A Formal Semantics of UML Statecharts by Model Transition Systems [⋆]

Dániel Varró

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

Abstract. UML Statecharts are well-known visual means to capture the dynamic behavior of reactive systems in the object-oriented design methodology. Since the UML standard only contains an informal description on how to execute such statemachines mathematically precise semantic frameworks are required for an automated analysis. The current paper presents a formal semantics for UML statecharts based on a combination of metamodeling and graph transformation that is (i) simultaneously visual and precise, and (ii) clearly separates derived static concepts (like priorities, conflicts, etc.) from their dynamic interpretation thus scaling up well for different statechart variants (with, e.g., various priority strategies) and potential future changes in the standard.
Keywords: UML Statecharts, graph transformation, model transition systems, metamodeling

1 Introduction

For the recent years, the Unified Modeling Language (UML) [18] has become the de facto standard modeling language in the design process of object–oriented systems including a variety of complex applications ranging from object–oriented software to embedded real-time systems. Both static and dynamic aspects of such systems are captured visually, by a series of diagrams. The dynamic behavior of system objects are described by *UML Statecharts*, which is a statemachine variant having its origins in the well–known formalism introduced for the first time by Harel in [8].

However, the growing complexity of IT systems revealed several shortcomings of the language stemming from the fact that UML lacks a precise dynamic semantics. Whereas the static semantics is described by metamodels and a constraint language up to a certain level of preciseness, its execution semantics is only given informally in a natural language.

Unfortunately, there is a huge abstraction gap between the "graphical" world of UML and all the mathematical models of describing dynamic semantics (such as transition systems, Petri nets, abstract state machines, process algebras, etc.). Consequently,

[⋆] This work was carried out during my visit to Computer Science Laboratory at SRI International (333 Ravenswood Ave., Menlo Park, CA, U.S.A.) and supported by the National Science Foundation Grant (CCR-00-86096), and the Hungarian Information and Communication Technologies and Applications Grant (IKTA 00065/2000) systems engineers will require the back–annotation of analysis results into the original UML formalism, as well as an easy–to–understand, visual specification of the dynamic semantics.

Graph transformation (see e.g. [17]) provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm. Its potential domains of application include e.g. pattern recognition, functional programming languages, database systems, distributed systems, and, recently, transformations within and between UML diagrams (cf. [6, 9, 23]).

In the current paper, we present a *rule-based, visual specification of statecharts semantics by* means of *model transition systems* (a combination of metamodeling and graph transformation with explicit control structures) that provides a better understanding for systems engineers by separating derived static concepts (conflicts, priorities, etc.) and their dynamic interpretation (enabledness, fireability).

In fact, the framework presented here was implemented to form the semantic front-end for several analysis methods within our general, transformation-based formal verification and validation framework of UML models based on the VIATRA environment [3]. For instance, in [21], we propose an automated encoding of model transition systems into the SAL (Symbolic Analysis Laboratory [1]) intermediate language to provide access to *wide range of verification methods* provided by the SAL environment. The UML statechart semantics of the current paper served as the benchmark application for evaluating this encoding, however, a detailed discussion of this approach is out of the scope of the current paper.

1.1 Related Statecharts Semantics

Since the original formalism of Harel [8], the theory of statecharts has been under an extensive research and many different semantic approaches evolved from the academic world (a comparison of different approaches can be found in e.g. [24]). However, as the industrial interest is rather limited to the Statemate and UML variants, therefore, the majority of recent approaches for statecharts semantics have typically focused on the formalization of those variants. In this section, we restrict our attention to compare only proposals for the UML dialect with a stress on the support of formal verification.

Extended Hierarchical Automata, which form the structural basis of our statechart semantics, were introduced in [14] for Statemate and in [13] for UML. In a second phase, both approaches transform their models into Promela code and verify them by the model checker SPIN [10]. A major stress is put on formal verification in [20] where UML statecharts are encoded into a PVS [15] specification enabling the access to automated theorem proving of UML design, while in [12], the model checking of UML statecharts is aimed.

An entire verification round-trip is reported in [2] and [16] where the results of model checking are represented visually in the original UML models. The semantic core of statecharts is formalized in those papers by means of abstract state machines and state term graphs, respectively.

Despite their success from a verification point of view, the use of precise, formal mathematics is also the common weakness of all these approaches: they fail to provide a high level of abstraction that can be properly understood (and implemented) by systems engineers. Previous proposals in the field of graph transformation (e.g., [7, 11]) have tried to tackle this problem by providing a *visual specification* of statechart semantics.

Even though these proposals derive their internal graph representation for UML models directly or indirectly from the standard UML metamodel, semantic concepts are typically hard coded into the semantic rules, which does not scale up well for different statechart languages or the future evolution of the UML language itself. For instance, to implement the inverse priority concepts of Statemate semantics would require a major revision in all these approaches.

In the current paper, we define the dynamic behavior of UML statecharts by combining metamodeling and graph transformation techniques. However, our main contribution is to simultaneously *include the purely syntactic* (states, transitions, events) *and derived static semantic concepts of statecharts* (like conflicts, priorities, etc.) *in the metamodel, but separate them from their dynamic operational semantics,* which is specified by graph transformation rules. This philosophy keeps the metamodel and the graph transformation rules easy to be understood and maintained for statechart variants. Additionally, our methodology provides direct access to the formal verification of UML statecharts by applying the techniques investigated in [21].

The rest of the paper is structured as follows. Section 2 gives a brief introduction to UML Statecharts and Extended Hierarchical Automata [13], which latter one will serve as the underlying mathematical structure. In Sec. 3.1, an theoretical overview is provided on model transition systems, while Sec. 3.2 formalizes the semantics of statecharts. Finally, Sec. 4 concludes our paper.

2 An Informal Introduction to UML Statecharts

UML statecharts (see the example in Fig. 1) are an object–oriented variant of classical Harel statecharts [8] that describe behavioral aspects of the system under design. In fact, the statechart formalism itself is an extension of traditional state transition diagrams.

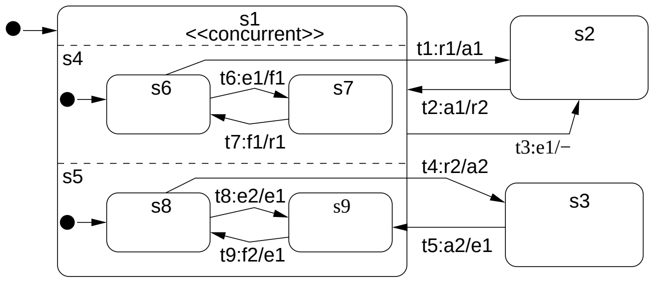


Fig. 1. A sample UML statemachine

UML **Statemachines** are basically constructed from *states* (including the top state of the hierarchy) and *transitions*.

States As one of the main concepts of statecharts is state refinement, states can be either *simple*, *composite* or *concurrent* (disregarding from pseudo states like *initial* and *final* states).

Simple states (like `s2` or `s3` in Fig. 1) are at the final level of refinement. State `s1`, on the contrary, is refined into two distinct regions (represented as a state in UML), `s4` and `s5`, each of them is refined in turn into an automaton consisting of further substates (e.g. `s6`, `s7`). States refined to sub–states are denoted as **composite**, additionally, `s4` and `s5` are called **concurrent** regions of the concurrent state `s1` as each of them has an active substate.

At one point in time, the set of all active states forms the active **configurations**. For instance, our sample system can be any of the following configurations: {`s1,s6,s8`}, {`s1,s6,s9`}, {`s1,s7,s8`}, {`s1,s7,s9`}, {`s2`}, {`s3`}.

Transitions A **Transition** connects a **source** state to a **target** state. A transition is labeled by a trigger **event**, a boolean **guard** and a sequence of **actions**.

A transition is enabled and can fire if and only if its source state is in the current configuration, its trigger is offered by the external environment and the guard is satisfied. In this case, the source state is left, the actions are executed, and the target state is entered.

In our example, if event `a1` is offered by the environment and the current configuration is {`s2`}, then state `s2` is left and state `s1` is entered. In particular, as `s1` is composite, we also have to define which are the *substates* that are reached. In the case at hand, they are the default ones specified by the **initial** states of `s4` and `s5`, namely, `s6` and `s8`. In a general case, the source and target state of a transition may be at a different level of the state hierarchy. Such a transition is denoted then as **interlevel**.

Event dispatching In general, more than one event can be available in the environment. The UML semantics assumes a **dispatcher**, which selects one event at a time from the environment and offers it to the state machine. As a result, more than one transition can be enabled, which may cause a **conflict** to be resolved if the intersection of the states left by the enabled transitions is not empty. Conflicting transitions are tried to be resolved by using priorities: a *transition has higher priority* than another transition if its source state is a substate of the other transition's source state. If conflicts cannot be resolved by priorities, any of the enabled transitions can be fired, moreover, according the the run-to-completion step, all transitions from the non–conflicting subset of enabled transitions are fired at a time.

Non–standard UML extensions In the current version of our statechart semantics, several concepts of the UML standard (such as history states, deferred events) have been omitted for space limitations. We hope that our formalization concepts in Section 2.1 will demonstrate that the "neglected" parts can easily be integrated into a future version.

On the other hand, we also had to extend the original UML standard due to the lack of a proper specification for event queues. Currently, one queue is attached to one object and stores a *sets* of events, however, our approach can easily be extended to alternate queue models (FIFO, multiset, etc).

2.1 Extended Hierarchical Automaton

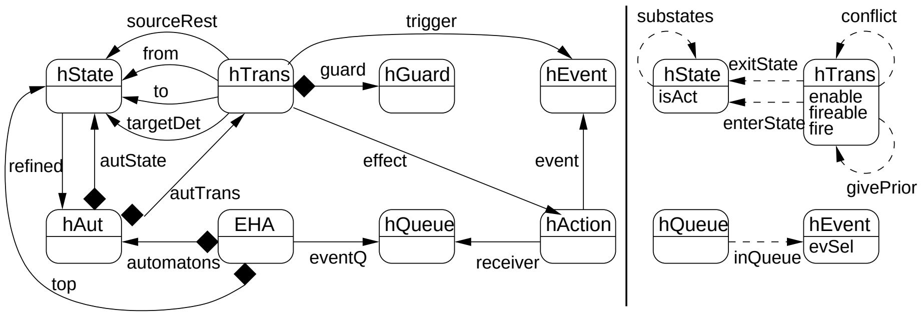
The UML version of Extended Hierarchical Automaton (EHA) were introduced in [13] to provide an alternate representation and a formal operational semantics for Statechart diagrams by a small number of complex transition rules. In this paper, the EHA model is considered to be only an *alternate structural representation of statecharts*, while the original semantic domain is replaced by a set of dynamic attributes and relations manipulated by graph productions.

Note that the formalization method to be presented below could be applied straightly to the language of UML statecharts. We believe that the intermediate EHA representation provides greater flexibility when further statechart variants (e.g. the Statemate or Matlab dialects) are considered in the future. In fact, the EHA notation can be derived automatically from the original UML statechart notation by a rather syntactic graph transformation process (see a detailed discussion in [22]).

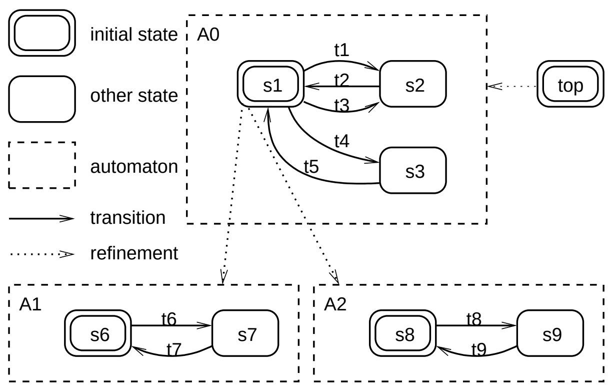
Metamodel and model of Extended Hierarchical Automaton The structural basis of Extended Hierarchical Automaton are defined by its metamodel in Fig. 2(a), while the EHA encoding of our sample statechart is shown in Fig. 2(b). The classes of the EHA metamodel are prefixed with the letter *'h'* in order to avoid name clashes with the original notions of UML statecharts.

– An Extended Hierarchical Automaton (EHA) is composed of a top hState, and sequential automatons of class hAut. Additionally, an EHA is attached to an arbitrary number eventQueues. Each instance of a UML class that is associated with a statemachine is projected into a distinct EHA instance.
– A sequential automaton hAut is generated for (i) each non-concurrent composite state and (ii) for each regions of a concurrent composite state in a UML statechart. It is composed of hStates (referred by autStates) and hTransitions (accessed by autTrans).
– Each UML state that is not a region of a concurrent state is transformed into a hState. The state refinement relation is preserved by the refined association linking states to their subautomatons.
– A transition in UML statecharts has a corresponding transition hTrans in its EHA equivalent linking the from hState and the to hState. All EHA transitions are non-interlevel thus connecting states that belong to the same sequential automaton, which is the EHA equivalent of the *least common ancestor*[1] state in the UML statemachine. The original source state(s) of transition is denoted by the source restriction relation (sourceRest). The target determinator (targetDet) of a transition relates the set of hStates that (i) have a simple state equivalent in the original UML model, and (ii) they must be entered implicitly when the transition is fired (thus more than a single target determinator state may be connected to e.g., transition t2).
– A hTransition is triggered by a related hEvent, and its effect is defined by its corresponding hAction. In the paper, we restrict our attention to send actions, i.e.,

[1] The *least common ancestor* of a UML transition is the lowest level state that is a superstate of both the source and the target states



(a) An EHA metamodel



t	SR	EV	AC	TD
t1	s6	r1	a1	s2
t2	s2	a1	r2	s6 s8
t3	s1	e1	−	s2
t4	s8	r2	a2	s3
t5	s3	a2	e1	s6 s9
t6	s6	e1	f1	s7
t7	s7	f1	r1	s6
t8	s8	e2	e1	s9
t9	s9	f2	e1	s8

(b) A sample EHA model

Fig. 2. Extended Hierarchical Automaton

all the actions are considered as sending an event to the specified eventQueue. The guard condition is a boolean expression that must hold to allow the transition to be enabled.

– An EHA object is automatically associated to one event queue where the messages sent to the object arrive from.

In order to improve the clarity of Fig. 2(b), we represented the events (EV), actions (AC), source restrictions (SR) and target determinators (TD) of a transition in a table. Note that grey areas in the table represent the corresponding values for interlevel UML transitions.

In the metamodel, static parts of the metamodel (left from the vertical line) were kept separated from dynamic (and derived) relations and attributes (right from the vertical line). All the previous concepts are regarded as *static parameters*, since they are derived from the original UML model at compile time. For describing the *dynamic be-*

havior of statecharts in an easy-to-understand way, additional attributes and relations are required.

- The boolean attribute isAct of a hState will be set to true whenever the hState is active (i.e. member of the current configuration). The attribute evSel of a hEvent is true when this hEvent is selected by the dispatcher (as the dispatcher belongs to the environment, this attribute will be handled non–deterministically).
- The relation substates connects a hState to its descendent hStates in the EHA state hierarchy. When a hQueue is related to a hEvent by an inQueue edge, this fact denotes that the hEvent is a member of the hQueue set.
- The attributes enabled, fireable and fire will denote, respectively, (i) when a transition is triggered by the selected event and its origin is an active state, (ii) it is enabled and has sufficient priority to be fired, and (iii) it is selected to be fired.
- There are four additional relations for hTransitions, which are, in fact, static but not part of the EHA model introduced in [13]. The relation exitState explicitly enumerates all the states that have to be exited when the transition is fired. Similarly, the enterState relation lists all the states to be entered when a transition is fired. Two hTransitions are in a conflict relation (i.e. they might be in conflict when firing them) if their exitState set is not disjoint. While givePrior specifies the priority relation between hTransitions.

The major distinction between static and dynamic concepts is that static parts are not modified while an EHA is being operated by the upcoming graph transformation rules. Therefore, the dynamic attributes (and relations) are *initialized* and not *compiled*. With this respect, the semantic formalization of EHA will be divided into two subsequent phases, namely, (i) a *preprocessing* phase for generating derived properties and initializing dynamic constructs, and (ii) the *execution* of core EHA semantics itself.

3 A Rule-Based Visual Semantics for UML Statecharts

3.1 Theoretical background: Model Transition Systems

Up to this point, the only the *syntactic domain* of statechart models and metamodels were discussed. A traditional approach of defining *static semantics* for models and metamodels is provided by a mapping to directed, typed and attributed graphs. In this sense, a model will conform to its metamodel, if its model graph conforms to the corresponding type graph with respect to a typing homomorphism.

Type graphs and model graphs All classes are mapped into a graph node and all associations are projected into a graph edge in the **type graph**. The inheritance hierarchy of metamodels can be preserved by an appropriate *subtyping relation* on nodes (and possibly, on edges). Class attributes are derived into graph attributes where the latter may be treated mathematically as (possibly partial) functions from nodes to their domains.

Objects and links between them are mapped into nodes and edges, respectively, in the **model (instance) graph**. Each node and edge in the model graph is related to a corresponding graph object in the type graph by a corresponding *typing homomorphism*.

The operational dynamic semantics of Hierarchical Automaton will be formalized by *model transition systems* (introduced in [23]), which is a variant of graph transformation systems with a predefined set of control structures.

Definition 1. *A **model transformation rule** $r = (L, N, R)$ is a special graph transformation rule, where all graphs L, N and R are model graphs.*

The **application** of r to a **host graph** G (according to the single pushout approach [5]) replaces an occurrence of L (left-hand side, LHS) in G by an image of R (right-hand side, RHS) yielding the derived graph H. This is performed by

1. *finding an occurrence* of L in G, which is either an isomorphic or a non–isomorphic image according to M
2. *checking the negative application condition* N, which prohibit the presence of certain nodes and edges (negative application conditions are denoted by shaded grey/red areas labeled with the NEG keyword).
3. *removing* those nodes and edges of the graph G that are present in L but not in R yielding the **context graph** D (all dangling edges are removed at this point)
4. *adding* those nodes and edges of the graph G that are present in R but not in L attaining the **derived** graph H.

The entire model transformation process is defined by an initial graph manipulated by a set of model transformation rules (micro steps) executed in a specific mode in accordance with the semantics (macro steps) of a hierarchical control flow graph.

Definition 2. *A **model transition system** $MTS = (Init, R, CFG)$ with respect to (one or more) type graph TG is a triple, where $Init$ defines the **initial graph**, R is a set of **model transformation rules** (both compatible with TG), and CFG is a set of a **control flow graphs** defined as follows.*

– *There are six types of nodes of the CFG: **Start**, **End**, **Try**, **Forall**, **Loop** and **Call**.*
– *There are two types of edges: **succeed** and **fail**.*

The control flow graph is evaluated by a virtual machine which traverses the graph according to the edges and applies the rules associated to each node.

1. The execution starts in the **Start** and finishes in the **End** node. Neither types of nodes have rules associated to them.
2. When a **Try** node is reached, its associated rule is tried to be executed. If the rule was applied successfully then the next node is determined by the **succeed** edge, while in case the execution failed, the **fail** edge is followed.
3. At a **Loop** node, the associated rule is applied as long as possible (which may cause non-termination in the macro step).
4. When a **Forall** node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph.
5. At a **Call** node (which has an associated CFG and not a rule) the state of the CFG machine is saved and the execution of the associated CFG is started (in analogy with function calls in programming languages). When the sub CFG machine is terminated, the saved state is restored, and the execution is continued in accordance with the outgoing edge (**succeed** or **fail**).

Note that this CFG model follows the control flow concepts of the VIATRA tool. However, the use of "as long as possible" kind of control conditions (and additional negative application conditions) instead of forall nodes would almost directly yield the appropriate control conditions, for instance, for PROGRES [19].

.

3.2 An operational semantics of Extended Hierarchical Automaton

The semantics of Extended Hierarchical Automaton is defined by a model transition system. The **initial graph** is the *static model* generated by the SC2EHA transformation. The initialization of the dynamic aspects and the derivation of extensional attributes are separated into a *preprocessing phase*, while the execution of the semantic rules of EHA form the *operational phase*. The top–level *ehaSemantics* module thus consists of two **Call** nodes — one for the specification of each phases.

The preprocessing phase The *initDynamics* module is mainly responsible for *deriving those static relations* that are required for an easy-to-understand formalization of dynamic behavior. In addition, the initialization of dynamic attributes also takes place at this stage. The preprocessing phase consists of 10 rules (see Fig. 3) that are executed in according to the control flow graph in the following order. An alternate solution for this preprocessing phase is to use path expressions for such derived relationships.

1. *substatesR1*: The rules *substatesR1* and *substatesR2* build up the substate relationship in two steps. At first, if a hState S_1 is refined to a hAutomaton A, all the hStates S_2 of this automaton are substates of S_1 (a **Forall** execution).
2. *substatesR2*: Secondly, the transitive closure of the substates relation is calculated by looping rule *substatesR2* as long as possible. This means to add **substates** edges between hStates S_1 and S_3, if S_2 is substate of S_1, S_3 is a substate of S_2 but no substate edges are leading yet between S_1 and S_3
3. *exitStateR1*: Rules *exitStateR1* and *exitStateR2* explicitly connect the hStates that are exited by a hTransition when the transition is fired to the hTransition. At first, the **from** hState S of a hTransition T must be exited.
4. *exitStateR2*: Afterwards, all the **substates** S_2 of the **from** state S_1 of the hTransition T must also be exited.
5. *enterStateR1*: All the states target determinator hStates S of a firing hTransition T are should be entered when firing this transition.
6. *enterStateR2*: Additionally, all the states S_2 that are *superstates* of the target determinator state S_1 of a hTransition T but *not superstates* of the **to** hState S_3 must also be entered. Note that the states to be exited and entered when firing a transition are static information.
7. *conflictR*: The rule *conflictR* connects two hTransitions T_1 and T_2, if there exists a hState S that is linked to both of them by an **exitState** edge, thus it is exited by both transitions causing a *conflict*.
8. *givePriorityR*: According to this rule, a hTransition T_2 has lower priority than hTransition T_1 if for the corresponding source restriction hStates (S_2 and S_1, respectively), S_1 is a substate of S_2.

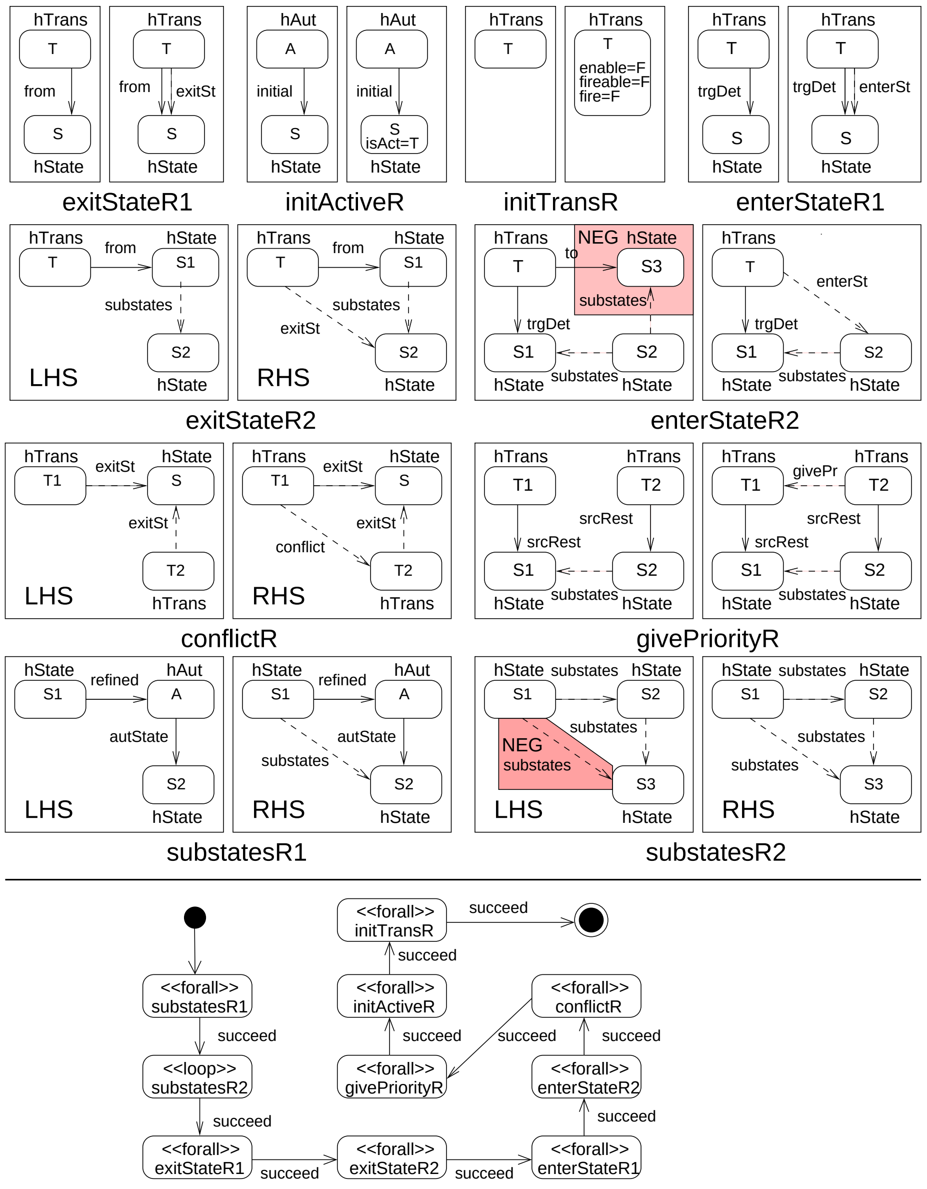


Fig. 3. Static relations in the EHA model

9. *initTransR* and *initActiveR*: Initially, all the dynamic attributes of hTransitions are set to false.
10. *initActiveR*: A state S becomes active if it is an initial state of some sequential hAutomaton A.

Example. By the end of the preprocessing phase of the EHA model in Fig. 2(b), we derived, for instance, that

– s_6, s_7, s_8 and s_9 are substates of s_1;
– the states to be exited when firing transition t_1 are s_1 (the **from** state), s_6, s_7, s_8 and s_9 (the substates of the **from** state);
– the states to be entered when firing transition t_2 are s_6, s_8 (the target determinators) and s_1 (which is the **to** state);
– transitions t_1, t_3 and t_6 are in conflict with each other since state s_6 is exited by all of them;
– however, t_3 gives priority to t_1 and t_6 as the source restriction state s_1 of t_1 is a superstate of s_6 (the source restriction of t_1 and t_6).

Operational phase Now we continue with the discussion of the "more semantical" operational phase (depicted in Fig. 4), where the run-to-completion step of statecharts are refined into a *sequence of more elementary operations*.

1. *selectEventR*: At first, an event E is non-deterministically selected from an event queue Q. If no such events are available, then the execution of EHA terminates. Note at this point, that different (more complex) event handling mechanisms can be selected at this point, however, this non-deterministic selection overapproximates the semantics of such mechanisms.
2. *enableR*: A hTransition T is *enabled* if its source restriction hState S is active, and its trigger hEvent E is selected by the dispatcher.
3. *fireableR*: An enabled hTransition T_1 becomes *fireable* if there are no enabled hTransitions T_2 of higher priority (see the negative condition).
4. *fireFirstR*: The first (fireable) hTransition T is selected to be fired non–deterministically by setting its **fire** attribute to true. If no such transitions found then the execution continues by resetting dynamic attributes of transitions (see the final step).
5. *fireNextR*: After the success of *fireFirstR*, the set of transitions to be fired fired is extended one by one (by looping `fireNextR`) until all the remaining enabled hTransitions are in conflict with at least one element in the fire set.
6. *exitR*: All the states S marked by an **exitState** edge leading from a firing hTransition T are exited.
7. *addQueueR*: As the effect of firing a hTransition T, the hEvent E associated to the (send) hAction A is added to the corresponding hQueue Q (if the negative condition is removed, then the event queue is modeled as a bag and not a set).
8. *enterR*: All the states S marked by an **enterState** edge leading from a firing hTransition T are entered. This step results in a valid configuration as (i) the origins of target determinator states were simple states, hence no states need to be entered at a lower level than the target determinators and (ii) all EHA transitions are non–interlevel thus no states had been exited at a higher level than the **from** and **to** hStates.

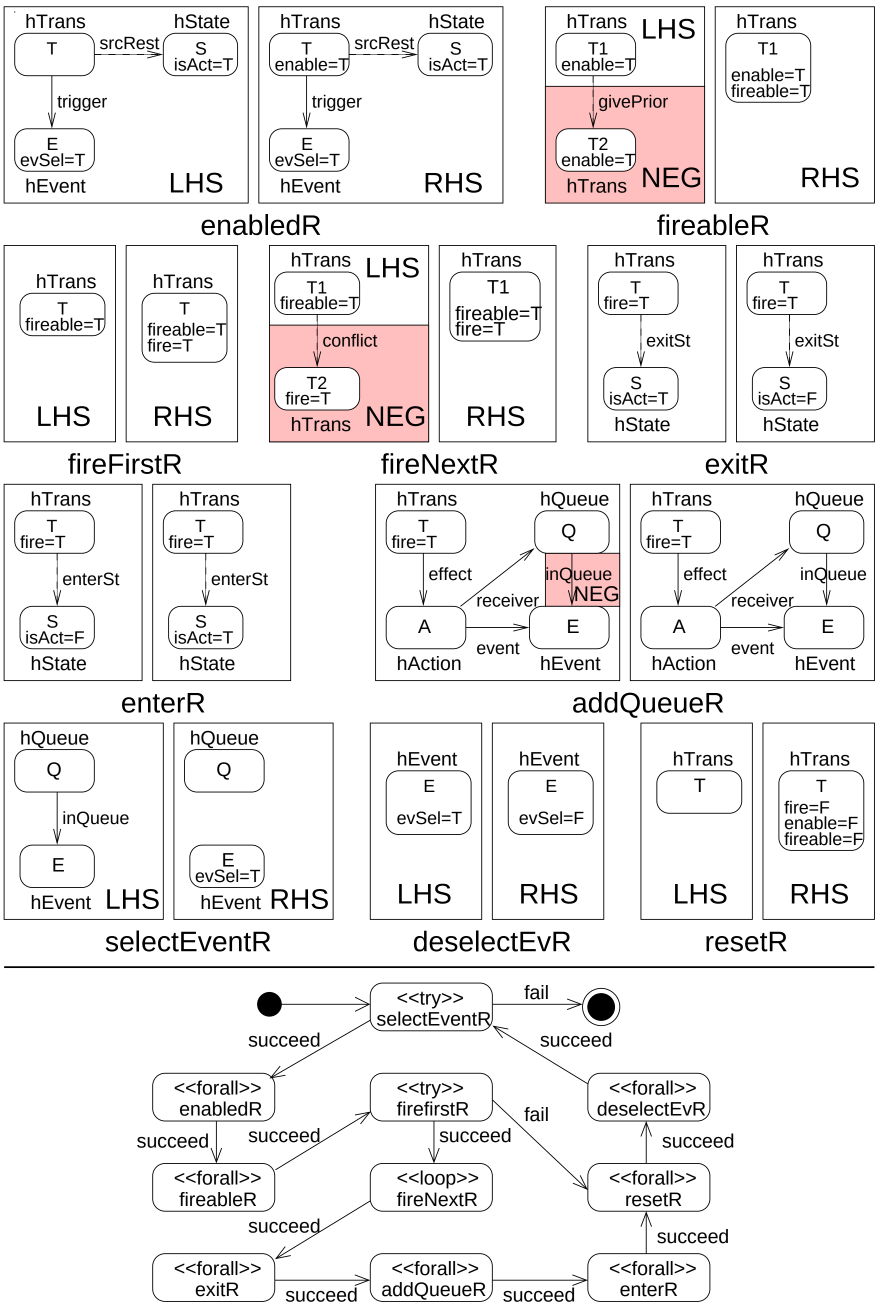


Fig. 4. The model transition system specifying the EHA semantics

9. *resetR* and *deselectEvR*: All the dynamic attributes of a transition T and an event E are set to false, and a new step of the EHA commences.

Example. Let us assume that the active states of our sample EHA model are the initial states (thus s_1, s_6, and s_8) and the event queue only contains the single event of e_1. According to the previous rules, a run-to-completion step of our hierarchical automaton proceeds as follows.

1. Transitions t_3 and t_6 are enabled by applying *enableR* as the source restriction states (s_1 and s_6, respectively) of both transitions are active.
2. From this enabled set of transitions, the application of *fireableR* eliminates t_3 since t_3 gives priority to t_6.
3. The set of transitions to be fired will consist of the single transition t_6.
4. By applying *exitR*, states s_1, s_6 and s_8 become inactive, while the application of *enterR* results in the activation of state s_2. Meanwhile, event f_1 is added to the event queue by rule *addQueueR*.
5. Finally, all dynamic attributes of all transitions are set to false, and a new run-to-completion step commences.

4 Conclusions

In the current paper, we proposed a visual operational semantics for UML statecharts based on metamodeling techniques (Extended Hierarchical Automaton as the underlying static structure) and model transition systems (for defining operational semantics). The main contribution of the paper is to partition the complex (but rather informal) semantic rules of statecharts in the UML standard into elementary operations separating derived static concepts (conflicts, priorities) from their dynamic interpretation (enabledness, fireability). In this respect, our approach can be easily adapted to different statechart variants (e.g., with different priority concepts) and upcoming changes in the UML standard.

The presented framework was tested within the VIATRA tool [23]. Moreover, following the guidelines of [21], we directly transformed our UML statechart semantics to SAL specifications [1] in order to provide access to a combination of symbolic verification techniques.

Acknowledgments

I would like to thank András Pataricza, István Majzik (Budapest University of Technology), John Rushby and many of his colleagues (at SRI International) the fruitful discussions, and their encouragement and support. Moreover, I am also very grateful to the anonymous reviewers for their valuable comments.

References

1. 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, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

2. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, 2000.
3. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, Edinburgh, UK, September 23–27 2002. Accepted paper.
4. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
5. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [17]*, chapter Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pages 247–312. World Scientific, 1997.
6. 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, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
7. M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
9. R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pages 11–21, Grenoble, France, April 2002.
10. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
11. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.
12. G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.
13. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.
14. E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *ASIAN'97 Third Asian Computing Conference. Advances in Computer Science*, volume 1345 of *LNCS*, pages 181–196. Springer-Verlag, 1997.
15. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS language reference, Version 2.3. Technical report, SRI International, September 1999.
16. I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In R. J. Hall and E. Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
19. A. Schürr, A. J. Winter, and A. Zündorf. *In [4]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

20. I. Traoré. An outline of PVS semantics for UML Statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, Nov. 2000.
21. D. Varró. Towards symbolic analysis of visual modelling languages. In P. Bottoni and M. Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *ENTCS*, pages 57–70, Barcelona, Spain, October 11-12 2002. Elsevier.
22. D. Varró and A. Pataricza. Mathematical model transformations for system verification. Technical report, Budapest University of Technology and Economics, May 2001.
23. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205–227, August 2002.
24. M. von der Beeck. A comparison of statecharts variants. In *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, 1994.