, and *iteration* by *parallel* and *sequential composition*. Furthermore, we consider the *choose* construct as a special notation for using non-deterministic choice functions, and the *create* construct to assign a fresh element from the reserve. The informal semantics of ASM rules is summarized in Table 3.

Formally, the semantics of standard ASMs (i.e., how a step of an ASM should be executed) is defined in [25] by assigning to each rule R an *update set* $[\![R]\!]^{\mathfrak{A}}_{\zeta}$ (where state $\mathfrak{A}$ is the current state with interpretation ζ) which – if consistent – is fired in state $\mathfrak{A}$ and produces the next state $\mathfrak{B} = \mathit{next}(\mathfrak{A}, \zeta)$.

An update set is a set of *updates*, i.e., a set of pairs (loc, val) where loc is a location, and val is an element in the domain of $\mathfrak{A}$ to which the location is intended to be updated. Informally, a *location* is a store in an array accessed via the index such as $a[i]$. Formally, it is an n-ary function name f with a sequence of length n of elements in the domain of $\mathfrak{A}$, denoted by $f\langle a_1, \ldots, a_n\rangle$.

An update set U is called *inconsistent*, if u contains at least two pairs (loc, v_1) and (loc, v_2) with $v_1 \neq v_2$, otherwise it is called *consistent*.

Example 13 (Consistency of update sets) Let $curr$ be a unary function symbol. The update set U consisting of three updates $\{(curr\langle s1\rangle, \top),\ (curr\langle s2\rangle, \bot),\ (curr\langle s1\rangle, \bot)\}$ is inconsistent since two different values ($\top, \bot$) are assigned to the same location $curr\langle s1\rangle$.

The update set $\{(curr\langle s1\rangle, \top),\ (curr\langle s2\rangle, \bot),\ (curr\langle s1\rangle, \top)\}$ is consistent. Although it contains two updates for location $curr\langle s1\rangle$, the values of these updates are identically $\top$.

For a consistent update set U and a state $\mathfrak{A}$, the state $\mathit{fire}_{\mathfrak{A}}(U)$, resulting from firing U in $\mathfrak{A}$ is defined as a state $\mathfrak{B}$ which coincides with $\mathfrak{A}$ except for locations appearing in the update set, formally, it assigns $f^{\mathfrak{B}}(a) = val$ for each $(f\langle a\rangle, val) \in U$. Firing an inconsistent update set is not allowed, i.e., $\mathit{fire}_{\mathfrak{A}}(U)$ is not defined for inconsistent U. This definition yields the (partial) next state function $\mathit{next}_R(\mathfrak{A}, \zeta) = \mathit{fire}([\![R]\!]^{\mathfrak{A}}_{\zeta})$ which describes one application of R in a state with a given environment function ζ. Often we also write $\mathit{next}(R)$ instead of next_R.

$[\![x]\!]^{\mathfrak{A}}_{\zeta}$	Variable assignment ($\zeta(X)$)
$[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta}$	Interpretation of functions ($f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_n]\!]^{\mathfrak{A}}_{\zeta})$)
skip	Do nothing
$f(t_1, \ldots, t_n) := s$	In the next state, the value of the function f at the arguments $t_1, \ldots, t_n$ (i.e., at location $f\langle t_1, \ldots, t_n\rangle$) is updated to s.
R **par** $S \equiv R\ S$	Rules R and S are executed in parallel.
if φ **then** R **else** S	If φ is true, then execute R, otherwise execute S.
let $x = t$ **in** R	Assign the value of t to x and execute R.
forall x **with** φ **do** R	Execute R in parallel for each x satisfying φ.
R **seq** $S \equiv R$ **;** S	Execute first R and then S as an "atomic" sequence.
choose(x) **with** φ **do** R	Select non-deterministically an element x satisfying condition φ and execute R
create(x) **do** R	Create a new element x from the "reserve" which does not belong to any of the existing domains.
$r(t_1, \ldots, t_n)$	Call a defined rule

Table 3 Informal semantics of transition rules in ASMs

Example 14 (Rules of traffic lights) To demonstrate the dynamic behavior of ASMs, let the vocabulary Σ_{traf} of a (Hungarian) traffic light consist of three nullary dynamic functions $\{red, yellow, green\}$. Initially, in state $\mathfrak{A}$, let us define the following evaluation $[\![red]\!]^{\mathfrak{A}} = \top$, $[\![yellow]\!]^{\mathfrak{A}} = \bot$, and $[\![green]\!]^{\mathfrak{A}} = \bot$.

1. The rule **if** red **then** $yellow := \top$ should switch on the yellow light in addition to the red.
2. The rule **if** $red \wedge yellow$ **then** $red := \bot; yellow := \bot;$ $green := \top$ should switch on green and switch off the other two lights.
3. The rule **if** $green$ **then** $green := \bot; yellow := \top$ should switch off green and switch on yellow.
4. The rule **if** $\neg red \wedge yellow$ **then** $red := \top; yellow := \bot$ switch on red and switch off the yellow light.

When applying Rule 1 in state $\mathfrak{A}$, the update set U is $\{(yellow, \top)\}$ the next state $\mathfrak{B}$ is $[\![red]\!]^{\mathfrak{B}} = \top$, $[\![yellow]\!]^{\mathfrak{B}} = \top$, and $[\![green]\!]^{\mathfrak{B}} = \bot$.

Now if we apply Rule 2 in state $\mathfrak{B}$ the update set U is $\{(red, \bot), (yellow, \bot), (green, \top)\}$ the next state $\mathfrak{C}$ is $[\![red]\!]^{\mathfrak{C}} = \bot$, $[\![yellow]\!]^{\mathfrak{C}} = \bot$, and $[\![green]\!]^{\mathfrak{C}} = \top$.

B SAL Descriptions for Dining Philosophers

B.1 Model-Level Encoding of Dining Philosophers

Note that only the behavior of philosopher ph1 is depicted there due to the symmetric nature of the problem to improve clarity.

```
ForkID : TYPE = {f1,f2,f3};
PhilID : TYPE = {ph1,ph2,ph3};
StatusRng : TYPE = {think, hungry,
                    hasL, eat, hasR};
dinphil_gt : MODULE =
BEGIN
 GLOBAL hold:   ARRAY PhilID OF
                ARRAY ForkID OF BOOLEAN
 GLOBAL status: ARRAY PhilID OF StatusRng
INITIALIZATION
  status[ph1] = think;
  hold[ph1][f1] = FALSE;
  hold[ph1][f2] = FALSE;
  hold[ph1][f3] = FALSE;
TRANSITION
  status[ph1] = think  -->
    status[ph1] = hungry; []
  status[ph1] = hungry AND
  NOT hold[ph2][f1] -->
    hold[ph1][f1] = TRUE;
    status[ph1] = hasL; []
  status[ph1] = hasL AND
  NOT hold[ph3][f3] -->
    hold[ph1][f3] = TRUE;
    status[ph1] = eat; []
  status[ph1] = eat -->
    hold[ph1][f1] = FALSE;
    status[ph1] = hasR; []
  status[ph1] = hasR -->
    hold[ph1][f3] = FALSE;
    status[ph1] = think;
END;
```

B.2 Meta-Level Encoding of Dining Philosophers

The meta-level encoding of a single dining philosopher is listed below in the SAL format.

```
Conf_dom:  TYPE = {think, hungry, hasLeft,
                   hasRight,eat,free,held};
Step_dom:  TYPE = {t1,t2,t3,t4,t5,t6,t7};
Event_dom: TYPE = {acq, rel};
Aut_dom  : TYPE = {ph1,ph2,ph3,f1,f2,f3};
pc_rng   : TYPE = {fireR, addQR};

dinphil_sc_gt : MODULE =
BEGIN
 GLOBAL isAct  :ARRAY [Aut_dom] OF
                ARRAY [Conf_dom] OF BOOLEAN;
 GLOBAL inQueue:ARRAY [Aut_dom] OF
                ARRAY [Event_dom] OF BOOLEAN;
 GLOBAL fire   :ARRAY [Aut_dom] OF
                ARRAY [Step_dom] OF BOOLEAN;
 GLOBAL pc     :ARRAY [Aut_dom] OF pc_rng;
INITIALIZATION
 isAct[ph1][think] = TRUE;
 isAct[ph1][hungry] = FALSE;
 isAct[ph1][hasLeft] = FALSE;
 isAct[ph1][eat] = FALSE;
 isAct[ph1][hasRight] = FALSE;
 isAct[ph1][free] = FALSE;
 isAct[ph1][held] = FALSE;
 inQueue[ph1][acq] = FALSE;
 inQueue[ph1][rel] = FALSE;
 fire[ph1][t1] = FALSE;
 fire[ph1][t2] = FALSE;
 fire[ph1][t3] = FALSE;
 fire[ph1][t4] = FALSE;
 fire[ph1][t5] = FALSE;
 fire[ph1][t6] = FALSE;
 fire[ph1][t6] = FALSE;
 pc[ph1] = fireR;
TRANSITION
% fireNoEvtNoInStR
 pc[ph1] = fireR AND isAct[ph1][think]) AND
 NOT fire[ph1][t1] -->
   isAct[ph1][think] = FALSE;
   isAct[ph1][hungry] := TRUE;
   pc[ph1] := addQR;
   fire[ph1][t1] := TRUE; []
% fireNoEvtWithInStR
 pc[ph1] = fireR AND isAct[ph1][hungry]) AND
 isAct[f1][free] AND NOT fire[ph1][t2] -->
   isAct[ph1][hungry] := FALSE;
   isAct[ph1][hasLeft] = TRUE;
   pc[ph1] := addQR;
   fire[ph1][t2] := TRUE; []
% fireNoEvtWithInStR
 pc[ph1] = fireR AND isAct[ph1][hasLeft]) AND
```

```
 isAct[f3][free] AND NOT fire[ph1][t3] -->
   isAct[ph1][hasLeft] := FALSE;
   isAct[ph1][eat] = TRUE;
   pc[ph1] := addQR;
   fire[ph1][t3] := TRUE; []
% fireNoEvtNoInStR
 pc[ph1] = fireR AND  isAct[ph1][eat]) AND
 NOT fire[ph1][t4] -->
   isAct[ph1][eat] := FALSE;
   isAct[ph1][hasRight] = TRUE;
   pc[ph1] := addQR;
   fire[ph1][t4] := TRUE; []
% fireNoEvtNoInStR
 pc[ph1] = fireR AND  isAct[ph1][eat]) AND
 NOT fire[ph1][t5] -->
   isAct[ph1][hasRight] := FALSE;
   isAct[ph1][think] = TRUE;
   pc[ph1] := addQR;
   fire[ph1][t5] := TRUE; []
% addQueueNoActR
 pc[ph1] = addQR AND fire[ph1][t1] -->
   pc[ph1] := fireR;
   fire[ph1][t1] := FALSE; []
% addQueueWithActR
 pc[ph1] = addQR AND fire[ph1][t2] -->
   pc[ph1] := fireR;
   inQueue[f1][acq] := TRUE;
   fire[ph1][t2] := FALSE; []
% addQueueWithActR
 pc[ph1] = addQR AND fire[ph1][t3] -->
   pc[ph1] := fireR;
   inQueue[f3][acq] := TRUE;
   fire[ph1][t3] := FALSE; []
% addQueueWithActR
 pc[ph1] = addQR AND fire[ph1][t4] -->
   pc[ph1] := fireR;
   inQueue[f1][rel] := TRUE;
   fire[ph1][t4] := FALSE; []
% addQueueWithActR
 pc[ph1] = addQR AND fire[ph1][t5] -->
   pc[ph1] := fireR;
   inQueue[f3][rel] := TRUE;
   fire[ph1][t5] := FALSE; []
END;
```

B.3 Model-level Encoding of the UML Statecharts for Dining Philosophers

This description was yielded by adapting the techniques described in [34] to the SAL and Murϕ systems. Only the transitions of a single philosopher are included this time as well.

```
ForkID :    TYPE = {f1,f2,f3};
PhilID :    TYPE = {ph1,ph2,ph3};
Event_dom : TYPE = {acq, rel};
fork_status_rng : TYPE = {free, held};
phil_status_rng : TYPE = {hungry, think,
                          hasL, hasR, eat};
dinphil_sc : MODULE =
BEGIN
 hold:     ARRAY [PhilID] OF
           ARRAY [ForkID] OF boolean;
 ph_status:ARRAY [PhilID] OF phil_status_rng;
 f_status: ARRAY [ForkID] OF fork_status_rng;
 f_queue:  ARRAY [ForkID ] OF
           ARRAY [Event_dom] OF boolean;
INITIALIZATION
  hold[ph1][f1] = FALSE;
  hold[ph1][f2] = FALSE;
  hold[ph1[f3]= FALSE;
  ph_status[ph1] = think; ...
TRANSITION
  ph_status[ph1] = think  -->
    ph_status[ph1] = hungry; []
  ph_status[ph1] = hungry AND
  f_status[f1] = free
  AND NOT hold[ph2][f1] -->
    hold[ph1][f1] = TRUE;
    f_queue[f1][acq] := true;
    ph_status[ph1] = hasL; []
  ph_status[ph1] = hasL AND
  f_status[f3] = free
  AND NOT hold[ph3][f3] -->
    hold[ph1][f3] = TRUE;
    f_queue[f3][acq] := true;
    ph_status[ph1] = eat; []
  status[ph1] = eat -->
    hold[ph1][f1] = FALSE;
    f_queue[f1][rel] := true;
    status[ph1] = hasR; []
  status[ph1] = hasR -->
    hold[ph1][f3] = FALSE;
    f_queue[f3][rel] := true;
    status[ph1] = think;
END;
```