Formalizing UML State Machine Semantics for Automatic Verification–the PAT Approach

Liu Shuang
National University of Singapore
lius87@comp.nus.edu.sg

Supervisor: Dr. Bimlesh Wadhwa and A/P Dong Jinsong

August 28, 2012

Abstract

UML state machine is widely used in modeling the dynamic behavior of object-oriented designs in industry. But UML state machine specification, which is maintained by Object Management Group(OMG), is documented in natural language instead of formal language. The inherited ambiguity of natural language may introduce inconsistencies to the resulting state machine model. Formalizing UML state machine specification will solve the ambiguity problem and provide a uniformed view to software designers and developers. It also provides a foundation for automatic verification of UML state machine models, which can help to find software design vulnerabilities at an early stage and reduce the development cost. In this report, we are going to provide a thorough survey of existing work related to formalizing UML state machine semantics and automatic validation of UML state machine model dynamic behavior. We also discuss the shortcomings of existing approaches and propose our own solution for this problem.

Contents

1 Introduction

UML state machine specification, published and managed by Object Management Group (OMG) [4], is an object-oriented variation of Harel Statechart [33]. UML state machine has become a standard for modeling dynamic behaviors of object-oriented designs in industry. It is inevitable for errors to exist in the models since modeling is a human-intensive activity. Detecting such errors in the modeling phase will dramatically reduce the cost in the software development cycle. But there is a pending problem left open for years, i.e. the informal natural language description of UML specification introduces a lot of ambiguities and inconsistencies. To make things worse, the inconsistencies between the natural language descriptions are tedious for manual detection and are hard to be verified automatically due to its informal nature.

There are some work done in the literature to formally define UML state machine semantics[54, 44, 27, 39, 47, 37, 11]. But all of those approaches just provide a formalization for a subset of UML state machine. Further, most of those approaches are not meant for building an automatic verification tools, and some of the semantic domains they used such as Abstract State Machine(ASM), are not suitable for automatic verification.

Another kind of related work explores a translation approach[45, 40, 42, 41, 43, 25, 22, 13, 55] in which a UML state machine model is translated into languages used by Model checkers. Model checking is then conducted on these translated models. But we notice that translation based approaches suffer from four drawbacks. Firstly, it is hard to trace back to the original model if a bug is detected in the translated model. Secondly, the translation may introduce more redundant behaviors which slows down the verification process. Thirdly limited by the translated language, most approaches consider quite restrictive subsets of the UML syntax defined by OMG [6]. Lastly and the most importantly, it is hard to verify whether the translation procedure and verification results are correct or not since it is done in an unsystematic and informal way. Due to the above four drawbacks, translation approaches are less reliable.

We have also done a survey on the available tools to automatically verify UML state machines. To the best of our knowledge, there are no tools available to directly support model checking UML state machines. This also motivates us to formally define and implement the operational semantics of UML state machine to bridge the gap.

The rest of this report is organized as follows. Section 2 and Section 3 discuss formalization approaches and translation approaches separately. In Section 4, we surveyed the existing tools which support formal verification of UML state machines. We analyzed UML behavior state machine specifications [6] thoroughly and discuss our own understanding in Section 5. In Section 6, we proposed our formalization for UML v2.4.1 state machine. Future work is discussed in Section 7 and Section 8 concludes the report.

2 Formalizing UML behavior state machine semantics

Formalizing UML behavior state machine semantics is the most related work we are interested in. Different approaches use different semantic domains. Since our purpose is to support automatic verification of UML state machine, especially model checking instead of other forms of formal verification, such as theory proving, we will focus on the formalization of UML state machine into those semantic domains related to model checking, such as Kripke structure, petri nets. We categorize our survey based on the formal formats used as semantic domains.

2.1 Labeled Transition System as Semantic Domain

Some researchers choose to use LTS as the semantic domain and make it seamless to combine with the automatic model checking of UML state machines since most model checkers use LTS to represent the search space of a system. [44] and [54] are representative approaches in this direction. Latella et al. [44] are among the pioneers who begin to focus on formalizing UML statecharts(instead of Harel statechart [33]) semantics and the semantic domain their formalization adopted was Kripke structure. They use a slightly modified variant of Extended Hierarchical Automata(EHA) as an intermediate model and map the UML-statecharts into an EHA. The hierarchical structure of UML statecharts and EHA make the translation structured and intuitive. Then they define the operational semantics for EHA in the domain of LTS, i.e. using LTS to express the dynamic semantics of EHA. This approach covers a quite restricted subset of UML state machine structures, no pseudostates (exclusive of initial pseudostate) are considered, no actions associated with states, i.e. entry/exit/do action, deferred events, are considered and the triggering events are restricted to signal and call events without parameters. Although this work provides a promising direction of formalizing UML state machine dynamic behaviors, it is far from the described goal, i.e. automatically model checking UML state machines, since even very basic structures commonly used by system designers such as final state, completion events and entry/exit actions are not supported. A more serious problem may be that EHA cannot express the meaning of local transitions as is defined by OMG UML state machine specifications. This is explained and illustrated in Appendix A.

[54] also formalized a partial set of UML statecharts, which partially based on the work proposed in [44]. But it supports some more features such as history mechanisms, entry and exit actions compared to [44]. The syntax used in this work is called UML-

statechart terms, which is inductively defined on three kinds of terms, i.e. Basic term, Or-term and And-term. All of them contain basic information about a state such as a unique ID, entry and exit actions, and sub-terms(for Or-term and And-term) which contains the hierarchical information of a UML state machine. UML-statecharts terms basically represent static information about UML-statecharts vertices. Inter-level transitions are captured by explicitly specifying source restriction and target determinator in an Or-term, this notation follows the idea of Latella et al [44]. The dynamic behavior of UML-statecharts is represented by Configurations. A configuration captures the complete current status of a given UML-statecharts term, i.e. the hierarchical structure is considered and all currently active substates within the given term are computed. The definition of semantics takes three stages. The first stage is to define extra operations which are imposed by entry/exit actions and history mechanisms considered in this paper. Based on those defined operations, auxiliary semantics(five SOS rules) are defined to process a single input event. The auxiliary semantics is defined as a mapping from a UML state machine to a LTS. Each state in the LTS is a UML-statecharts term. A semantic transition is defined to proceed a single input event. Last, Complete semantics is given based on auxiliary semantics and a Kripke structure, i.e. the semantic domain of the complete semantics is Kripke structure. Instead of represent a UML state machine into an EHA, Beeck [54] chooses to use a UML-statechart term as the syntax domain of a UML state machine. A UML-statechart term also has the power of expressing hierarchical information as EHA does, it is more flexible than EHA since there are no restrictions about the source and target of a transition, thus transition $t3, t5$ in Figure A.1 can be expressed in a UML-statechart term. One limitation of this approach is that too few features are considered, though more than that supported by [44], and we will see in our proposed approach in Section 6 that adding those features are not trivial.

Kwon proposed another approach [43] which utilizes Kripke Structure as the semantic domain and aimed at model checking UML state machines. Similar to [54], Kwon uses Term, which represent state hierarchy in the form of subterms as field in Term, as the syntax domain of UML state machine. But Kwon [43] use the conditional rewrite rules to represent the transition relation in a UML state machine(while Breeck [54] explicitly defined 5 SOS rules). Then the semantics of UML state machines is defined as a Kripke Structure. This paper [43] also provides a translation from the defined Kripke Structure to the input language of SMV Model checker. We will discuss it in more detail in Section 3.

All the work we surveyed above consider only a single statechart and leave alone the interactions between different statecharts. Nevertheless, they provide a good direction by utilizing Kripke structure as the semantics domain.

2.2 Abstract State Machine as Semantic Domain

Another branch of related approaches[15, 16, 18] in formalizing UML state machine adopts Abstract State Machine(ASM) as the semantic domain. ASM can offer the most general notion of state in the form of structures of arbitrary data and operations which can be tailored to any desired level of abstraction. Function update, which is less abstract but is more close to the nature flow of the dynamic behavior of UML state machines, is used to represent operations. On the other hand, the notion of multi-agent(distributed) ASMs can naturally reflect the interaction between objects. Although this is a little unrelated to model checking UML state machines, as compared with those approaches using LTS[44, 54, 43] as semantic domains, these approaches always cover more features of UML state machines, thus provide some ideas about solving ambiguities and semantic variant points in UML state machines. These approaches also consider interactions between objects instead of a single object considered by [44, 54, 43]. So we are going to briefly survey these approaches.

Börger et al. [15, 16, 18] are among the pioneers in formalizing UML state machines into ASMs, which contains a collection of states and a collection of rules(conditional, update, Do-forall etc) which updates those states. [15] is the first piece of work in this direction. UML static structures are represented as states and transitions which belongs to the abstract sets STATE and TRANSITION. Simple states, composite state, orthogonal composite state and final state, initial and history pseudostates are considered. Transitions are further partitioned into External transitions, Internal transitions and Completion transitions. The syntax domain of UML state machine is multi-sorted first-order structures, i.e. sets with relations and functions. Agents execute UML state machines using the ASM update rule. These ASM update rules are of the form "if *Condition* then *Updates*", where *Updates* is a set of function updates which are simultaneously executed when *Condition* is true. This approach covers most UML state machine features, including deferred events, completion events and internal activities associated with states which are mostly left out by other approaches. But pseudo states such as fork, join, junction, choice, terminate are not considered. In contrast, the authors argue that these constructs can find their semantically equivalent constructs in their defined subset, where they use a transition from(resp. to) the boundary of a orthogonal composite state to replace the join(resp. fork) pseudostate. But in terms of join(resp. fork) pseudostate, we can decide which substates of the target orthogonal composite state are going to be entered simultaneously, while this semantic meaning is not expressible by the equivalent construct they provided.

[16] add concurrent behaviors in the context of event deferring and run-to-completion to the the formalization in [15]. This is achieved by providing new submachines(subagents) covering transitions from and to(cross the boundary of) orthogonal composite states.

[18] provides some further discussions about the ambiguities in the official semantics of UML state machines[5] and their solutions. The work by Eörger et al.[15, 16, 18] covers more features compared to those LTS approaches [44, 54, 43] and the formalization is much easier to follow due to the similarities of ASM notations with pseudo code. But no automatic verification has been done based on these work so far. We believe in the importance of automatic verification on UML state machines and want to make the work move further than just formalization on papers.

[22] is another approach which uses ASM as semantic domain to formalize UML state machines. To be precise, the semantic domain they use is called extended ASM, in which they extend the ASM to represent inter-level transitions with multiple transitions which do not cross any boundary of states. This extension makes it easier to deal with interruptions, it also makes the formalization procedure more structured and layered(Since inter-level transitions break the hierarchical structure of UML state machine and such a decomposition of inter-level transitions into multiple transitions preserve such an hierarchical structure). It shares similar idea with [15, 18] in other aspects of the formalization, which maps a UML state machine directly to an ASM. Agents are used to process executions of UML state machines. But [22] provides an Activity Agent, which is responsible of modeling the execution of an activity associated with a node.(In [15], a rule Generate Completion Event is used for this purpose.) The execution of agents are divided into different modes, which indicates what kind of rules(operations) the current agent should take.

Jin et al. [39] provides an approach which syntactically defines UML statecharts as attributed graphs which are described by Graph Type Definition Language(GTDL). The abstract syntax of the attributed graphs is also provided as a six-tuple, which specifies vertices, Transitions, mapping from a transition to its source and target state respectively, a container function which specifies state hierarchy and a mapping from attribute associated with a vertex/transition to its value. They further provide some constrains in the form of predicates to specify the well-formedness rules of statecharts, which is considered as the static semantics of a UML statecharts since the rules specifies how a UML statecharts should be constructed. The semantic domain is defined as a Object Mapping Automata(OMA [36]), which is a variant of ASM. Given the abstract syntax(of the attributed graph) of a well-formed statechart, they first "compile" it into OMA algebraic structures, which specifies "advanced static semantics" of a UML statecharts by taking pseudostates into account. Based on OMA algebraic structures, two rules, viz initialization rule and run-to-completion rule are defined to describe the dynamic behaviour of a UML statecharts. The syntax and semantics provided by this approach, benefiting from the highly compatibility of the abstract syntax of attributed graph with UML statecharts, are more intuitive and easy to follow. But it supported a limited subset of UML state machine features and does not even include concurrent composite states as well as choice vertex.

Seen from the number of works, ASM is more preferred by researchers than LTS in formalizing UML state machine semantics. The reason may be that the update rules of ASM are more suitable to express the complex and cumbersome semantics of UML state machines. That is also the reason that ASM based approaches always consider more features of UML state machine than LTS based approaches. We survey the ASM related work here because there are some semantic formalization of UML state machine features worth consultation. But ASM is not very related to automatic verification and there are also few tool support, so we would not adopt it as semantic domain in our work.

2.3 Petri Nets as Semantic Domain

Petri nets is a mathematical modeling language with formal semantics. Colored Petri Net(CPN) [38] is a special case of Petri net in which the tokens identifies attributes(types). It is less intuitive than Petri net, but scale better to large problems than Petri net since allowing tokens to have an associated attribute dramatically increases the expression power.

Some approaches [20, 31, 12, 11] in literature adopt (Colored)Petri Nets as semantic domain to formalize UML state machines. Robert et al. [31] presents an approach which use colored Petri Nets to model and validate the behavior characters of concurrent object architectures modeled by UML. The authors discussed how to map active/negative objects as well as message communications into CPN. Synchronous as well as asynchronous communications are discussed in message communications. Though not specifically dealing with UML state machines, this paper provide a general idea of transforming UML diagrams to Petri Nets.

Luciano et al. [12] propose another approach to formalize UML with high-level Petri Nets, i.e. Petri Nets whose places can be refined to represent composite places, and Class Diagrams, State Diagrams and Interaction Diagrams are considered. Customization rules are provided for each diagram. But the authors does not provide details about those customization rules, instead, they illustrate the steps with the Hurried Philosopher Problem, which extends the dining philosopher problem by allowing new philosophers to be temporarily invited at the table. The analysis and validation are also discussed, especially how to represent the properties, such as absence of deadlocks, fairness etc, in UML as well as how to translate them into Petri Net representations.

Christine et al. [20] propose a similar approach which formalize UML state machines in hierarchical colored Petri Nets. They provide a detailed pseudo algorithm for the formalization procedure. They map simple states of UML state machine into Petri Net places and composite states of UML state machine into Composite Petri Net places. Transitions in UML state machines are mapped to arcs in Petri Nets and corresponding triggering events are properly labeled. An extra place called *Events* is modeled with an event place in Petri Nets, with each type of event assigned different color type. Entry and exit actions of UML state machines are modeled with an arc in Petri Nets which labeled with proper event type and ends in the event place. Though the mapping from UML state machine to HCPN is clearly expressed compared to [12], a very limited subset of UML state machine features are supported, only the very basic features such as simple state, composite state, transitions, triggering event and entry/exit actions are discussed. How to type the events and how to deal with concurrency invocations of a concurrent composite state are not discussed.

Étienne et al [11]. extend the work by Christine [20] and supports a larger subset of UML state machine features, including state hierarchy, internal/external transitions, entry/exit/do activities, history pseudostates, etc. They also discussed the type of token. But concurrent composite states are still out of the scope of this work and most other pseudostates are also not considered.

Petri Nets is used in modeling work flow in industry. It is more rigorous benefiting from its mathematical supporting. But it is always hard for non-experts to understand

and it suffers from state explosion problem. People always prefer not to use it as the modeling language of software systems.

2.4 Other Semantic Domains

There are some other approaches which use different semantic domains from our surveyed category, but they are also representative and provide useful information for our own work. We discuss these approaches in this subsection.

[27] uses core state machine as the semantic domain for UML state machines. Core state machine is a 7-tuple including states, do actions, deferred events, transitions, initial state, set of variables and initial variable assignment. History is explicitly described by a mapping from a region to its direct substate. A csm-configuration w.r.t. a core state machine M is defined as a 9-tuple and is used to describe dynamic behaviour of a core state machine. [27] firstly formalize both syntax and semantics of the core state machine and then provides five steps(in natural language, including many informal "rules" specifying how the transformation should be conducted.) to transform a UML state machine into core state machine. This paper considered more UML state machine features, and accordingly more rules need to be defined since more scenarios may present. 14 configuration steps are provided on the core state machine, which forms the dynamic semantics of it. The transformation steps from UML statecharts to core state machine is also provided. But the steps are not formally stated, instead of using a structured(hierarchical) representation, as has been introduced in[44, 54], only natural language descriptions with an example illustration are given. Further, the translation is very complex since a lot of auxiliary vertices need to be added, such as enter/exit vertex. In this sense, it is less promising to be automatically verified. This approach does not explicitly model the event dispatching and generating either.

There are other approaches [47, 46, 25, 37] whose semantic domains are not known as formal structures.

[47] is an early work to formalize all features of UML statecharts diagrams. This approach takes two steps to complete the formalization. They First transform the structure of a UML state machine into a term rewriting system where the hierarchical structure of UML statecharts are rewritted into hierarchical terms. The resulting UML statemachine is a 13-tuple. The semantic domain of a UML statemachine is defined as a hypothetical machine(with the form of pseudo code), which has an event queue, an event dispatch mechanism and an event processor. A run-to-completion step is capable of dispatching

and executing events on its event queue until the current state configuration stabilizes. In-between, the run-to-completion step will also take care of deferred events, history information, call events, entry/exit actions and completion events. (Actually, [47] implement the run-to-completion(RTC) algorithm with a loop, which considers the whole lifecycle of a UML statecharts instead of a single run-to-completion step as is defined in UML superstructure specifications[6]). A large subset of features are considered compared to approaches using EHAs[44, 54], including deferred events, completion events and call events etc. But the authors still leave some features such as join, fork, junction, choice vertices unspecified and instead, claiming that these pseudostates can be replaced with extra transitions.

Jens et al. [37] provide a very comprehensive analysis about UML 2.0 behavioral state machine, including discussions about detailed semantics of each feature and the ambiguity statements. They also has a separate paper [28], which discussed 29 undeclared points in UML2.0 state machine specifications. This approach covers almost all features of UML2.0 state machines, except for junction and choice vertices, which are considered as syntax sugar and are said can be easily represented by separate transitions. Termination pseudostates and completion events are also left out unconsidered. Syntax of each construct in UML state machine is represented by tuples capturing the components of each construct. The hierarchy of states is captured by the path/name of the states, thus prefixing operation on strings is used to decide whether a state is enclosed within a region and vice versa. The semantics are defined in the form of rules/operations on states or transitions along the process of a run-to-completion step. configurations/history configurations and functions compute successor configurations/history configurations are defined. They are capable to enumerate a run-to-completion step of UML state machine move. An auxiliary function is defined to collect all actions generated in that run-to-completion step and put them in the event pool. The first contribution of [37] is the semantics which covers most UML state machine constructs. But they contributes more on the analysis of the whole procedure of defining formal semantics and the detailed discussion about the semantics of UML 2.0. In term of the formal semantics they had defined, though achieves a high coverage, share the same defect with their other work [27], i.e. there are no final semantic steps to model the execution of the whole state machine and the complex formulas and symbols presented in the papers [37, 27] are hard to follow.

Jori and Tommi[25] proposed a similar approach with [37] in the form of semantic formalization. But less semantic variation points are considered. Instead, the authors prefer to fix some semantic variation point, for example, fix the event pool as a queue. In terms of supported features, choice pseudostates, which is not considered by most approaches, is discussed here. But other commonly considered constructs, such as history pseudostate, is not included in their formalization.

Dániel [53] provide a different approach from all the previously mentioned work, which is based on a combination of metamodeling and graph transformation. The formalization is rule-based, visual specification of sate machine semantics by means of model transition systems, which organizes all the model transformation rules in a control flow graph(CFG) like control structure. They also use EHA as an intermediate representation, as is done in [44] and modular model transformation rules are provided based on dynamic semantic steps(along the procedure of execute transitions) of an UML state machine. At last, a model transition system, which organize all the model transformation rules with an explicit CFG, is provided as the semantic domain of the UML state machine. Another contribution of this paper is that the author encourages the separation of dynamic attributes from static attributes of UML state machine. Conflicts and priorities are considered derived static semantic concepts of UML state machines in this paper. The separation of static with dynamic attributes may abstract away from diagram specific features and focus on the dynamic behavior, which is the core of semantics, of UML state machine.

2.5 Summary

In addition to all the papers surveyed above, Crane and Dingel [23] provide a comprehensive survey which covers some approaches we have survey here. They also categorize all the surveyed approaches based on the underlying formalism of the approaches. But we are focusing on those approaches which are related to automatic verification of UML state machine. So our survey is more problem-specific compared to [23]. Among all the surveyed approaches, approaches using ASM as semantic domain tend to support more UML state machine features than the other approaches but are less related to automatic verification. LTS-based approaches are most related to automatic verification, specifically model checking, but always support less features compared to other approaches due to the unstructured feature of UML state machine. We would like to fill the gap by supporting more UML state machine features while using LTS as the formal semantic domain in our formalization, so that model checking UML state machine can be properly handled. Table 1 provides the summarization of all the surveyed approaches in this section and discussed the corresponding advantages and disadvantages of the corresponding semantic domain.

Semantic Domain	Paper	Advantage	Disadvantage
LTS	[44], [54], [43]	Naturally coincide with Model checking tools	Hard to express the complex semantics of UML state machine
ASM	[15], [16], [18], [22], [39]	Suitable to express the complex semantics of UML state machine	Few automatic tool support
Petri Nets	[20], [31], [12], [11]	Rigorous mathematical reasoning support	Hard to understand and use
Others	[27], [47], [46], [25], [37], [53], [25]	Flexible formalization procedure	Less formal proof of the rigorousness of the underlying semantic domain used

Table 1: categorization of formalization approaches

3 The Translating Approach

There are another branch of efforts which try to automatically verify UML state machine by translating UML state machine into a description language of a Model checker such as Spin, SMV, UPPAAL and then model checking various properties on the translated model. [14] provide a good survey on this kind of approaches. But it focuses on many variance of Harel statechart [33, 35, 34], such as RSML, UML. Our focus is just on UML state machine, which is the object oriented variances of Harel statechart. In this section, we are going to have a thorough survey on this kind of approaches according to the Model Checker's input language they adopted.

3.1 Approaches using Spin

Latella et. al are among the first few researchers who contribute to the formal verification of UML state machines. They utilize Extended Hierarchical Automaton(EHA) as an intermediate representation of UML state machine, then they define formal semantics of EHA with Kripke structure as the semantics domain [44]. Based on this formalization work [44], they proceed one step further to provide translations from UM-L state machine diagrams to PROMELA, the input language of SPIN model checker in [45]. The translation function takes an hierarchical automaton as input and generates PROMELA code as output. This approach uses STEP PROMELA process to simulate a run to completion step in UML state machine, which includes dispatch of events from the environment; identify candidate transitions to fire; solve conflicts and select firable transitions; actual execution of the selected transitions. The last step in the previous procedure includes identifying the next configuration after execution of the current transition and maybe side effects, which are events generated during the execution of actions associated with the transition. The run to completion step in UML state machine is, as indicated by the name itself, non-interruptable(but can be stopped[1].). This is guranteed by the PROMELA atomic command. The translation process is structured since it is based on the pre-defined formal semantics of EHA [44]. The authors also provides proof for the translation to guarantee the correctness of the procedure.

[40] is another approach to translate UML state machine into PROMELA. It supports initial, choice pseudo states as well as deferred events and completion events. It further provides an action language, a subset of the Jumbala [26] action language, which

[1]The difference between interrupt and stop is that, interrupt means stop temporally and need to be resumed afterwards. But stop means permanently stop without resuming

is used to specify guard constraints and the effects of transitions of a UML state machine. They also implement a tool called PROCO, which takes a UML model in the form of XMI files and out put a PROMELA model.

[42, 41] provide a method to model checking UML state machines as well as collaborations with the other UML diagrams. They compile UML state machines into a PROMELA model and collaborations into sets of Büchi automata and then invoke the SPIN model checker to verify the model against the automata. Each state in the state machine is mapped to an individual PROMELA process. Two additional PROMELA processes are generated to handle event dispatching and transitions. The event queue is modeled as buffered channels and communication among processes are models via unbuffered channels, i.e. they are synchronized. This approach further considers the consistencies between UML diagrams, i.e. collaboration diagram and state machine diagram. The possible communications among objects shown in a collaboration diagram should be consistent with the dynamic behavior represented in the state machine diagram. By translate collaboration diagram into sets of Büchi automata, which is the form of property to be checked against the model, this approach cleverly checked the consistencies between the two diagrams.

3.2 Approaches using SMV

[43] formalize UML 1.3 statecharts semantics by rule-rewriting systems and provides a translation approach from the formalized semantics to SMV model checker. Although no detailed implementation is discussed in this paper, we assume it is possible to build such a front end since the translation procedure is described.

[25] first provide an symbolic encoding(formal semantics) for a UML state machine, which has been discussed in Section 2.4. Then they performed a translation from the defined semantics to the input language of NuSMV [21] model checker. The detailed translation steps are not discussed in the paper, but some experiment results are reported.

[22] is another approach which uses SMV as the back end model checker to automatically verify UML state machines. Kein et al. [22] first defines the formal semantics with ASM as semantic domains for UML state machine, which has been discussed in Section 2.2. Then an SMV model checker which is based on SMV model checker is invoked to verify the SMV specification of a UML state machine. This approach is different from the other translations approaches in that it does not provide a translation from (some

form of formalization of a) UML state machine to the input language of a model checker. Instead, it relies on a translation tool from ASM to SMV [24].

[13] also provides a translation from UML diagrams to the input language of SMV model checker. Instead of focusing on just UML state machines, it focuses on the collaborations of different UML diagrams such as class diagrams, state machine diagrams and activity diagrams. This paper does not describe the detailed translation rules, but instead, illustrate their translation approach with an ATM machine example. Noticing that high-level model designers are unfamiliar with LTL and CTL properties which are used by model checkers, the authors also provide some aid in the form of ask and answer questions to aid the property writing.

3.3 Approaches using other Model Checkers

[30] is another translation approach, which is also based on the formalization of UML state machine in their early work [44]. The translation is from a hierarchical automaton into semantic automaton(LTS), which need to be further translated into FC2 format, which is the standard input format to Jack. But in this case, the FC2 format is nothing but a LTS.

Shaojie and Yang [55] provide a translation approach which translate UML state machines into CSP#, an extension of CSP language, which serves as the input modeling language of PAT [50]. Different from other translation approaches, the transformation is not based on a pre-defined formal semantics, but directly from the meaning of each UML state machine construct. But this approach provides experiment results of the verification of UML state machine with PAT [50].

[42] present an approach to translate timed UML state machines into timed automata which is used by the UPPAAL Model Checker. But the translation is not based on a formal semantics of timed UML state machines. Event queue and UML state machine are separatedly modeled by timed automata and the communication is modeled with a channel. This approach is implemented in a prototype tool, HUGO/RT, which can verify whether scenarios specified by ML collaborations with time constraints are consistent with the corresponding set of timed UML state machines.

3.4 Summary

The translation approaches aim at utilizing the automatic verification ability of different model checkers. So the advantage of these approaches is that most of them will provide the translation rules as well as the implementation of those rules in the corresponding model checkers. But we notice that translation based approaches suffers form four drawbacks. Firstly, it is hard to link back the original model if a bug is detected in the translated model. Secondly, the translation may introduce more redundant behaviors which may slow down the verification process. Thirdly limited by the translated language, most approaches consider quite restrictive subsets of the UML syntax defined by OMG [6]. Lastly and the most importantly, it is hard to verify whether the translation procedure is correct since it is done in an unsystematic and informal way, thus cannot guarantee the correctness of the verification model and the verification result. So we are going to build our tool directly based on the operational semantics we defined for UML state machine.

4 Current Tool Support

There are commercial as well as academic tool supports for UML modeling. Current Commercial tools just support the design of UML models. Some academic prototype tools were developed based on the translation approaches, which aim at automatic verification of UML state machine, as we surveyed in Section 3. We are going to survey these tools in this section and concentrate mainly on those automatic verification tools.

4.1 Static Checking and Graphical Editing Tools

There are many commercial company concentrating on the UML modeling language and provide tool support for graphically designing UML diagrams. A lot of software tools(34 as categorized by wiki[10], out of which 18 are free) have been developed to meet the requirement of model-driven development. IBM Rational Rhapsody, Microsoft Visual Modeler, Papyrus are representative for this kind of software. Rational Rhapsody and Microsoft Visual Modeler are commercial software and Papyrus is free software. We briefly introduce the three software since they are representative in functionality among all the 34 tools.

IBM Rational Rhapsody [1] is a visual model developing environment based on UML, which supports developing of embedded system, real-time systems. It also support automatically generating of software applications from the graphical model in various languages such as C, C++, C#, Ada and Java. Some static checking functionality such as consistency checking is also supported.

Microsoft Visual Modeler [3] provides visual modeling of class and component diagrams based on a subset of UML, including class diagram, component diagram, activity diagram, use case diagram and sequence diagram. It is integrated with Microsoft Visual basic and Visual C++ to support generation of basic and C++ code automatically. It also support a reverse generation, i.e. from C++ or basic code to class and component diagrams. But UML state machine diagram is not supported.

Papyrus [8] is an open source tool for UML2 graphical modeling, and has now become an eclipse plug-in project [7]. Papyrus aims at providing full support for OMG UML specifications.

Though there are many UML developing tools, they just support graphically modeling and a little static checking, such as syntax checking and consistency checking, of UML models. Tools which support dynamic checking and simulation are needed.

4.2 Dynamic Behavior Simulation and Property Verification Tools

We surveyed dynamic verification tools for UML diagrams in this subsection. All of these tools are prototype tools based on the translation approaches surveyed in Section 3.

vUML

[47] reports a tool vUML which aims at automatically verify UML model behaviors specified by UML statecharts diagrams. This tool utilizes SPIN model checker as a backend to perform model checking and creates a UML sequence diagram according to the counter example provided by SPIN. The formal semantics(supporting UML 1.3) is defined in [47] and they also conduct a case study with the production cell example in [46].

vUML aims at checking collaborations of UML models instead of a single UML state machine. So the PROMELA specification for a UML model is generated from class diagrams, statecharts and collaboration diagrams, where each UML class is mapped into a PROMELA process-type, each UML object is mapped into a PROMELA process and each link in the collaboration diagram is converted into a PROMELA channel for objects to exchange messages. vUML provide an event generator to emulate external events without parameters(close models) and remove external events carrying parameters in order to avoid state space explosion.

vUML can check the following properties: deadlock, livelock, reaching an invalid state, violating a constraint on an object, sending an event to a terminated object, overrunning the input queue of an object, overrunning the deferred event queue.

In order to verify those properties, two extra stereotypes are introduces, namely <<invalid>> and <<progress>>. UML model developers need to add those extra stereotypes into their models in order to use vUML to model check safety and liveness properties. Constraints also act as a way to specify properties(invariants over attributes and states). vUML cannot verify LTL formulas due to unavailable to express a LTL formula in a UML diagram.

[47] has developed such as tool called vUML, which make use of Spin model checker. It first transform a UML statecharts into Promela, which is the modeling language used by the Spin model checker. Then, model checking is done by the Spin model checker. Whenever a counter example is provided by the Spin model checker, vUML will transform the counter example back to a UML sequence charts and report to the user.

HUGO

HUGO [42] is a tool which contains three components, each of which supports one functionality. The first is the code generation component, which is used to automatically generate Java code from UML state machines. The second is model checking component, used to verify the consistency of UML state machines against specifications expressed as collaboration or sequence diagrams. HUGO can support LTL model checking provided knowledge of the underlying model checker(SPIN) and the structure of the translation. The third component is a back end for the real time model checker UPPAAL.

HUGO also adopts the translation approach where it translates UML state machines into PROMELA models and utilize the SPIN model checker to do the verification(the same applies to the real-time components where the UML state machines are translated into the UPPAAL modeling language). HUGO is developed based on UML version 1.4.

SMV-based Verification Tool

[49] introduced a tool based on ASM model checker(which is based on SMV model checker). The semantics they adopted is defined in [22](UMLversion1.3 or early, though not explicitly mentioned in their technical report). This tool set supports both static(structural) and dynamic(behavioral) check of a UML diagrams. The static checking handles class diagrams and object diagrams, while the dynamic checking deals with statechart diagrams. This tool takes UML diagrams specified in XMI as input and output a counter example in the form given by the SMV model checker(since the dynamic checking component of this toolset is based on the SMV model checker). The counter example trace can be fed to their analysis tool, which will analysis of the error trance and produce some UML diagram such as sequence diagram or a collaboration diagram to the users. In contrast to vUML [47] and HUGO [42] which proposed a translation approach, [49] formalizes UML semantics in ASM and utilizes ASM as both the semantic and validation model. Then a ASM model checker(based on SMV model checker) is directly used to model checking the dynamic properties of the model. In this way, they guarantees that the model is correctly defined and the verification result reflect the real problem in the model.

TABU

[13] introduced a tool called TABU(Tool for the Active Behavior of UML). TABU takes UML diagrams(activity and state diagrams) in the form of XMI as input, automatically

translate it into .smv representation and call the Cadence SMV Model Checker to verify the UML model. In addition, TABU also provides assistant for writing (LTL/CTL) properties to verify against the model. This feature makes the underlying model and the translation procedure transparent to the users, and solved the problem faced by vUML [47] to some extend.

This tool is attractive in the sense that it deals with UML 2.0 specifications, which is much closer to recent UML standard. The translation covers most UML features(though not described in detail in this paper) except for synchronization states, events with parameters and dynamic creation and destruction of objects. It also provides guides in writing properties. But it is still a translation approach and suffers all the common defects of translation approaches. Further, the counter example is given in the form of SMV model checker, which is not intuitive for model designers to map to their models.

JACK

[30] provide an algorithm to support direct model checking UML statecharts(v1.1) based on the formal semantics they have defined in [44]. The implementation is based on the tool set JACK[17], which is an environment based on the use of process algebras, automata and temporal logic formalism and supports many phases of the system development process by integrating different editing tools and verification tools. Different components of the JACK tool set communicate with the FC2 format. There is a model checking tool in the JACK tool set named AMC, which supports ACTL model checking. The system should be translated into the FC2 format first in order to utilize the AMC component. The user also need to specify their own ACTL property according to the model. This requires users to have a knowledge of model checking, the underlying model as well as temporal logic formulas.

PROCO

Another tool which translate a UML state machine in the form of XMI forms into PROMELA, the input language of Spin model checker is discussed in [40], which has been discussed in section 3.

4.3 Summary

We notice that all the existing tools just provide a front-end supporting translation from UML state machine to languages of model checkers. Such a translation will introduce extra cost for the verification procedure. Due to the limitation of input languages to model checkers, the complex semantics of UML state machine cannot be fully supported. There are also problems for the utility of those tool, i.e. it is hard to map the found vulnerabilities to the original model. The informal translation procedure cannot guarantee the soundness of the obtained model either. In order to conquer all the above weaknesses, we are motivated to develop a tool which support direct model checking of UML state machine diagrams.

5 Analysis of UML state machine specifications

In this section, we are going to thoroughly analyze the basic constructs of UML behavioral state machine specifications as documented in [6, Chapter 15]. We are also going to briefly describe UML state machine features. The syntax and semantics, which are based on the discussion here, will be defined formally in Section 6. The description is according to OMG UML2.4.1 Superstructure Specification [6, Chapter 15]. We also refer to [11] as another important reference since it is the most recent related work and it covers a lot of features of UML 2.2 behavioral state machine. We analyze three kinds of basic constructs in UML behavioral state machine, i.e. state, pseudostate and transition. We selectively cite some parts of the original descriptions in OMG UML 2.4.1 Super Structure Specification [6, Chapter 15] for better clarity.

5.1 States

States in a UML behavior state machine have three types, i.e. simple state, composite state and submachine state. Composite state is further divided into orthogonal composite state and composite state depending on whether it has exactly one or more than one regions.

"A Submachine state is semantically equivalent to a composite state."

[6, Chapter 15.3.11, Section Description, Submachine state, p.560]

Thus we can convert a submachine state into a composite state easily in the formalization and verification procedure. Currently we support simple state and composite state (including orthogonal composite state). In our continued work, we plan to support submachine state since it can be converted to a composite state.

Final state is a special kind of state. It does not have regions, entry/exit behaviors or do activities. But it still belongs to the class of state, since it has a fundamental difference with pseudostate, viz an object can temporally "stay"[2] in a final state while it cannot "stay" in any of the pseudostate.

5.2 Pseudostate

Pseudostates are introduced to connect transitions to form compound transitions which have a more complete semantic meaning, or to aid the construction of a state machine to improve the expressiveness of it. There are 10 kinds of pseudostates specified in [6, Chapter 15.3.8] and they are described bellow. We group some pseudostates which share

[2]waiting for some event to happen or waiting for some processing procedure to be completed

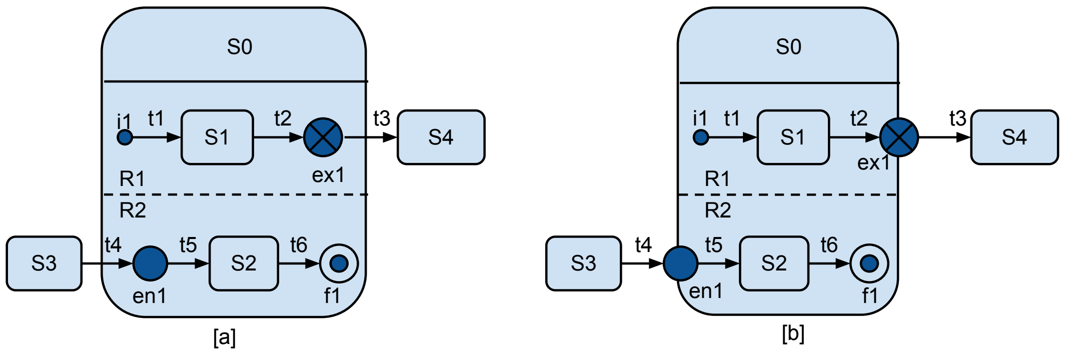


Figure 1: Entry and Exit Vertex Example

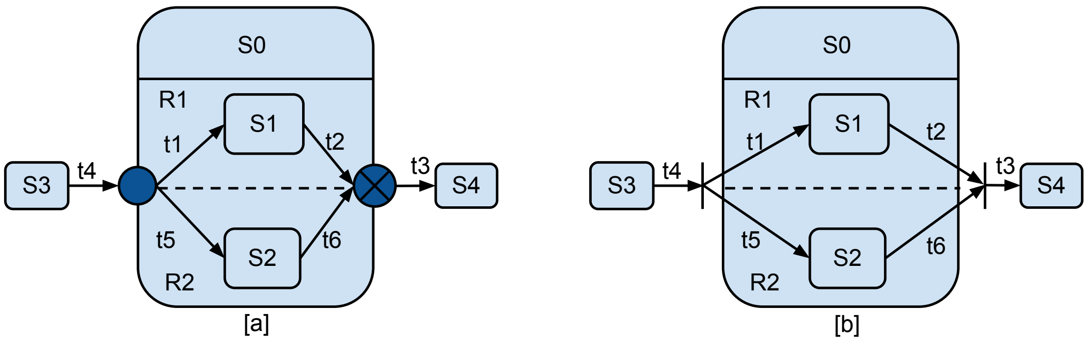


Figure 2: Entry and Exit Vertex in Orthogonal Composite Sate

similar specifications(entry point and exit point, shallow and deep history, fork and join) together.

- Junction pseudostate is a syntax free vertex and is introduced as syntactic sugar to merge/split incoming transitions into outgoing transitions sharing the same transition path, i.e. guard condition, target vertex.

  > "junction vertices are semantic-free vertices that are used to chain together multiple transitions."
  >
  > [6, Chapter 15.3.8, Section Semantics, p.551]

- Entry point (resp. exit point) pseudostate is a way to explicitly indicate the execution of entry (resp. exit) behavior. On entering a composite state, it can also play the role of a junction or fork (resp. join) pseudostate.

  > "An entry point is equivalent with a junction pseudostate (fork in case the composite state is orthogonal)"
  >
  > [6, Chapter 15.3.11, Section Semantics, Submachine state, p.565]

But the direct ancestor of an entry (resp exit) pseudostate is not clear.

> "An entry point pseudostate is an entry point of a state machine or composite state"
>
> [6, Chapter 15.3.8, Section Semantics, p.551]

Which implies that an entry point belongs to a composite state, at least will represent the scope of a composite state. However:

> "Entering an exit point within any region of the composite state or s-tate machine referenced by a submachine state implies the exit of this composite state or submachine state, etc."
>
> [6, Chapter 15.3.8, Section Semantics, p.551]

This seems to imply that an exit point pseudostate belongs to a region. For example, There is no difference between Figure 1[a] and Figure 1[b]. In both figure, the exit vertex "graphically" belongs to region $R1$ and the entry vertex "graphically" belongs to the region $R2$, but entering the entry vertex means entering the composite state $S0$, i.e., both region $R1$ and $R2$. The same rule applies to the exit vertex. Another possible scenario is shown in Figure 2[a]. In this case, the entry (resp. exit) point pseudostate belong to the composite state.

Since in UML state machine specification [6, Chapter 15], the description of s-tate [6, Chapter 15.3.11, Section Associations, p561] and state machine [6, Chapter 15.3.12, Section Associations, p573] both have associations named connection-Point, which refers to entry/exit pseudostates. But in the description of region [6, Chapter 15.3.10, Section Associations, p557], there is no such associations. So we will follow this description and make the entry/exit point pseudostate associated with a state or a state machine.

Further, transition emanates from an entry vertex cannot target a join vertex, as restricted by "...In each region of the state machine or composite state it has at most a single transition to a vertex within the same region" [6, Chapter 15.3.8, Section Semantics, p.551]. Since the target vertex should be in the same region with the entry vertex, a join vertex will not satisfy such a constrain.

One of the benefit of introducing entry/exit point pseudostate is for the purpose of using submachine state, which need a mechanism to explicitly indicate the entrance and exit of it. Another benefit of entry and exit vertices is the fine-grained behavior granularity. For example in Figure 1, without the exit vertex, the action sequence of the run-to-completion step starts from $S1$ and ends in $S4$ will be $\mathcal{E}xf(S0)$; $t2.\alpha$; $t3.\alpha$; $\mathcal{E}nf(S4)$. However, with the exit vertex, the action sequence will be $t2.\alpha$; $\mathcal{E}xf(S0)$; $t3.\alpha$; $\mathcal{E}nf(S4)$.

Another possible scenario is shown in Figure 2, the only difference between 2[a] and 2[b] is that, in 2[a],the entry behavior of state S_0 will be executed before the actions associated with transition $t1, t5$, but in 2[b], the order is reversed. This observation is obtained from:

> "An entry point is equivalent with a junction pseudostate (fork in case the composite state is orthogonal): Entering via an entry point implies that the entry behavior of the composite state is executed, followed by the (partial) transition(s) from the entry point to the target state(s) within the composite state."
>
> [6, Chapter 15.3.11, Section Semantics, Submachine state, p.565]

- join vertices:

> "Join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions, etc., fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices."
>
> [6, Chapter 15.3.8, Section Semantics, p.551]

So join (resp. fork) pseudostate is used to represent execution of transitions from (resp. to) vertexes in orthogonal regions simultaneously.

> "The transitions entering a join vertex cannot have guards or triggers, etc. The segments outgoing from a fork vertex must not have guards or triggers"
>
> [6, Chapter 15.3.8, Section Semantics, p.551]

Guarantees that transitions emanating from (resp. ending in) a fork (resp. join) vertex can be executed simultaneously. They are different from the other pseudostates in that they are emanating (targeting) a set of vertices instead of one vertex.

> "transitions outgoing a fork vertex must target states in different regions of an orthogonal state"
>
> [6, Chapter 15.3.8, Section Constraints, p.550]

So the target states of a fork vertex must be states(as opposed to pseudostate). But we do not see such constraints on a join transition.

- Initial state is used to indicate the default initial state (not pseudostate) of a region of a composite state, it cannot act as target of a transition. Further constraint is as follows:

  > "The outgoing transition from the initial vertex may have a behavior, but not a trigger or guard"
  >
  > [6, Chapter 15.3.8, Section Semantics, p.551]

- History pseudostate is a mechanism for the state machine to remember the previous "snapshot" of its containing state. History pseudostate is further divided into shallow history and deep history pseudostate.

  > " deepHistory represents the most recent active configuration of the composite state that directly contains this pseudostate(e.g. the state configuration that was active when the composite state was last exited)... shallowHistory represents the most recent active substate of its containing state(but not the substates of that substate)"
  >
  > [6, Chapter 15.3.8, Section Semantics, p.551]

  Shallow (resp. deep) history pseudostates indicates the last active substate (resp. configuration) of its containing state. It can act as both source and target of a transition. Transition emanating from a history pseudostate indicate the default history (in case a history pseudostate is reached but the containing state has never been activated or the last active substate was a final state). They are still different from the other pseudostates since they have an extra "invisible transition", which is the recorded last active state within the containing composite state and is changing during execution.

- Choice pseudostate, though also connects transitions to form compound transitions, has its guard on the transitions emanating from it evaluated dynamically, which means that the guard evaluation of transitions emanating from the choice pseudostate may depend on the execution of the previous transitions and cannot be decided until the choice pseudostate is reached. In this aspect, choice pseudostate more resembles states instead of pseudostate. Theoretically, transitions emanating from a choice vertex can end in any vertex (except for initial vertex). But we forbid transitions emanating from choice vertex to end in a join vertex since join vertex will involve more than one region and the choice vertex may cause ambiguity. For example in Figure 3, if the current configuration is $\{S3, S5, S6, S4, S1, S0\}$ and transition $t9, t10$ are enabled, since transitions emanating from a choice vertex should be evaluated during execution, the result of evaluating $t11$ is unknown. If the later evaluation of $t11$ is true, then the best case happens and we proceed with the current "routine". If, however, transition $t13$ is evaluated to true, then we are in a dilemma, since such a conflict is not specified in the specification. Thus this situation should be avoided, i.e., a transition emanating from a choice

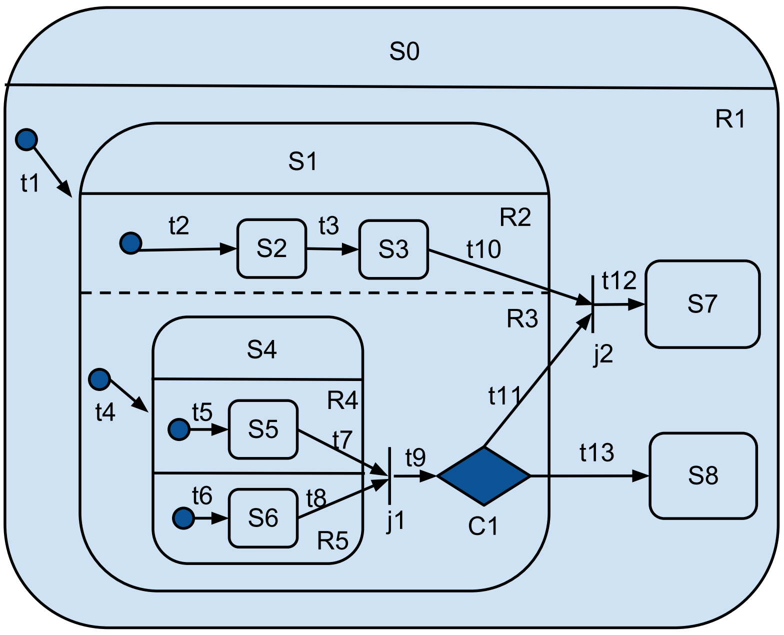


Figure 3: Choice vertex Example

vertex cannot end in a join vertex. We do not forbid the use of junction vertex in this case since all transitions of a junction vertex are evaluated statically (before execution), no ambiguity will be caused. Another point we should mention about the choice vertex is that instead of evaluate one single transition emanating from a choice vertex, we should evaluate the sequence transition with the first of it emanates from the choice vertex. For example in Figure 3, we should evaluate sequence transition $t11.t12$ instead of $t11$ alone in the choice vertex $C1$.

- Entering a terminate pseudostate represents the termination of object which was active on the current state machine. Without exiting any states nor executing any exit actions, the state machine terminates immediately.

5.3 Transitions

"A transition is a direct relationship between a source vertex and a target vertex."

[6, Chapter 15.3.14, Section Description, p.581]

So a transition can emanate and target a pseudostate as well.

" A compound transition is a derived semantic concept, represents a "semantically complete" path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states"

[6, Chapter 15.3.14, Section Semantics, p.583]

So a compound transition can be composed of a series of transitions with more than one pseudostates inbetween the source and target states. To the best of our knowledge, no work in the literature has considered compound transitions. We illustrate a compound transition in Figure 4.

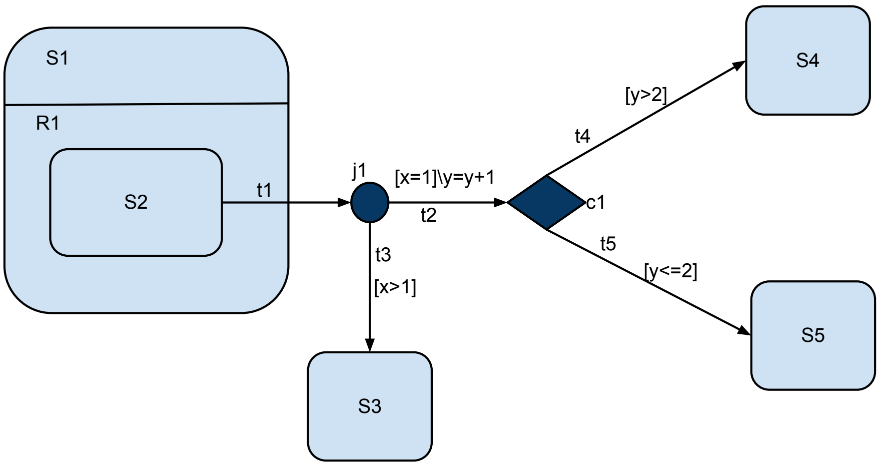


Figure 4: Multiple pseudostates in a compound transition

As is shown in the figure 4, we can have a compound transition connected by a junction and choice pseudostate. Actually, according to the definition of a compound transition, we can have a compound transition connected by any number of junction, choice and join pseudostates with one fork pseudostate (since fork pseudostate can only target states in orthogonal regions).

Triggers, guards and effects are associations of a transition. We use *Trigger*, G and B to represents the set of all possible triggers, guards and effects separately. In our current settings, we do not consider time-event as triggers.

We have listed all possible source and target vertex of a transition we considered in Table 2. As is shown, transitions emanating from initial, history and fork vertex can only target states. Initial vertex cannot act as target of transitions. transitions emanating from a choice or entry point vertex cannot target a join vertex as has been explained in Subsection 5.2.

Source	Target	Remarks
S	$S \cup S_f \cup PS \backslash I$	initial vertex cannot act as targets
I	$S \cup S_f$	the semantics of initial vertex is to indicate the default state of a composite state instead of other vertex
$H^* \cup H$	$S \cup S_f$	history pseudostates represents the most recent active substate (shallow history) or configuration (deep history), which is a set of trees of states [6, p.564]
C	$S \cup S_f \cup PS \backslash I \backslash J$	transitions emanating from choice vertex cannot target a join vertex
J	$S \cup S_f \cup PS \backslash I$	a transition emanate from a join vertex can target any vertex except for initial vertex
F	$S \cup S_f$	"transitions outgoing a fork vertex must target states in different regions of an orthogonal state" [6, p.550], so the target states of a fork vertex must be states(as opposed to pseudostate).
$Junc$	$S \cup S_f \cup PS \backslash I$	transitions emanating from a junction vertex and target any vertex except for initial vertex
En	$S \cup S_f \cup PS \backslash I \backslash J$	transition emanates from a entry vertex cannot target a join vertex
Ex	$S \cup S_f \cup PS \backslash I$	transitions emanating from an exit point pseudostate can target any vertex except for initial pseudostate.

Table 2: Valid transitions

Another important point we should notice is the scope of a transition. The scope of a transition, along with the source and target state, indirectly provides the type of a transition, i.e., internal, external or local.The suggested container is the Least Common Ancestor(LCA)[3] of source and target states of the transition:

"The owner of a transition is not explicitly constrained, though the region must be owned directly or indirectly by the owing state machine context. A suggested owner of a transition is the LCA of the source and target vertices"

[6, Chapter 15.3.14, Section Semantics, Compound transitions, p.583]

[3]the smallest common container of the two states.

But it is not sufficient to identify all types of transitions. For example in Figure 5, transition $t3$ would have been mistakenly treated as internal transition since the LCA of its source and target are both $S4$.

6 Formal semantics of UML statecharts–Our Approach

Our work is partially based on the work proposed by Jen Schoñborn et al. [37]. But [37] considers too many possibilities about the ambiguous nature of UML state machine specifications, some of which are unnecessary. Besides, junction, choice and terminate pseudostates are left out in their approach. We argue that even though those pseudostates are considered syntactic sugar, they cannot be easily replaced by the existing transitions, especially choice pseudostate, which hold the meaning that guards are evaluated dynamically inbetween transitions within a run-to-completion step. Furthermore, the formal semantics provided in [37] are not complete, actually they just provide rules to formalize transition steps of UML state machine execution How to model the whole behavior state machine is not discussed. So in our definition of UML state machine semantics, we will consider all of the pseudostates. Further, for those semantic variation points, we will choose to fix those obvious ones to the widely adopted semantics. This will be discussed in the definition of each concrete semantics.

We first introduce two basic sets(types) we defined for the convenience of defining syntax.

Let N_{atom} be a set of state, region, pseudostate and transition names in the UML state machine. In Figure 1[a], the set $N_{atom} = \{S_0, S_1, S_2, S_3, S_4, R_1, R_2, f_1, i_1, en_1, ex_1, t_1, t_2, t_3, t_4, t_5, t_6\}$[4].

Let N be a set of composite names, each of which is composed of a sequence of elements in N_{atom}. The form is elements in N_{atom} connected by dots. For example, in Figure 1[a], the state S_2 will be represented as $R_0.S_0.R_2.S_2$ and the exit point pseudostate will be represented by $R_0.S_1.R_1.en_1$ in the form of composite names.

6.1 Syntax of UML state machine

We define the formal abstract syntax in this subsection. The syntax we defined is based on the description of UML state machine specifications. We used different symbols to represent different domains. The detailed explanation of the symbols is listed in Appendix B.

Definition 1 *A state is defined as a 9-tuple(name, type, subvertex, deffer, entrybehavior, exitbehavior, doactivity, entrypoint, exitpoint) where:*

[4]In [6], the outermost construct is a state machine, which has the same semantics with a composite state except that, it represent a complete object behavior. We can ignore the outermost state (machine) and represent the outermost hierarchy as a default region R_0.

- *name* $\in N$ *is the name of the state which uniquely identifies the state.*
- *type* : $enum\{isSimple, isOrthogonal, isComposite, isSubmachineState\}$ *is the type of the corresponding state.*
- *subvertex* : $\mathbb{P}R$ *is the set of direct containing regions of this state. In case of a simple state, the set is empty.*
- *deffer* : $\mathbb{P}Trigger$ *is the set of defferrable triggering events associated with this state.*
- *entry* : B *is the optional entry behavior of the state. In terms of no entry behavior is defined for the state, this value is set to* ϵ*.*
- *exit* : B *is the optional exit behavior of the state. In terms of no exit behavior is defined for the state, this value is set to* ϵ*.*
- *do* : B *is the optional do behavior of the state. In terms of no do behavior is defined for the state, this value is set to* ϵ*.*
- *entrypoint* : En *is the entry point reference associated with the state. In terms of no entry point reference is defined for the state, this value is set to* ϵ*. entrypoint can only be associated with a composite state.*
- *exitpoint* : Ex *is the entry point reference associated with the state. In terms of no entry point reference is defined for the state, this value is set to* ϵ*. exitpoint can only be associated with a composite state.*

Definition 2 *A region is defined as a 3-tuple(name, subvertex, containingpseudostate) where:*

- *name* $\in N$ *is the name of the region which uniquely identifies the region.*
- *subvertex* : $\mathbb{P}(S \cup S_f)$ *is the set of direct containing states of this region.*
- *containingpseudostate* : $\mathbb{P}PS$ *is the set of pseudostates contained in this region.*

Definition 3 *A pseudo state is defined as tuple (name, type) where:*

- *name* $\in N$ *is the name of the pseudostate which uniquely identifies the pseudostate.*
- *type* : $enum\{I, T, En, Ex, J, F, Junc, C, H^*, H\}$ *is the type of the corresponding pseudostate.*

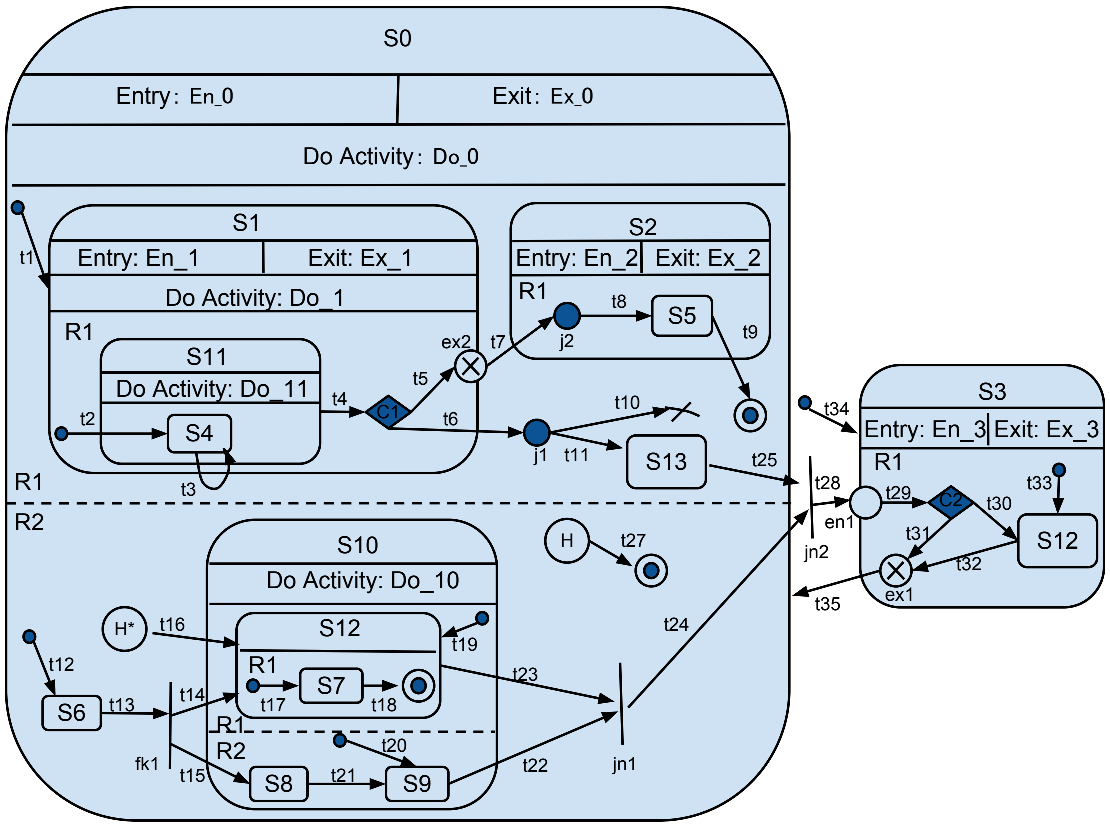


Figure 5: Example of a UML behavior state machine

Definition 4 *Final state is a special kind of state, which is defined as a tuple (name, container)*

where:

- *name* $\in N$ *is the name of the final state which uniquely identifies the finalstate*
- *container* : R *is the direct container of the final state.*

Definition 5 *A transition tr is defined as a 9-tuple(name,* $\widehat{S}$, ε, φ, α , $\widehat{T}$, ι, $\widehat{F}$, $\widehat{J}$*)*

where:

- *name* $\in N$ *is the name of the transition which uniquely identifies the transition.*
- $\widehat{S} \in Src$ *is the source state (in the case of a join transition will it be the join pseudostate).*
- $\widehat{T} \in Trg$ *is the target state (in the case of a fork transition will it be the fork pseudostate).*
- $\varepsilon \subset Trigger$ *is the set of triggers, and it can be an empty set.*

- *$\varphi \in G$ represents the guard of the transition. It is a boolean expression of triggers, if there is no guard for a given transition, this field is set to true by default.*

- *$\alpha \in B$ is the effect behavior to be performed when the transition fires. A transition can have no effect behavior, in such a case, this field is set to be ϵ*[5]*.*

- *ι is the scope of a transition.*

- *$\widehat{F}$ is a tuple of the form (fn, TAP), where $fn \in F$ is the name of the fork pseudostate and $TAP \triangleq \{(n, trg, eff, \iota) | trg \in \widehat{T}, eff \in B, \iota \in S \backslash S_{simp} \bigcup R\}$. n is the name and unique identifier of of one of the transitions emanating from the fork vertex. trg is the target vertex of the transitions. eff is the effect associated with the transition and ι is the scope of the transition. If a transition is not a fork transition, this field is set to $(\epsilon, \epsilon, \phi, \epsilon)$ by default. For notation convenience, we will use the single transition ending in the fork vertex to represent the fork transition and this transition is referred to as the main transition. For example, in Figure 5, $t13$ alone will represent the fork transition $t13(t14 \mid\mid t15)$ and is called the main transition.*

- *$\widehat{J}$ is defined as a set of tuples of the form (jn, SAP) where $jn \in J$ is the name of the join pseudostate. $SAP \triangleq \{(n, src, eff, \iota) | src \in \widehat{S}, eff \in B, \iota \in S \backslash S_{simp} \bigcup R\}$. n is the name and unique identifier of of one of the transitions emanating from the fork vertex. src is the source vertex of the transition. eff is the effect associated with the transition and ι is the scope of the transition. If a transition is not a join transition, this field is set to $(\epsilon, \epsilon, \phi, \epsilon)$ by default. For notation convenience, we will use the single transition emanating the join vertex to represent the join transition and this transition is referred to as the main transition. For example, in Figure 5, $t24$ alone will represent the join transition $(t22 \mid\mid t23)t24$ and is called the main transition.*

In the definition of Transition, we explicitly note join and fork pseudostate and the associated transitions but leave the other pseudostates alone. Our consideration for this is that fork and join pseudostates are special since they require a parallel semantics, i.e., a fork pseudostate requires the transitions emanating from the fork pseudostate to fire simultaneously. Such semantics will be very troublesome to capture if we do not explicitly note them. For the convenience of later usage, we explicitly note every field of a transition as follows. For $\forall\, t \in Tr$, its subdomain are denoted as follows:

[5] *Note that ϵ should not be in Trigger or B since we use it to represent empty action here.*

- $t.n$ is the name of transition t.
- $t.\widehat{S}$ is the source state of transition t.
- $t.\widehat{T}$ is the target state of transition t.
- $t.\epsilon$ is the set of triggering events of transition t.
- $t.\varphi$ is the guard of transition t.
- $t.\alpha$ is the action associated with transition t.
- $t.\iota$ is the scope of transition t.
- $t.\widehat{F}$ refers to the pair $(\mathit{fn}, \mathit{TAP})$.
- $t.\widehat{J}$ refers to the pair $(\mathit{jn}, \mathit{SAP})$.

Further, we define function Isfork: $\mathit{Tr} \rightarrow \mathbb{B}$ check whether a transition is a fork transition[6].

$$\mathit{Isfork}(t) = \begin{cases} T, & \mathit{if} t.\widehat{F} \neq (\varepsilon, \epsilon, \phi, \varepsilon) \\ F, & \mathit{otherwise} \end{cases}$$

Function Isjoin: $\mathit{Tr} \rightarrow \mathbb{B}$ check whether a transition is a join transition[7].

$$\mathit{Isjoin}(t) = \begin{cases} T, & \mathit{if} t.\widehat{J} \neq (\varepsilon, \epsilon, \phi, \varepsilon) \\ F, & \mathit{otherwise} \end{cases}$$

Function main: $\mathit{Tr} \rightarrow \mathit{Tr}$ will return the main transition of a fork (resp. join) transition. Formally, $\mathit{main}(t) = t'$, where $t'.n \doteq t.n, t'.\widehat{S} \doteq t.\widehat{S}, t'.\widehat{T} \doteq t.\widehat{T}, t'.\epsilon \doteq t.\epsilon, t'.\varphi \doteq t.\varphi, t'.\alpha \doteq t.\alpha, t'.\iota \doteq t.\iota, t'.\widehat{F} \doteq (\varepsilon, \epsilon, \phi, \varepsilon), t'.\widehat{J} \doteq (\varepsilon, \epsilon, \phi, \varepsilon)$. Where $\doteq$ means assign a value to the left hand side formula.

As an example, we consider transition $t7$, $t1$ and $t2$ in Figure 6, their syntactic representations are shown in Table 3.

transition name	syntax representation
$t1$	$(t1, S1, f1, \{e1\}, g1, a1, R0, (f1, \{(t3, S2, a3, R0), (t4, S3, a4, R0)\}), (\epsilon, \phi))$
$t2$	$(t2, j1, S4, \{e2\}, g2, a2, R0, (\epsilon, \phi), (j1, \{(t5, S2, a5, R0), (t6, S3, a6, R0)\}))$
$t7$	$(t7, \mathit{init}1, S1, e7, g7, a7, R0, (\epsilon, \phi), (\epsilon, \phi))$

Table 3: representation of transitions

[6]The compound transition connected by a fork pseudostate.
[7]The compound transition connected by a join pseudostate.

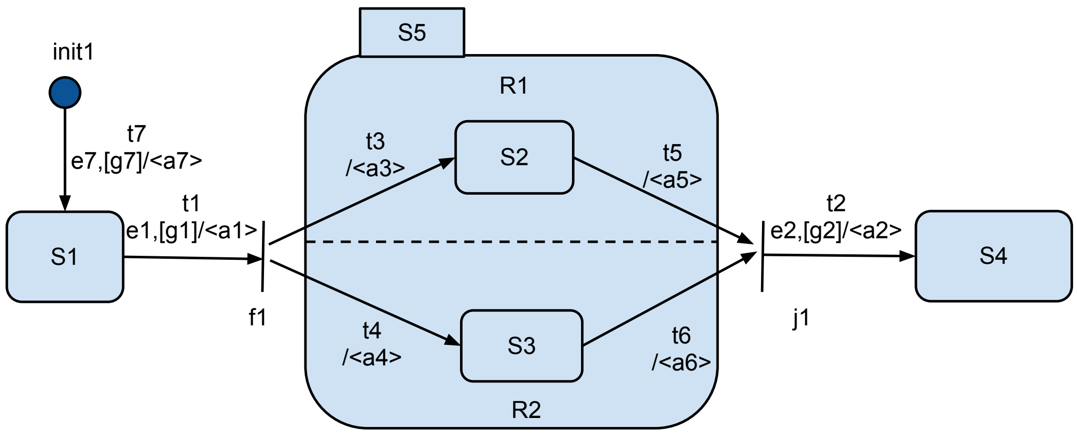


Figure 6: The syntax of transitions

Definition 6 (State machine) *A UML state machine M is defined by a 5-tuple (S, S_f, PS, R, Tr),*
where

- *S is the set of states contained in the state machine.*
- *S_f is the set of final states contained in the state machine.*
- *PS is the set of pseudo states contained in the state machine.*
- *R is the set of regions contained in the state machine.*
- *$Tr : S \cup PS \backslash T \rightarrow S \cup S_f \cup PS \backslash I$ is the set of all transitions tr.*

In our syntax definition, we try to follow the UML behavior state machine specification as much as possible. We keep the state hierarchy in the definition of states with the field of *subvertex*. States, regions and pseudostates hierarchy can all be dealt this way. In a UML state machine, only the inter level[8] transitions break the hierarchical structure. So we do not include transitions into state hierarchies as opposed to the EHA approaches proposed by [44, 54]. In this way, we keep the hierarchical structure of a UML state machine tidy and at the same time obey OMG UML behavior state machine specification as much as possible compared to [37]. For example the state machine in Figure 7 is represented in Table 4.

[8]transitions which cross the boundary of states

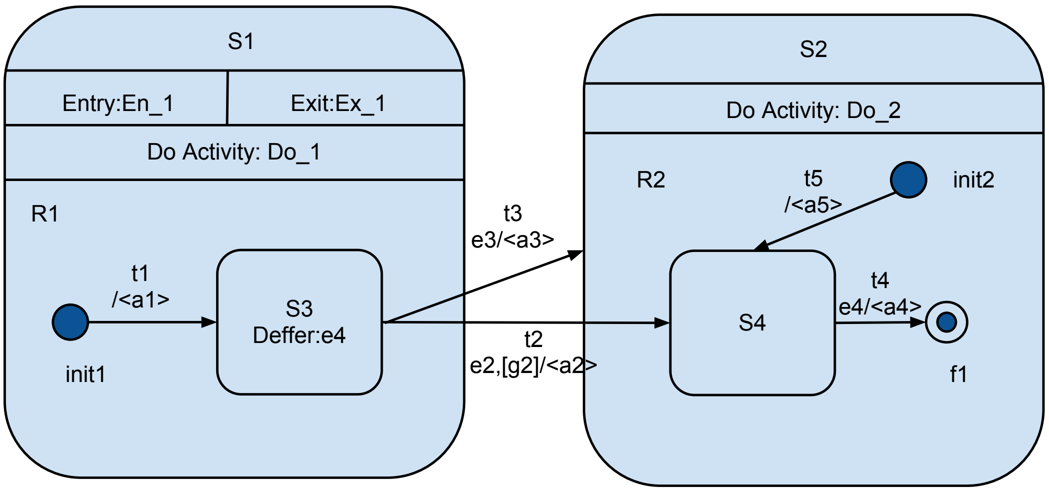


Figure 7: The syntax of a state machine illustration

6.2 Semantics of UML state machine

We introduce our semantic definitions of UML state machines in this subsection. Firstly some auxiliary functions(from Definition 7 to Definition 11) are introduced. Then we proceed to introduce the formalization procedure of our approach. For those ambiguity or inconsistent descriptions, we explicitly cite the original description in OMG UML 2.4.1 Super Structure Specification [6, Chapter 15] in our discussion for better clarity.

concatenation $\frown$: $N \times N \to N$ is used to concatenate two composite names to form a new composite name. For example, $S_0 \frown R_0 = S_0.R_0$

length $N \to \mathbb{N}$ (the set of Nature Number) will return the length of a composite name in terms of the total number of dots it contains. Let n be the total number of dots in a composite name s, then $length(s) = n + 1$.

prefix $\preceq$: $N \times N$ is a partial relation between two composite names. $\preceq \triangleq \{(s', s) | s', s \in N \wedge \exists\, s_0 \in N_{atom} : s' \frown s_0 = s \vee (s' \frown s_0, s) \in \preceq\}$. Note that in our definition above, the prefix of a string includes itself. We define the set of proper prefix of a string $\prec$: $N \times N$ as the set $\preceq \setminus \{(s, s)\}$.

first $N \to N_{atom}$ is an operation which returns the first sub-component of a given component name. $first(s) \triangleq \{s_0 | s \in N, \exists\, s_0 \in N_{atom} \wedge s' \in N : s_0 \frown s' = s\}$.

last $N \to N_{atom}$ is an operation which returns the last sub-component of a given component name. $last(s) \triangleq \{s_0 | s \in N, \exists\, s_0 \in N_{atom} \wedge s' \in N : s' \frown s_0 = s\}$.

construct type	name	syntax representation
state	$S1$	$((S1, \mathit{isComposite}, R1, \epsilon, \mathit{En\_1}, \mathit{Ex\_1}, \mathit{Do\_1}, \epsilon, \epsilon)$
	$S2$	$(S2, \mathit{isComposite}, R2, \epsilon, \epsilon, \epsilon, \mathit{Do\_2}, \epsilon, \epsilon)$
	$S3$	$(S3, \mathit{isSimple}, \epsilon, \{e4\}, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon)$
	$S4$	$(S4, \mathit{isSimple}, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon)$
final state	$f1$	$(f1, S_f)$
region	$R0$	$(R0, \{S1, S2\}, \phi)$
	$R1$	$(R1, \{S3\}, \{\mathit{init}1\})$
	$R2$	$(R2, \{S4, f1\}, \{\mathit{init}2\})$
pseudostate	$\mathit{init}1$	$(\mathit{init}1, I)$
	$\mathit{init}2$	$(\mathit{init}2, I)$
transition	$t1$	$(t1, \mathit{init}1, S3, \epsilon, \epsilon, a1, R1, (\epsilon, \phi), (\epsilon, \phi))$
	$t2$	$(t2, S3, S4, \{e2\}, g2, a2, R0, (\epsilon, \phi), (\epsilon, \phi))$
	$t3$	$(t3, S3, S2, \{e3\}, \epsilon, a3, R0, (\epsilon, \phi), (\epsilon, \phi))$
	$t4$	$(t4, S4, f1, \{e4\}, \epsilon, a4, R2, (\epsilon, \phi), (\epsilon, \phi))$
	$t5$	$(t5, \mathit{init}2, S4, \epsilon, \epsilon, a5, R2, (\epsilon, \phi), (\epsilon, \phi))$

Table 4: Syntax representation of state machine in Figure 7

component $N \times \mathbb{N} \rightarrow \mathbb{P}N_{atom}$ returns the ith sub-component specified by the index(i) of a given dot string. $\mathit{component}(s, i) \triangleq \{s_0 | s_0 \in N_{atom} \wedge \exists\, s', s" \in N : \mathit{length}(s') = i - 1 \wedge s' \frown s_0 \frown s" = s$.

Definition 7 (Direct container $\Uparrow$) $(S \cup R \cup S_f \cup PS) \rightarrow \mathbb{P}(S \backslash S_{simp} \cup R)$ *returns the direct container of a given vertex.* $\Uparrow(s) \triangleq \{s' | s' \in S \backslash S_{simp} \cup R : s \in s'.\mathit{subvertex}\}$

The direct container of a vertex or region is its direct ancestor, i.e. the state or region which has it as a subvertex. For example in Figure 7, the direct container of state S3 is $\Uparrow(S3) = \{R0.S1.R1\}$.

In UML 2.4.1 superstructure specification [6] for behavior state machine, the outermost construct should be a state machine. The only difference between a state machine and a composite state is that, a state machine does not have entry behavior, exit behavior, do activities and deferred event associated with it. Thus we can safely regard the outermost constructs as a composite state. If the outermost composite state has only one region[9], as is shown in Figure 7, we can even leave out the composite state and just utilize its directly containing region as the outermost construct. So in our formalization, we use R0 to represent the outermost construct of a state machine by default.

Definition 8 (Container $\uparrow$) $(S \cup R \cup S_f \cup PS) \rightarrow \mathbb{P}(S \backslash S_{simp} \cup R)$ *will return the set of all the containers of a given vertex.* $\uparrow(s) \triangleq \{\Uparrow(s)\} \cup \uparrow(\Uparrow(s))$.

[9]Actually, we can always add a wrapper region for a given state

The container operation returns all the regions and states which directly or indirectly contains the given vertex. For example, in Figure 7, container of state S3 is $\uparrow(S3) = \{R0.S1.R1, R0.S1, R0\}$.

Definition 9 (Containing $\downarrow$) *$(S \backslash S_{simp} \cup R) \to (S \cup R \cup S_f \cup PS)$ maps a state or a region to the set of vertices/regions contained in it. Formally: $\downarrow(s) \triangleq \{s.subvertex\} \cup \downarrow(s.subvertex)$.*

In contrast to container, the containing operation operates in the other direction, i.e., inwards from the given vertex (not including itself). For simple state, final state and pseudostates, their containing state set is empty. In the example of Figure 7, the containing vertex of composite state S2 is $\downarrow(S2) = \{R0.S2.R2.S4, R0.S2.R2.init2, R0.S2.R2.f1\}$.

Definition 10 (DirectContaining $\Downarrow$) *$(S \backslash S_{simp} \cup R) \to \mathbb{P}(S \cup R \cup S_f \cup PS)$ returns the direct containing substate of a given vertex. $\Downarrow(s) \triangleq \{s.subvertex\}$.*

For example, the direct containing vertex of composite state S2 is $\Downarrow(S2) = \{R0.S0.R1.S2.R1\}$. For simple state, final state and pseudostates, they do not have direct containing state, we use $\varnothing$ to represent the notation of on direct containing state.

Definition 11 (Restriction $\upharpoonleft$) *$(S \cup R \cup S_f \cup PS) \times (S \backslash S_{simp} \cup R) \to \mathbb{P}(S \backslash S_{simp} \cup region)$ selects a subset of containers of a given vertex according to the given restriction bound (scope). Formally: $\upharpoonleft(s, \iota) \triangleq \{s' | \forall (s, s') \in \uparrow : (s', \iota) \in \uparrow\}$.*

Restriction, as indicated by the shape of the operator, will return the containers of a given vertex restricted by a bound/scope. For example in Figure 7, the restriction of composite state S4 by S02is $\upharpoonleft(S4, S2) = \{R0.S2.R2\}$. The purpose to introduce this operator is to calculate the set of states exited and entered as a result of firing some transitions. The restriction bound/scope is the scope ι of the transition in this case.

Definition 12 (Configuration) *The set of all configurations of the current state machine is defined as: $\mathcal{K} \triangleq \{\widehat{S} \subset S \cup S_f | \forall s \in \widehat{S}, \uparrow(s) \subset \widehat{S} \wedge \forall r \in R : \Uparrow(r) \in \widehat{S} \Rightarrow | \Downarrow(r) \in \widehat{S} | \leqslant 1\}$.*

The first part of the definition constrains that for any state in the configuration, its container must also be in the configuration. This is required by the hierarchical state structure and the behavior semantics of the UML behavior state machine. The second part of the definition indicates two points: Firstly, if an orthogonal state with more than one region is in the configuration, then all its directly contained regions should also be activated. Secondly, within each orthogonal region which is activated currently, there is exactly one of its directly contained state in the configuration. It exactly captures the two invariant requests about a configuration as specified by

> "If a composite state is active and not orthogonal, at most one of its substates is active. If the composite state is active and orthogonal, all of its regions are active, with at most one substate in each region."
>
> [6, Chapter 15.3.11, Semantics, Composite State, p.564]

In the following, we will use $\mathcal{K}_c : \mathcal{K}$[10] and $\mathcal{K}_x : \mathcal{K}$ to represent the current configuration and the next configuration which can be reached from the current configuration given a triggering event.

Definition 13 (Pseudo Configuration) *A Pseudo Configuration represents the set of vertices the state machine may temporally resides inbetween a run-to-completion step, i.e., it is the set of vertices the state machine is in after a transition is executed. Formally,*
$\mathcal{PSK} \triangleq \{\widehat{S} \subset S \cup S_f \cup PS \backslash I \,|\, \forall\, s \in \widehat{S}, \uparrow(s) \in \widehat{S} \wedge \forall\, r \in R : \Uparrow(r) \in \widehat{S} \Rightarrow |\Downarrow(r) \in \widehat{S}\,| \leqslant 1\}$.
The definition of a pseudo configuration is no different from a configuration except that pseudostates are allowed to present. Configuration is a special case of pseudo configuration, i.e., $\mathcal{K} \subset \mathcal{PSK}$. *For example, in Figure 5, if the current pseudoconfiguration is* $\{S0, S1, C1, S6\}$ [11] *and join transition* $t6$ *is fired, the next pseudoconfiguration will be* $\{S0, j1, S6\}$.

Definition 14 (SeqTrans) $STr \in N$ *is a set of sequence transitions which is defined as follows:*

1. $\forall\, t \in Tr, t \in STr$
2. $\forall\, t_i, t_j \in STr \wedge last(t_i).\widehat{T} \subset first(t_j).\widehat{S}, t_i \frown t_j \in STr$
3. $\forall\, t_i, t_j \in STr \wedge IsJoin(last(t_i)) \wedge IsJoin(first(t_j)) \wedge last(t_i).\widehat{T} = \{first(t_j).\widehat{J}.jn\} \wedge last(t_i).\widehat{J}.jn \in first(t_j).\widehat{S}, t_i \frown t_j \in STr$

In our semantics definitions, we consider the name of each transition as the unique identifier for a transition. So when we write $t \in Tr$, we are referring to $t.n$. (We choose

[10] We may use some defined concepts as types in this paper. Actually any type is just a set of elements. So any set we have defined can be used as a type. We use $\mathcal{K}$ as a type and the comma : can be interpreted as "of type".

[11] *Since each vertex and region in this example has a unique name, we just use their name instead of the hierarchical dotted string here.*

to include only transition names in a sequence transition to avoid unnecessary costs.) For example, in Figure 5, transition t_6 and t_{11} form a sequence transition by the junction pseudostate j_1, then $t_6 \frown t_{11}$ is denoted as $t_6.t_{11}$.

Transition is a special case of sequence transitions which does not have pseudostates as connectors.

If a sequence transition's target state is a subset of another transition's source states, then they can be connected to form a new sequence transition.

The last condition is specially for connecting two or more join transitions in a row. For example in Figure 5, the two join transitions $t24$ and $t28$ are connected consecutively. The transition $t24$ is public to both, i.e., part of compound transition connected by join vertex $jn1$ as well as part of transition connected by join vertex $jn2$. Thus the specified condition 2 is not suited here. In condition 3, we require that the second join vertex must be the target vertex of the first join transition $last(t_i).\widehat{T} = \{first(t_j).\widehat{J}.jn\}$ and the first join vertex must be in the set of source vertex of the second join transition $last(t_i).\widehat{J}.jn \in first(t_j).\widehat{S}$. Fork transitions do not have such a concern since it is required that a fork transition must target states (as opposed to pseudostate) in orthogonal regions.

Compound transitions, connected by join and fork vertices contain multiple transitions. Some of them may require simultaneous invocations for orthogonal regions, Therefore, they need to be executed together. In our syntax notation, i.e. in Definition 5, we had explicitly represented the compound transition connected by one join or fork vertex. In Figure 5, $t24.t28$ (which represents$(((t23 \mid\mid t22)t24) \mid\mid t25)t28)$ is a sequence transition. In Figure 5, transition sequence $t1, t4.t5, t4.t5.t7.t8.t9$ are also elements of STr.

Definition 15 (evaluate) *evaluate* $: S \cup S_f \cup PS \backslash I \times Trigger \rightarrow \mathbb{P}STr$ *will return the set of sequence transitions of which all the guards of all its component transitions are evaluated to true under the current Active vertex and the dispatched event matches its triggering event. Formally,* $evaluate(S, e) \triangleq \{st \in STr \mid first(st).\widehat{S} \in S \wedge last(st).\widehat{T} \in (S \cup S_f \cup T \cup C) \wedge \nexists st' \in evaluate(S, e) : st' \prec st \wedge (\forall\, i \in [1, length(st)] : enable(component(st, i), e)))\}$.

$$enable(t, e) \triangleq \begin{cases} T, & t.\varphi = T \wedge (t.\epsilon = e \vee t.\epsilon = \phi) \\ F, & otherwise \end{cases}$$

We consider a set of enabled sequence transitions. Such a sequence of transitions may emanate from a set of arbitrary vertex but must ends in either a set of states, or choice or terminate pseudostates. We also explicitly deal with history vertex. Our consideration here is as follows

- We need to temporarily pause at the choice pseudostate because we need to make a dynamic decision as to which outgoing transition to follow.
- For the terminate state, since it means the termination of the whole state machine, there is no need to proceed any more.
- Pause at choice and terminate pseudostates will not affect the evaluation of enabled transitions. For choice pseudostate, there must be a default transition emanating from it to guarantee the well-formedness in case guards of all the other transitions are evaluated to false.

  > "If more than one of the guards evaluates to true, an arbitrary one is selected. If none of the guards evaluates to true, then the model is considered ill-formed. (To avoid this, it is recommended to define one outgoing transition with the predefined "else" guard for every choice vertex.)"
  >
  > Chapter 15.3.8, Semantics, choice, p551

  For terminate pseudostate, no further operation is needed.
- For fork (resp. join) transitions, since transitions entering a join (resp. outgoing from a fork) vertex must not have triggers and guards [6, Chapter 15.3.8, Semantics, p.551], we can treat a join (resp. fork) transition the same way as a normal transition and considers only the transition emanating from a join (resp. entering a fork) vertex.

Note that, after each evaluation is called, we need to add the result sequence transition to the current Firable transition set.

Definition 16 (Enabled Transitions) *$\mathcal{K} \times Trigger \to \mathbb{P}STr$ is a function which maps the current configuration and triggering event to a set of enabled sequence transitions which emanates from the states in the current configuration and are activated by the triggering event. Formally, $EnTrans\ (\mathcal{K}_c, e) \triangleq evaluate(\mathcal{K}_c, e)$.*

Enabled transitions are those sequence transitions evaluated to be enabled in current configuration.

Definition 17 (Leave) *$Tr \times \mathcal{PSK} \to \mathbb{P}(S \cup PS)$ maps a transition to the set of states and regions it leaves in the current configuration on firing. Formally:*

$$Leave(t, \mathcal{PSK}_c) \triangleq \begin{cases} \phi, & t.\iota \in t.\widehat{S} \\ \bigcup_{t' \in t.\widehat{J}.SAP} Leave(t', \mathcal{PSK}_c) \bigcup Leave(main(t), \mathcal{PSK}_c), & IsJoin(t) \\ \bigcup_{t' \in t.\widehat{F}.TAP} Leave(t', \mathcal{PSK}_c) \bigcup Leave(main(t), \mathcal{PSK}_c), & IsFork(t) \\ \downarrow(t.\iota) \cap \mathcal{PSK}_c & otherwise \end{cases}$$

The operator Leave is introduced mainly for deciding conflict transitions and calculate next configuration.

- If a transition is a internal transition, as indicated by $t.\iota \in t.\widehat{S}$ which means the source and target state are the same, then the set of states it left on firing is empty.
- If a transition is a join (fork) transition, we need to count not only the main transition, but also those transitions ending in the join (emanating from fork) vertex.
- For other transitions, the states left on firing are all the containers and containing states of the source state which are in the current pseudo configuration $\mathcal{PSK}_c$. In other words, all the subvertices of $t.\iota$ (the scope of transition t) which are in the current pseudo configuration will be exited. This is captured by $\downarrow(t.\iota) \cap \mathcal{PSK}_c$.

For example in Figure 8, the set of states and regions it left on firing transition $t6$ is shown in table 5.

Fired transition	$t6$
$\mathcal{PSK}_c$	$S0, S2, S3, S5, S7$
states left	$S0, S2, S3, S5, S7$

Table 5: States left on firing a transition

Definition 18 (Conflict) *$Tr \times Tr \times \mathcal{K} \to \mathbb{B}$ is a function which decides whether two transitions t, t' conflict with each other. Whether two sequence transitions are in conflict with each other is decided by their first transitions.*

$$Conflict(t, t', \mathcal{K}_c) \triangleq \begin{cases} T, & ((Leave(t, \mathcal{K}_c) = \phi \vee Leave(t', \mathcal{K}_c) = \phi) \wedge t.\widehat{S} \bigcap t'.\widehat{S} \neq \phi) \\ & \vee\ Leave(t, \mathcal{K}_c) \cap Leave(t', \mathcal{K}_c) \neq \phi \\ F, & otherwise \end{cases}$$

"Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty."

[6, Chapter 15.3.12, Semantics, Conflicting transitions, p.575]

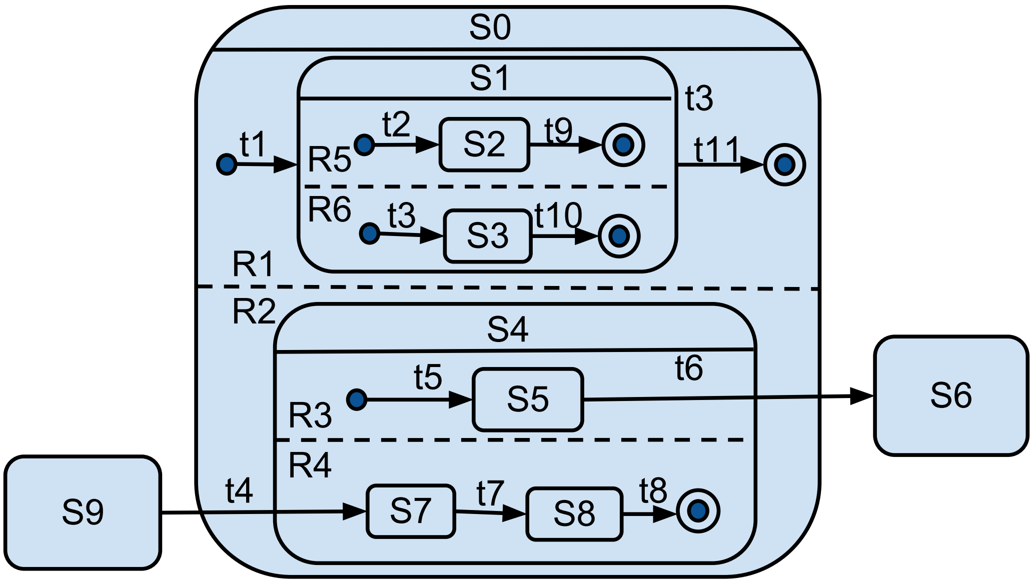


Figure 8: Transitions Enter and exit orthogonal composite states

This is captured by $\mathit{Leave}(t, \mathcal{K}_c) \cap \mathit{Leave}(t', \mathcal{K}_c) \neq \phi$.

"An internal transition in a state conflicts only with transitions that cause an exit from that state."

[6, Chapter 15.3.12, Semantics, Conflicting transitions, p.575]

This is captured by $(\mathit{Leave}(t, \mathcal{K}_c) = \phi \vee \mathit{Leave}(t', \mathcal{K}_c) = \phi) \wedge t.\widehat{S} \bigcap t'.\widehat{S} \neq \phi$.

We use configuration instead of pseudo configuration because that we only need to solve a conflict between transitions at the beginning of a run to completion step. In the middle of a run to completion step, if we encounter a choice vertex, we may need to do the evaluation again, but there is no need to solve conflict. Since it has been specified that if more than one guards of transitions emanating from the choice vertex are evaluated to true, an arbitrary one is selected to be executed.

"If more than one of the guards evaluates to true, an arbitrary one is selected"

[6, Chapter 15.3.8, Semantics, p.551]

Definition 19 (Priority) *$Tr \times Tr$ is a partial relation between two transitions. A pair of transitions $(t, t') \in$ Priority means that transition t and t' are conflicting with each other and transition t has higher priority over transition t'. Formally,* **Priority:**$\triangleq$ $\{(t, t') | \mathit{conflict}(t, t') \wedge \exists\, s \in t.\widehat{S} : \forall\, s' \in t'.\widehat{S}, \mathit{distance}(s, \mathit{LCA}(s, s') > \mathit{distance}(s', \mathit{LCA}(s, s')\}$.

"The priority of a transition is defined based on its source state"

[6, Chapter 15.3.12, Semantics, Firing Priorities, p.576]

. Here we consider two kinds of situations in solving conflict transitions, differentiated by whether an orthogonal composite state is involved. The first situation is that neither of the two conflict transitions is a part of a compound transition connected by join pseudostate, i.e., emanating from different regions of an orthogonal composite state. This scenario is specified as follows:

> "By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing state."
>
> [6, Chapter 15.3.12, Semantics, Firing Priorities, p.576]

In the rest of the paper, we refer this to the "hierarchical priority principle". For example, in Figure 9[12], transitions $t1$ and $t3$ are conflicting and $t1$ has priority over $t3$ in this case. As for $t1$ and $t9$, the priority is unsolved since they emanate from the same state.

The second situation is that at least one of the two conflict transitions is a part of a compound transition connected by join pseudostate. This situation is described as follows:

> "The priority of joined transitions is based on the priority of the transition with he most transitively nested source state"
>
> [6, Chapter 15.3.12, Semantics, Firing Priorities, p.576]

We refer this as " join priority principle" in the rest of the paper. The "most transitively nested source state" is ambiguous since the baseline is not clear. But the intuition to us is that the baseline should be the Least Common Ancestor of the involved states. However, another description about the suggested transition algorithm provided by [6, p.576] seems to provide the contradict idea:

> "States in the active state configuration are traversed starting with the innermost nested simple states and working outwards, etc. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels, this is resolved by terminating the search in each orthogonal state once a transition inside any one of its components is fired."
>
> [6, Chapter 15.3.12, Semantics, Transition selection algorithm, p.576]

This has been discussed in [37], where they did not get a conclusion about the ambiguous situation, but provide formalization for both situations. In our work, we follow the hierarchical priority principle (not the transition selection algorithm) and

[12]In this example, we remove all the triggering events, guards and effects for simplicity. We assume that guards of all transitions are satisfied and all the pair of transitions discussed here are triggered by the same event.

choose the Least Common Ancestor(LCA) as the baseline. Priority is given to the transition with the most transitively nested source states, i.e., the source state has the largest distance with the Least Common Ancestor (LCA) of the source states of all the conflicting transitions. The reasons for this choice is as follows:

- Since we need to compare states from orthogonal regions, which are not in the same branch of state hierarchy [13], we need a uniformed reference which is "fair" and convincing to both side. Referring to the innermost simple state(leaves in the tree) in either side is not fair nor convincing to the other side since they are not in the same branch of state hierarchy and hence not comparable. Further, in some cases, the priority is not solvable if we follow the algorithm suggested by [6, Chapter 15.3.12, Semantics, Transition selection algorithm, p.576]. For example in Figure 9, consider join transition $t7$ and $t8$, the priority is not solved since $S5$ and $S7$ are both the innermost simple states in each region.

- Since this is a conflict in the UML specification [6, Chapter 15.3.12, Semantics p.576] itself, we should follow the most reasonable one. We consider the priority principle as the first hand reference while the suggested transition selection algorithm as the second hand reference since it is based on the priority principle and should have supported it. Thus we choose to follow the priority principle in our work.

In Figure 9, Transition $t3$ and $t4$ are conflicting. Since $t4$ is part of a join transition, the priority is thus decided by $t3, t4$, and $t5$(since $t5$ is also part of the join transition). The source states of the three transitions are $S3, S3$ and $S7$ respectively, the LCA of them are $S0$. $S7$ has the largest distance [14] with $S0$. So the join transition has higher priority over $t3$, thus $t4$ has higher priority over $t3$. Apply the same rule, we can know that the join transition $t7$ [15] has higher priority over the join transition $t8$.

The other possibilities are considered undefined and we suggest to avoid those situations in the modeling.

Definition 20 (deferral Conflict) *deferral conflict is the conflict between a defferred event of a state and a transition consuming the event. We solve such a conflict and*

[13]Remind that all the vertex and regions in a UML state machine form a tree structure, the root of which is the outermost region. Each orthogonal composite state acts as a branching point and each region of it is the subroot of a subtree. Thus states in different subtrees are not comparable from bottom up(use leaves as baseline), but comparable from top down(use root as base line).

[14]We choose to leave out the region when considering the distance between states, thus the distance between $S7$ and $S0$ is 3 while the distance between $S3$ and $S0$ is 1.

[15]We use the single transition emanating from the join transition to represent the join transition. So $t7$ here represent $(t4, t5)t7$ and $t8$ represents $(t2, t6)t8$

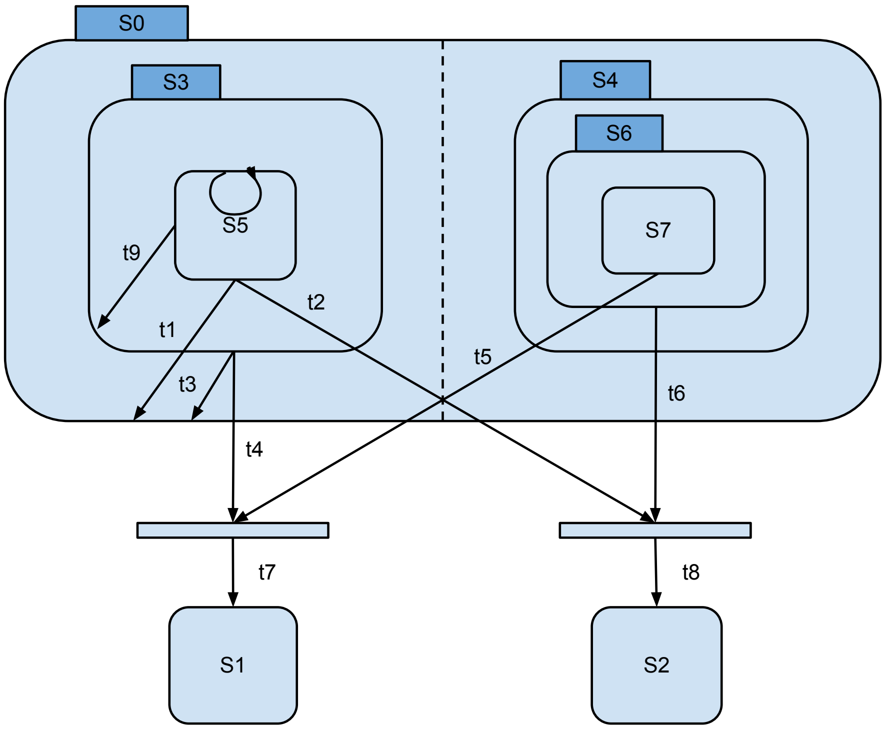


Figure 9: Conflicting transitions illustration

returns true if the event is defferred by the state and false if the event is consumed by a transition.

$$defConflict(t, \mathcal{K}_c, e) \triangleq \begin{cases} T, & t \in EnTrans(\mathcal{K}_c, e) \wedge \exists\, s, s' \in \mathcal{K}_c : \\ & (e \in s.deffer \wedge s' \in t.\widehat{S} \wedge s' \in \uparrow(s)) \\ F, & otherwise \end{cases}$$

In our definition, we try to solve the conflict following the specifications of [**?**], which is described as follows:

> “In case of a composite orthogonal state, substates of orthogonal regions may also introduce deferral conflicts. The conflict resolution follows the triggering priorities, where nested sates override enclosing states. In case of a conflict between states in different orthogonal regions, a consumer state overrides a deferring state”
>
> [6, Chaperter 15.3.11, Semantics, Deferred events]

So in our definition of deferral conflict, we consider two situations.

- If the confliction does not involve orthogonal composite state, which means that the involved states must be in the same branch of state hierarchy, then we give higher priority to substates. In our definition, $\exists\, s \in \mathcal{K}_c : e \in s.\mathit{deffer}$ capture the condition that the event e is in the defferred event set of a state in the current configuration. $\exists\, s' \in \mathcal{K}_c : \exists\, t \in \mathit{EnabledTrans}(\mathcal{K}_c, e) \wedge s' \in \mathit{first}(t.\widehat{S})$ captures the condition that event e also triggers some transitions whose source states are in the current configuration. Thus a confliction arises. $!\mathit{IsOrthog}(\mathit{LCA}(s, s')) \wedge s \in \uparrow(s')$ captures the condition that no orthogonal composite states are involved in the confliction and the state with event e in its defferred event set is the substate of the state which has transition with trigger e emanating from it.

- If the confliction involves orthogonal composite state, we do not compare the state hierarchy as has been done in solving conflicts between transitions in Definition 19, but directly give higher priority to transitions which consumes the current event.

In the case which join transitions are involved, since orthogonal composite states must be involved in this situation, the second situation should be applied. For example in Figure 10, the defferred event in state $S6$ can be conflict with transition $t8$, since $S6$ is a substate of $S5$, so the event deferral has higher priority. The defferred event in state $S6$ also conflict with transition $t2$, in this case, they resides in different orthogonal regions, i.e. composite state $S1$ is involved, so the priority if assigned to transition $t2$. The same condition applies to the confliction of defferred event in state $S6$ with transition $t7$, in this case, priority is also assigned to transition $t7$.

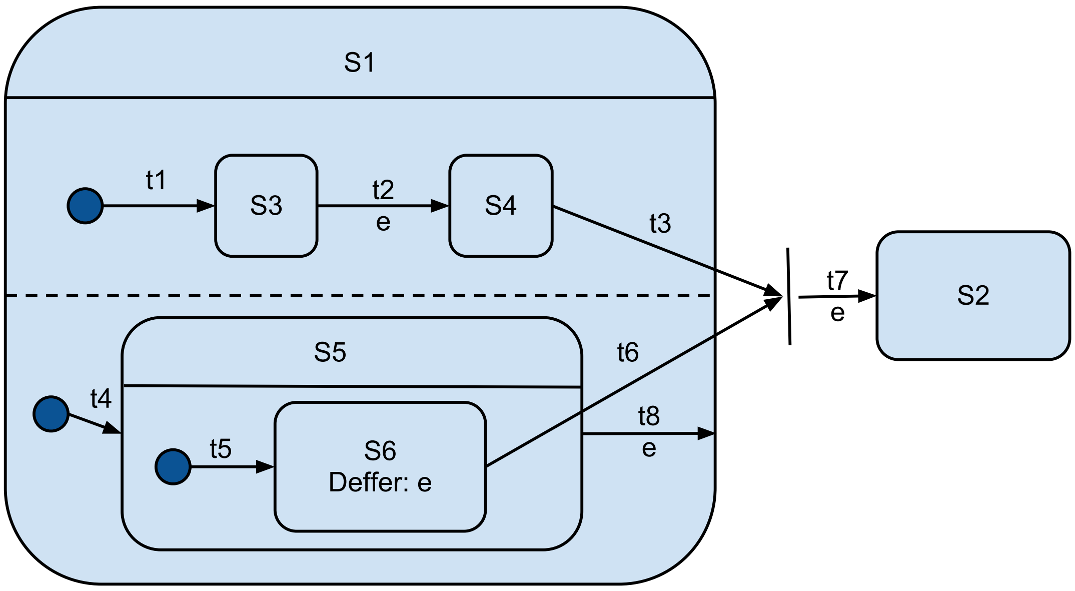


Figure 10: deferral Conflict illustration

Definition 21 (Firable Transitions) *$\mathcal{K} \times Trigger \rightarrow STr$ is the set of enabled transitions of which conflicts are solved by priority rules. Formally, we define Firable Transitions* $\textbf{FirTrans}(\mathcal{K}_c, e) \triangleq \{st| \exists\, st \in EnTrans(\mathcal{K}_c, e) \wedge !deferralConflict(first(st), \mathcal{K}_c, e) \wedge \nexists st' \in FirTrans(\mathcal{K}_c, e) : conflict(first(st), first(st')) \wedge \nexists st" \in EnTrans(\mathcal{K}_c, e) : conflict(first(st), first(st")) \wedge priority(first(st"), first(st))\}$

The purpose of the function Firable Transition is to select the largest non-conflicting subset from Enabled transitions such that transitions in the selected subset are non-conflicting and have higher priorities over the conflicting ones in the left part. The first step of deciding firable transitions is to check the deferral conflict. $!deferralConflict(\mathcal{K}_c, e)$ means there is no such confliction or transition is assigned higher priority, which is the basic condition before we proceed to check conflicts between enabled transitions.

Definition 22 (ActiveSubState) *$\mathbb{P}(S \cup S_f \cup R \cup PS \backslash I) \times Trigger \rightarrow \mathbb{P}S$ will return all the transitively/indirectly activated states due to the given set of active vertices. Formally, $ActiveSubState(S, e) \triangleq \bigcup_{s \in S} Active(s, e)$, where*

$$Active(s, e) \triangleq \begin{cases} (s.s = \phi \vee s.s \in S_f)?defhistory(s) : s.s, & s \in H \bigcup H^* \\ ActiveSubState(first(evaluate(\{s\}, e)).\widehat{T}, e), & s \in C \\ \bigcup_{s' \in \Downarrow s} inital(s') & s \in S \backslash S_{simp} \\ s, & s \in S_{simp} \cup S_f \\ initial(s), & s \in R \\ stop, & s \in T \\ \phi, & otherwise \end{cases}$$

This operation is introduced to deal with the varieties raised by history, choice, initial and terminate pseudostates as well as final states. The detailed explanations about each item in the definition are as follows:

- If we have reached a history vertex, that means we need to consider two possibilities, i.e., default history state and the recorded last active state. The operator " c ? a: b " is inspired by the C language which means that if condition c holds then return a, else return b. In the Enabled Transition step (Definition 16), we just guarantee that the sequence transition whose last subtransition emanates from a history vertex is enabled. This is the point where we should decide which states should we enter. $defhistory(s) \triangleq \{s'|s \in H \bigcup H^* \wedge \exists\, t \in Tr : t.\widehat{S} = \{s\} \wedge s' \in t.\widehat{T}\}$ will return the default history states pointed to by the default history transition emanating from history pseudostate s.

- If we have reached a choice vertex, we will first evaluate which sequence transition is enabled in the current environment and return the active substate of the target state of the first transition of the enabled sequence transition. The set evaluate is defined in Definition 15.

- If we have reached a composite state, we need to active its default substate indicated by the initial vertex. $\mathit{initial} : R \to S \bigcup S_f$ will map a region to its default substate indicated by its containing initial vertex. $\mathit{initial}(r) \triangleq \{s | \exists\, s' \in I : s' \in \Downarrow(r) \wedge \exists\, t \in \mathit{Tr} : t.\widehat{S} = \{s'\} \wedge t.\widehat{T} = \{s\}\} \cup \mathit{initial}(\Downarrow(s))$.

- If we have reached a final state, we need to wait for all the orthogonal regions of its container composite state have reached their final state and active its container composite state. Note that we cannot recursively active a given vertex in this situation. The semantic meaning of reaching a final state is to exit the container composite state, in a well-formed state machine, there should be a transition emanating from the composite state. If we recursively active the composite state, the result will be enter the substates indicated by the initial state of each region in the composite state, which is incorrect semantics. We use $\mathit{Infinal} : S \backslash S_{simp} \to \mathit{Bool}$ to wait for all the orthogonal regions of a composite state has reached their final state. Since we did not consider completion event currently, we just coarsely describe this concept.

- If we have reached a simple state, just active it directly.

- If we are in a region, active its default substate indicated by initial state.

- If we have reached a terminate state, then the state machine is terminated by definition. We use *stop* to represent this.

- If we have reached other vertex except for those discussed above, no active substate exists since those vertex do not have substates. We include these pseudo vertex just to provide a unified view of the semantic definition.

Definition 23 (Enter) *$\mathit{Tr} \times \mathit{Trigger} \to \mathbb{P}(S \cup R \cup S_f \cup PS)$ maps a transition to the set of vertices and regions it enters on firing. Formally,*

$$\mathit{Enter}(t, e) \triangleq \begin{cases} \bigcup_{(t',\_,\_,\_) \in t.\widehat{F}.TAP} \mathit{Enter}(t', e) \bigcup \mathit{Enter}(\mathit{main}(t), e) & \mathit{Isfork}(t) \\ \bigcup_{(t',\_,\_,\_) \in t.\widehat{J}.SAP} \mathit{Enter}(t', e) \bigcup \mathit{Enter}(\mathit{main}(t), e) & \mathit{Isjoin}(t) \\ \{s' | s' \in \upharpoonleft(s, t.\iota), s \in t.\widehat{T}\} \bigcup \mathit{ActiveSubState}(t.\widehat{T} \bigcup R, e) \bigcup t.\widehat{T}, & \mathit{otherwise} \end{cases}$$

where $R = \{r | s \in t.\widehat{T} \wedge \forall\, s_0 \subset \upharpoonleft(s, t.\iota) \bigcup t.\widehat{T} \wedge \mathit{IsOrthogonal}(s_0), r \in \Downarrow(s_0)\} \backslash (\bigcup_{s \in t.\widehat{T}} \uparrow(s))$.

In the rest definitions, we use underline _ to represent the element we do not care. Entering a state means entering all the containers of the state (up to the scope of the transition) and all the substates of the state. For example, in Figure 5, on firing the fork transition $t13$, the set of states/regions entered would be $\{R0.S0.R2.S10, R0.S0.R2.S10.R1, R0.S0.R2.S10.R2, R0.S0.R2.S10.R1.S12, R0.S0.R2.S10.R1.S12.R1, R0.S0.R2.S10.R2.S8, R0.S0.R2.S10.R1.S12.R1.S7\}$.

One situation we need to pay extra attention is entering an orthogonal composite state. We need to make all the default states in each orthogonal region of containers of the target state active. For example in Figure 8, firing of transition $t4$ will cause all the containers of $S7$ to be entered, since container $S0$ and $S4$ are orthogonal composite states, we also need to enter region $R1$ and $R3$, which means their default states ($S1$ and $S5$ respectively) are entered. This will be applied recursively and finally, the set of active states will be $\{S7, S4, S5, S1, S2, S3, S0\}$. We formalize this by using an auxiliary function called *ActiveSubState*, which will active the containing vertex of a given state/region. We capture the activation of orthogonal composite state by using the set $R = \{r | s \in t.\widehat{T} \wedge \forall\, s_0 \subset \upharpoonleft(s, t.scope) \bigcup t.\widehat{T} \wedge \mathit{IsOrthogonal}(s_0), r \in \Downarrow(s_0)\} \backslash (\bigcup_{s \in t.\widehat{T}} \uparrow(s))$, which contains all the orthogonal regions of a composite state, which is container of the target state except for the regions in the set of containers of the target state.

Definition 24 (Next Pseudo Configuration) *$Tr \times \mathcal{PSK} \times Trigger \to \mathcal{PSK}$ computes the next pseudo configuration which will be active after executing the current transition. Formally, $\mathit{NextConfig}(t, \mathcal{K}_c, e) \triangleq \mathcal{PSK}_c \backslash \mathit{Leave}(t, \mathcal{PSK}_c) \bigcup \mathit{Enter}(t, e)$.*

The definition of next pseudo configuration is straightforward. Starting from the current pseudo configuration, remove all the states which are exited on firing transition t and add all the states which are entered on firing transition t. But note that the order does matter in this definition. We must eliminate the set of states exited from the current pseudo configuration first, before adding the set of states entered. This is to prevent missing states when a transition first exit then re-enter a state. For example in Figure 5, transition $t3$ exits $S4, S11, S1$ and re-enter them. If we add the entered states first before removing exited states, the result will be missing these three states [16].

[16] The root reason is that, there is no duplicate elements in a set.

In order to represent the non-deterministic effect sequences, such as those effects on transitions ending in a join pseudostate or emanating from a fork pseudostate, and the do activity, which is activated by entering its belonging state, with the entry behavior of its substates etc, we introduce a new operator $\|$(parallel). n effects $e_1, e_2, \cdots, e_n$ execute in parallel is represented by $\|_{i=1}^{n} e_i$, which indicates that all the n effects are executed non-deterministically, i.e., no orders among them are forced. Another meaning of the parallel operator is that the events are executed interleavingly, without interference with each other. The operator *sequence*, denoted by a semicolon(;) is used to represent the execution order of two (collections of) effects. For example, $e_1;\ e_2$ represents that effect e_2 should be executed only after effect e_1 has finished. In the following definitions, we use AL to represent an ordered sequence of effects $e \in B$.

Definition 25 (ExitBehavior) *$\mathcal{PSK} \times Tr \rightarrow AL$ will return the ordered exit behavior of states a given sequence transition leaves. Formally:*

$$ExitBehavior(\mathcal{PSK}_c, t) = Exit(\Downarrow(t.\iota) \cap \mathcal{PSK}_c, \mathcal{PSK}_c)$$

$$Exit(s, \mathcal{PSK}_c) \triangleq \begin{cases} \|_{s' \in \Downarrow(\Downarrow(s)) \cap \mathcal{PSK}_c} Exit(s', \mathcal{PSK}_c);\ exitb(s), & isOrthog(s) \\ Exit(\Downarrow(\Downarrow(s)) \cap \mathcal{PSK}_c, \mathcal{PSK}_c);\ exitb(s), & !isOrthog(s) \\ exitb(s), & isSimple(s) \end{cases}$$

$$exitb(s) = Stop(s.do);\ s.exit$$

An active state configuration can actually be represented as a tree structure, as has been described:

"Furthermore, since the state machine as a whole and some of the composite states in this hierarchy may be orthogonal(i.e., containing regions), the current active "state" is actually represented by a set of trees of states starting with the top-most states of the root regions down to the innermost active substate. We refer to such a state tree as a state configuration."

[6, Chapter15.3.11, Semantics, Composite state, p.564]

In our current definition, the root of the tree should be the direct containing substate of the scope of the fired transition since we just focus on a single transition. The exit behaviors should be collected from the innermost state, i.e. from leaves to the root in the tree hierarchy. But we still need to avoid to collect duplicated behaviors. Since transition(instead of compound transition) is the basic unit of collecting exit behaviors and the exit behavior of a compound transition is composed of the exit behavior of its

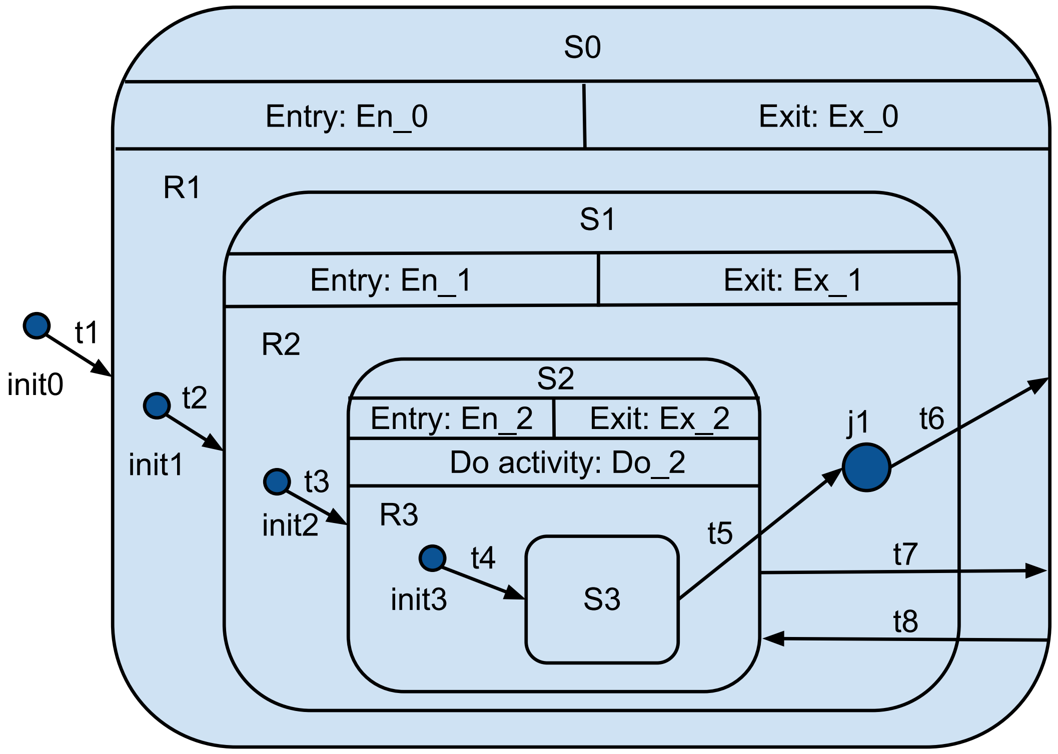


Figure 11: Collecting Exit Behavior

component transitions. We need to do the proper truncating so that no duplicate exit behavior is collected. For example, in Figure 11, in order to collect the exit behavior to be executed by firing compound transition $t5.t6$, we need to first collect the exit behavior caused by firing $t5$, then collect the exit behavior caused by firing $t6$. One point we need to pay extra attention is that on firing $t6$, states $S1$ is directly exited. But state $S2$ and $S3$ are also exited indirectly since they are substates of $S1$. Then we should avoid to collect duplicate exit behaviors since we had already collected the exit behavior of $S3$ and $S2$ when firing $t5$. This is guaranteed by $\mathcal{PSK}_c$. The $\mathcal{PSK}_c$ at the moment we fire transition $t6$ does not include states $S2$ and $S3$ since they had been exited on firing transition $t5$. So when we compute the set $\Downarrow(\Downarrow(s)) \cap \mathcal{PSK}_c$, there will be no duplicate states as has been exited by previous fired transitions. For a given pseudo configuration $\mathcal{PSK}_c$ and a fired transition t, The exit behavior is computed recursively from the root of the tree. Three conditions are distinguished here:

- if the current state is an orthogonal composite state, then all its direct containing substates which are in the current pseudoconfiguration are exited in parallel, followed by exiting the orthogonal composite state.

- if the current state is not orthogonal, then the direct containing substate of the current state is exited followed by exiting the current state.
- if the current state is a simple state, then its exit behavior is collected.

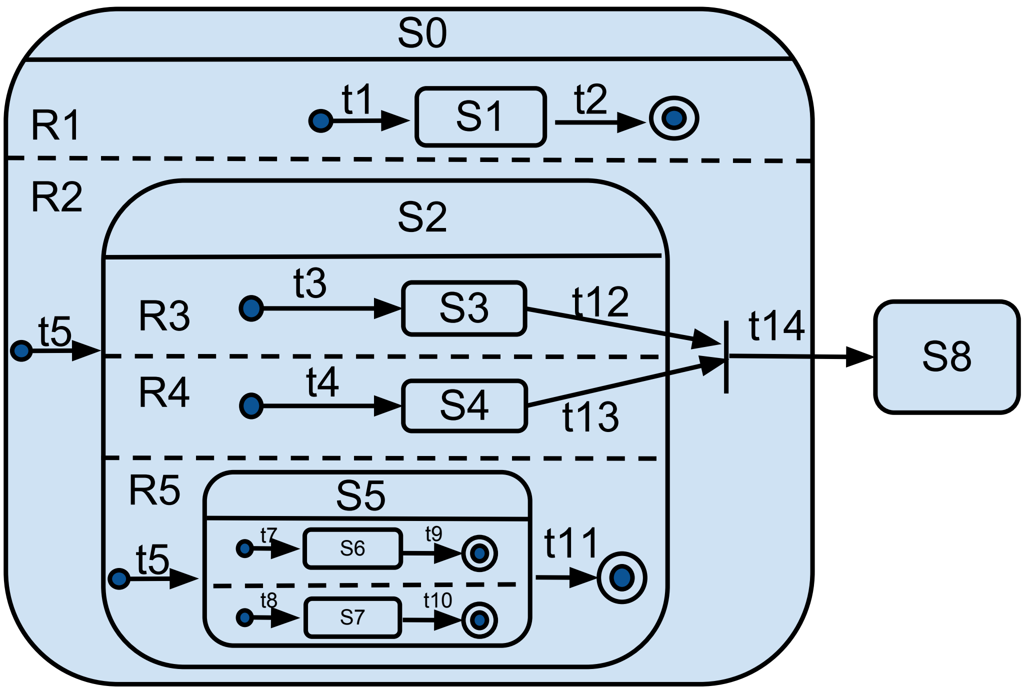


Figure 12: Exit Orthogonal Composite State on join transition

Definition 26 (EntryBehavior) *$Tr \times Trigger \to AL$ will return the ordered entry behavior of states a given transition enters. Formally:*

$$EntryBehavior(t, e) = Entry(\Downarrow(t.\iota) \cap \mathcal{PSK}_c, Enter(t, e))$$

$$Entry(s, S) \triangleq \begin{cases} s.entry;\ (\|_{s' \in \Downarrow(\Downarrow(s)) \cap S}\ Entry(s', S) \parallel s.do), & isOrthog(s) \\ s.entry;\ (Entry(\Downarrow(\Downarrow(s)) \cap S, S) \parallel s.do), & !isOrthog(s) \\ s.entry;\ s.do. & isSimple(s) \end{cases}$$

The entry behaviors should be collected from the outermost state to the innermost state. Similar to collecting exit behaviors, we operate from the root down to leaves.

- If a state is an orthogonal composite state, then its entry behavior should be invoked, all its direct containing substates which are present in the state tree should be activated in parallel following the activation of the orthogonal composite state.

- If a state is not orthogonal, then its entry behavior should be collected, followed by the entry behavior of its direct containing substate.
- If a state is a simple state, then its entry behavior and do activity are collected.

One thing we should pay additional attentions is that entry behavior and do behavior should both be executed on entering the state, and do behavior should be executed after the entry behavior has finished.

> "If defined, entry actions are always executed to completion prior to any internal behavior or transitions performed within the state"
>
> [6, Chapter 15.3.11, Associations, entry, p.561]

But there is no constraint on the terminating time of do behavior. Actually do activity of the current composite state will continue its execution until its belonging state is exited or it finishes execution.

> "The execution starts when this state is entered, and stops either by itself or when the state is exited whichever comes first"
>
> [6, Chapter 15.3.11, Associations, doActivity, p.561]

So it is possible that the do activity of the current composite state executes in parallel with the entry behavior/do activity of its substates. For example, in Figure 13, the do activity of composite state $S1$, i.e. Do_1 may be executed in parallel with $En\_2, En\_4, Do\_2, Do\_3, Do\_6$. There is no constraint specifies the execution order of a do activity of a composite state and the entry behavior of its direct containing substate, i.e. the execution order of Do_1 and En_2 in Figure 13 is not specified. But we know that Do_1 should follow En_1 and En_2 should also follow En_1. So we assume that En_2 and Do_1 can start execution at the same time. We use the parallel operator $\|$ to express the meaning that the two parts connected by the parallel operator should start execution at the same time and the activities in each part are executed interleavingly.

There is another issue about the parallel operator that we should notice: Actually we are using the concatenation operator ; to indicate the execution order between the do activity $\mathcal{D}of(s)$ and the rest of behavior $Entry(child(s))$. A tricky situation is that if a composite state do not have entry behavior, i.e. $\mathcal{E}nf(s) = \epsilon$, then we are unable to represent the invoking order of Do activities. For example, in Figure 13, we would like to express the meaning that Do_3 is invoked before Do_6, but we may mistakenly write $Do\_6 \parallel Do\_3$ in his case and the active order of Do_3 leading Do_6 is missing. So we chose to use ϵ to represent the missing entry behavior. ϵ here is a place holder, meanwhile, it will represent the invoking order. In the above example, we will write

$(Do\_3 \parallel (\epsilon;\ Do\_6))$ instead and it means Do_3 should be invoked before Do_6. We can also use the place holder ϵ to represent the missing do activity, but it does not affect the semantic meaning and we choose to remove it by default.

The sequence of entry and do behaviors invoked by firing fork transition $t1$ should be $En\_0;\ (Do\_0 \parallel En\_1;\ (Do\_1 \parallel En\_2;\ (Do\_2 \parallel ((\epsilon;\ (Do\_3 \parallel (\epsilon;\ Do\_6))) \parallel En\_4))))$, which is the same as computed by our definition.

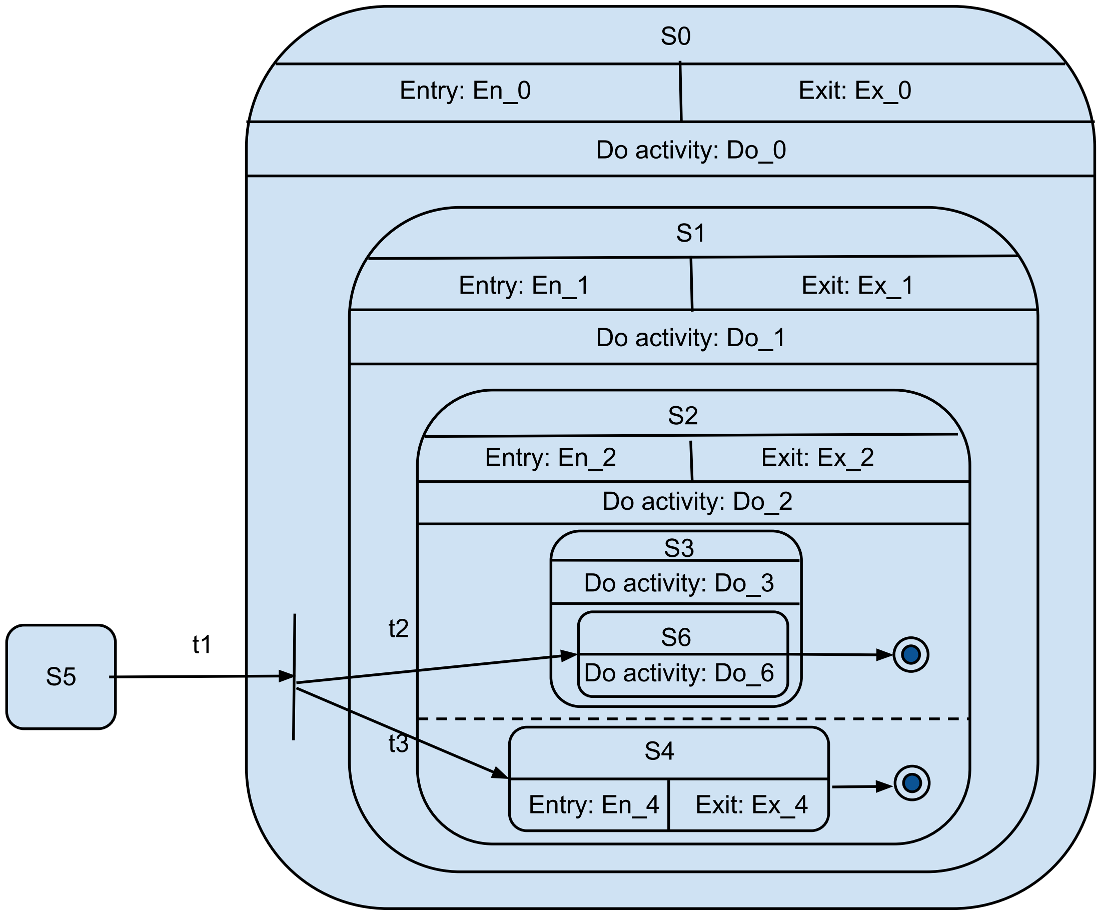


Figure 13: Collect Entry Behavior Illustration

Definition 27 (Collect Actions) *$CollectAct : Tr \times \mathcal{PSK} \times Trigger \to AL$ will collect all the effects, including exit behaviors, actions along the transition, entry behaviors and do activities, associated with the execution of the given transition. Formally, $CollectAct(t, \mathcal{PSK}_c, e) \triangleq ExitBehavior(t, \mathcal{PSK}_c);\ t.\alpha;\ EntryBehavior(t, e)$.*

Definition 28 (Event dispatch) *$\mathbb{P}Trigger \times (Trigger \times \mathbb{P}Trigger)$ will dispatch an event from the current event pool.*
Formally, $dispatch \triangleq \{(Trigger, e, Trigger') | Trigger = e \smile Trigger'\}$.

UML behavior machine specifications does not explicitly define the storage structure of events and the event dispatch orders in order to support different priority-based schemes. So we will use a general function to model the event dispatch mechanism and leave the details open for all possibilities

Here we choose to use the symbol $\smile$ instead of $\bigcup$ to represent the operation of join event e into the event pool *Trigger′* and avoid the confusion of treating the event pool as a set.

Definition 29 (Event Merge) *$(\mathbb{P}\,Trigger \times \mathbb{P}\,Trigger) \times \mathbb{P}\,Trigger$ will merge two event pools into one event pool. Formally, $Merge \triangleq \{(\epsilon, \epsilon', \epsilon")|\epsilon \smile \epsilon' = \epsilon"\}$.*

Given all the above auxiliary functions, we can finally provide the formal semantics of a UML behavior state machine. We use the form of inference rule to define the formal semantics, the formula above the line represent the premises and the formula below the line represent the conclusion, which is the formal semantics.

Firstly, we will introduce the single step rule, which describes the semantic transition step of firing a single transition.

Definition 30 (Single Step Rule)

$$\frac{t.\widehat{S} \xrightarrow[t.\alpha]{e,t.\varphi} t.\widehat{T}, t.\widehat{S} \subset \mathcal{PSK}_c, !defConflict(t, \mathcal{K}_c, e)}{\mathcal{PSK}_c \xrightarrow[CollectAction(\mathcal{PSK}_c,t,e)]{e,t.\varphi} NextPC(\mathcal{PSK}_c, t, e)} \; [\, (Trigger, !t.\alpha, Trigger' \in Merge) \,]$$

The premise indicate that the transition t has a triggering event e, which is the same as $t.\epsilon$, guard $t.\phi$ and behavior $t.\alpha$ and its source state is within the current pseudo configuration. When transition t fires, the semantics is that the state machine will transform from the current pseudo configuration $\mathcal{PSK}_c$ to the next pseudo configuration, which is computed by $NextPseudoConfig(\mathcal{PSK}_c, t, e)$ and all the behaviors happened along the execution of the transition are collected by function $CollectAction(\mathcal{PSK}_c, t, e)$.

Each action along a transition may generate new events which should be joined into the event pool. We use the operator ! to represent that the action along transition t, i.e. $t.\alpha$ can generate new events and the generated event is represented as $!t.\alpha$. This checking is done each time a single transition is executed.

Definition 31 (RTC Wandering Rule)

$$\frac{t.\widehat{S} \xrightarrow{e} \varnothing, t.\widehat{S} \subset \mathcal{K}_c, !defConflict(t, \mathcal{K}_c, e)}{\mathcal{K}_c \xrightarrow{e} \mathcal{K}_c} \; [\, RTC - Wandering \,]$$

Wandering rule is used to represent the situation when no transitions are triggered and no defferred events in the current pseudo configuration match the event e dispatched from the event pool. Then the state machine stays at the current pseudo configuration and has the dispatched event e consumed.

Definition 32 (RTC Deferral Rule)

$$\frac{e \in s.\mathit{deffer}, s \in \mathcal{K}_c, \mathit{defConflict}(t, \mathcal{K}_c, e)}{\mathcal{K}_c \xrightarrow{e} \mathcal{K}_c} \ [\ \mathit{RTC} - \mathit{Deferral}\ ]$$

Deferral rule is used to capture the situation when there exists deferral conflict and the deferred event get higher priority over the conflicting transitions. Then the state machine stays at the current pseudo configuration and the dispatched event is added back to the deferred event pool.

Definition 33 (RTC Progressing Rule)

$$\frac{\begin{array}{l} t_i.\widehat{S} \xrightarrow[t_i.\alpha]{e, t_i.\varphi} t_i.\widehat{T}, ;\ _{i\in[1,n]} t_i \in \mathit{FirTrans}(\mathcal{K}_c, e), t_i.\widehat{S} \subset \mathcal{PSK}_i, \\ t_i.\widehat{T} \subset \mathcal{PSK}_{i+1}, \mathcal{PSK}_{i+1} = \mathit{NextPC}(t_i, \mathcal{PSK}_i, e), \\ \mathcal{PSK}_1 \in \mathcal{K}, \mathcal{PSK}_n \in \mathcal{K}, !\mathit{defConflict}(t_i, \mathcal{K}_c, e) \end{array}}{;\ _{i\in[1,n]} \mathcal{PSK}_i \xrightarrow[\mathit{CollectAct}(t_i, \mathcal{PSK}_i, e)]{t_i.\epsilon, t_i.\varphi} \mathcal{PSK}_{i+1}} \ [\ \mathit{RTC} - \mathit{Progressing}\ ]$$

A Run to completion step is composed of one or more Single Steps. It represent the complete semantics of dispatching one event. The premise of the rule indicate that there are n transitions $t_i \cdots t_n$ and they form a sequence transition. Each target state set will be in the next pseudo configuration after executing the transition. The first and last pseudo configuration in the sequence both belong to the set of configuration. Actually firing each transition in the sequence transition can be seen as using the Single Step Rule, and the semantics for a run to completion step is defined as a connection of those single steps in order.

Definition 34 (Semantics of a UML state machine) *The semantics of a UML s-tate machine is defined as a Labeled Transition System (LTS)* $\mathcal{L} \triangleq (\mathbb{S}, \Rightarrow, \mathcal{S}_{\mathit{init}})$.

- $\mathbb{S} \triangleq \mathcal{K} \cup \mathbb{P}\mathit{Trigger}$ *is the state of* $\mathcal{L}$*. Trigger is the event pool associated with the current configuration, waiting for dispatching.*

- $\Rightarrow \subset \mathbb{S} \times \mathbb{S}$ *is the transition relation of* $\mathcal{L}$

- $\mathcal{S}_{\mathit{init}} = (s_0, \mathit{Trigger})$ *is the start state of* $\mathcal{L}$

$$\frac{\mathcal{K}_c \xrightarrow[\widehat{\alpha}]{e} \mathcal{K}_x}{(\mathcal{K}_c,\, \mathit{Trigger}) \xrightarrow[\widehat{\alpha}]{e} (\mathcal{K}_x,\, \mathit{Trigger}')}\ [\ (\mathit{Trigger},\, e,\, \mathit{Trigger}') \in \mathit{dispatch}\ ]$$

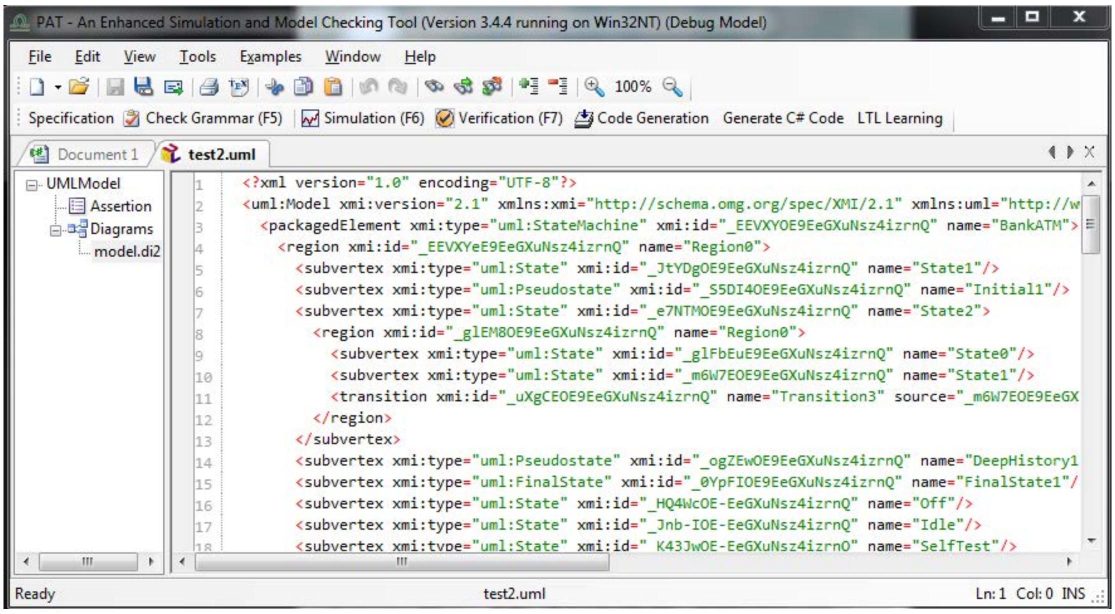
where $\mathcal{K}_x \in \mathcal{K}$ is the configuration the state machine is in after a RTS step. $\widehat{\alpha}$ represents the behaviors collected along the RTC step.

7 Future Work

In this section, we are going to discuss some promising future work related to the current problem. We firstly describe the ongoing work, implementations of the operational semantics we defined in Section 6 in PAT. Then we discuss two other related problems, i.e. formal verification of Stateflow and automatic generation of state diagrams from natural language documentation. All the work we are concerning are related to supporting formal verification of design models, which is quite important and meaningful in software engineering life cycle.

7.1 Implementation in PAT

Program Analysis Toolkit(PAT) [50] is our home grown framework to support simulation, model checking of concurrent and real time systems. PAT implements various model checking techniques which support different properties such as LTL properties with fairness assumptions, deadlock-freeness, divergence-freeness, reachability as well as probability model checking. Different models have been build inside the PAT framework, including LTS models, real-time models, CSP models as well as stateflow models etc. We are implementing our defined operational semantics of UML state machine within PAT for the purpose of directly supporting model checking UML state machine models. We



```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:uml="http://w
  <packagedElement xmi:type="uml:StateMachine" xmi:id="_EEVXYOE9EeGXuNsz4izrnQ" name="BankATM">
    <region xmi:id="_EEVXYeE9EeGXuNsz4izrnQ" name="Region0">
      <subvertex xmi:type="uml:State" xmi:id="_JtYDgOE9EeGXuNsz4izrnQ" name="State1"/>
      <subvertex xmi:type="uml:Pseudostate" xmi:id="_S5DI4OE9EeGXuNsz4izrnQ" name="Initial1"/>
      <subvertex xmi:type="uml:State" xmi:id="_e7NTMOE9EeGXuNsz4izrnQ" name="State2">
        <region xmi:id="_glEM8OE9EeGXuNsz4izrnQ" name="Region0">
          <subvertex xmi:type="uml:State" xmi:id="_glFbEuE9EeGXuNsz4izrnQ" name="State0"/>
          <subvertex xmi:type="uml:State" xmi:id="_m6W7EOE9EeGXuNsz4izrnQ" name="State1"/>
          <transition xmi:id="_uXgCEOE9EeGXuNsz4izrnQ" name="Transition3" source="_m6W7EOE9EeGX
        </region>
      </subvertex>
      <subvertex xmi:type="uml:Pseudostate" xmi:id="_ogZEwOE9EeGXuNsz4izrnQ" name="DeepHistory1
      <subvertex xmi:type="uml:FinalState" xmi:id="_0YpFIOE9EeGXuNsz4izrnQ" name="FinalState1"/
      <subvertex xmi:type="uml:State" xmi:id="_HQ4WcOE-EeGXuNsz4izrnQ" name="Off"/>
      <subvertex xmi:type="uml:State" xmi:id="_Jnb-IOE-EeGXuNsz4izrnQ" name="Idle"/>
      <subvertex xmi:type="uml:State" xmi:id="_K43JwOE-EeGXuNsz4izrnQ" name="SelfTest"/>
```

Figure 14: Interface of UML Module in PAT

show some preliminary results of the UML Module under development. Figure 14 shows the interface of the UML module. Our module accepts a UML state machine diagram in XMI(XML Metadata Interchange) format[17], which basically is a specific use of XML. We have finished the XML parser and are now working on the semantics programming part.

7.2 Automatic verification of Stateflow

Stateflow [9] is a statechart-like language and is also referred to as an industry tool developed by MathWorks company which provides a design environment used to develop state charts and flow graphs. It[18] provide a rich set of graphical language elements for the convenience of modeling. Although Stateflow's modeling language shares a lot of common features with UML state machine, they have some differences. Stateflow does not have as many graphical features as UML state machine does. Some similar features supported by Stateflow, such as condition, condition actions and transition actions cannot be simply aligned with those in UML state machine. The most important difference is that the Stateflow semantics is completely deterministic, while some pseudostates such as junction, choice and conflicting transitions introduces non-determinism into UML state machine diagrams.

Stateflow tool extends Simulink with a state chart and flow graph designing environment. It is tightly integrated with MATLAB and Simulink tool suite and benefits from the mature framework. It also supports simulation, code generation as well as some static checks, such as ill-specified truth tables checking. What is worth mentioning is that, Stateflow also supports some runtime checking, such as checking for transition conflicts, cyclic problems, state inconsistencies, data-range violations, and overflow conditions. But those checkings are quite primitive and only at syntax level. It would be exciting if we can also check some semantic properties of the underlying model, such as deadlock-freeness, reachability and LTL properties. But to the best of our knowledge, there are no such tools available now.

Stateflow's user guide is documented in [2]. It is an integrated documentation which provides usage information of the Stateflow tool as well as Stateflow language specifications. The document is written in purely naturally language and no formal semantics are provided. Since Stateflow is widely used in model-based development in embedded systems, and it has a quite different semantics from UML state machine. It is quite

[17]XMI is an OMG standard for exchange metadata information.

[18]Stateflow represents the modeling language as well as the integrated environment used to model systems. Here it refers to the language.

meaningful for us to provide a formal semantics for Stateflow, which can act as the base of developing automatic verification tools for Stateflow diagrams. There exist some related work [32] [19] towards formalizing and automatic validation of Stateflow diagrams. Grégoire and John provide an operational semantics for Stateflow [32] and built a prototype tool based on their defined semantics. This tool translate a Stateflow diagram into the SAL language, which is used by SRI's model checkers. Another approach proposed by us is to systematically translate Stateflow diagrams into CSP#, the input language of our home grown model checker PAT [50]. These translations approaches share the same disadvantage with the translation approaches of UML state machine. So we plan to provide direct model checking support to Stateflow diagrams.

7.3 Automatic generation of UML state machine diagrams

Another related work would be to apply Natural Language Processing(NLP) techniques to project documentations and automatically extract UML state machine diagrams from those documentations. There are some related work [51, 52, 29, 56, 48] which extract useful information form software-related natural language descriptions such as program comments [51, 52, 29] and API documentations [56, 48]. This provides us a clue of possibility of extracting a UML state machine model from the natural language description of a project with the help of NLP techniques.

Lin et al. [51, 52] extracts useful information, such as lock-requiring and lock-releasing constraints, program functions invocation order, from program comments as well as program source code to detect inconsistencies between program comments and source code. Those inconsistencies may indicate either bugs or bad comments which may be misleading and cause potential maintenance problems. Hao et al [56, 48] explores API documentations to extract constraints on the usage of API functions. [56] extracts constraints based on a pre-defined template which include operations such as object creation, manipulating and destroy, they also considered lock and unlock operations. [48] defined more complicated templates in order to obtain program contract information.

Those approaches all use very primitive NLP techniques such as POS tagging, shallow parsing, and some machine learning techniques such as Decision Tree Learning and Hidden Markov Model(HMM). Limited by those simple NLP techniques, those approaches can only extract constraints from simple sentences with significant "sign-post"(imperative words, topic-related keywords). We believe that applying more advanced NLP techniques will find us deeper-hidden information, which may need analysis of multiple sentences.

Automatically generate UML state machine diagrams from natural language descriptions will further automate software developing from analyzing informal natural language documents to formal verification of system models.

8 Conclusion

In this report, we thoroughly surveyed existing work on formalizing UML state machine semantics and automatic verification of UML state machine as well as tool support. We found that although a lot of work have been done in this area, most of the work just support a limited subset of UML state machine features. The automatic verification tools are all prototype tools built on top of existing model checkers based on a translation approach, which is less efficient, and hard to prove correctness. We proposed a more comprehensive semantics for UML state machine to bridge the gap. A tool which is based on the proposed semantics is being developed in the PAT [50] environment. We are also planing to support the direct model checking of Stateflow diagrams in PAT in the future. Another promising future work we are considering is to automatically generate UML state machine models from the extracted information of a project's natural language descriptions documentation.

Bibliography

[1] Ibm rational rhapsody. http://www-01.ibm.com/software/rational/uml/, 05-08-2012.

[2] Mathworks stateflow user guide, version 5. http://www.mathworks.com/help/releases/R13sp2/pdf_doc/stateflow/sf_ug.pdf.

[3] Microsoft visual modeler, http://msdn.microsoft.com/en-us/library/dd409436, 05-08-2012.

[4] Object management group. http://www.omg.org/.

[5] OMG unified language superstructure specification(formal). version 1.4, 2011-08-06. http://www.omg.org/spec/UML/1.4/PDF/index.htm.

[6] OMG unified language superstructure specification(formal). version 2.4.1, 2011-08-06. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[7] Papyrus eclipse plug-in project, http://www.eclipse.org/modeling/mdt/papyrus/, 05-08-2012.

[8] Papyrus uml tool, http://www.papyrusuml.org/scripts/home/publigen/content/templates/show.asp?L=EN&P=55&vTicker=alleza&ITEMID=3, 05-08-2012.

[9] Stateflow, http://www.mathworks.com/products/stateflow/.

[10] Wiki uml tool category, http://en.wikipedia.org/wiki/Category:UML_tools, 05-08-2012.

[11] Étienne André, Christine Choppy, and Kais Klai. Formalizing non-concurrent UML state machines using colored Petri nets. In *Proceedings of the 5th International workshop UML and Formal Methods (UML&FM)*, Paris, France, August 2012. To appear.

[12] L. Baresi and M. Pezze. On formalizing uml with high-level petri nets. *Concurrent Object-Oriented Programming and Petri Nets*, pages 276–304, 2001.

[13] M. Encarnación Beato, Manuel Barrio-Solórzano, Carlos E. Cuesta, and Pablo de la Fuente. UML automatic verification tool with formal methods. *Electron. Notes Theor. Comput. Sci.*, 127(4):3–16, April 2005.

[14] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *Arxiv preprint cs/0407038*, 2004.

[15] E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of uml state machines. In *Abstract State Machines-Theory and Applications*, pages 167–186. Springer, 2000.

[16] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. Modeling the meaning of transitions from and to concurrent states in uml state machines. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 1086–1091, New York, NY, USA, 2003. ACM.

[17] Amar Bouali, Stefania Gnesi, and Salvatore Larosa. The integration project for the jack environement. *BULLETIN OF THE EATCS*, 54:207–223, 1994.

[18] Egon Brger, Alessandra Cavarra, and Elvinia Riccobene. On formalizing UML state machines using asms. *Information Software Technology*, 46(5):287, 2004.

[19] C. Chen, J. Sun, Y. Liu, J.S. Dong, and M. Zheng. Formal modeling and validation of stateflow diagrams. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–19, 2012.

[20] C. Choppy, K. Klai, and H. Zidani. Formal verification of uml state diagrams: a petri net based approach. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.

[21] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, pages 359–364, London, UK, UK, 2002. Springer-Verlag.

[22] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. Technical report, 2000.

[23] M.L. Crane and J. Dingel. On the semantics of UML state machines: Categorization and comparision. In *In Technical Report 2005-501, School of Computing, Queens.* Citeseer, 2005.

[24] G. Del Castillo and K. Winter. Model checking support for the asm high-level language. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, 2000.

[25] Jori Dubrovin and Tommi Junttila. Symbolic model checking of hierarchical UML state machines. Technical Report B23, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2007.

[26] Jori Dubrovin, Ab Teknillinen Korkeakoulu, and Jori Dubrovin. Jumbala– an action language for uml state machines.

[27] H. Fecher and J. Schönborn. UML 2.0 state machines: Complete formal semantics via core state machine. *Formal Methods: Applications and Technology*, pages 244–260, 2007.

[28] H. Fecher, J. Schönborn, M. Kyas, and W.P. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. *Formal Methods and Software Engineering*, pages 52–65, 2005.

[29] Zachary P. Fry, David Shepherd, Emily Hill, Lori L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verbdirect object pairs in all the right places. *IET Software*, pages 27–36, 2008.

[30] S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using jack. In *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, pages 46–55. IEEE, 1999.

[31] H. Gomaa. Validation of dynamic behavior in uml using colored petri nets. 2000.

[32] G. Hamon and J. Rushby. An operational semantics for stateflow. *Fundamental Approaches to Software Engineering*, pages 229–243, 2004.

[33] D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[34] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *Software Engineering, IEEE Transactions on*, 16(4):403–414, 1990.

[35] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[36] J.W. Janneck and P.W. Kutter. *Mapping automata: simple abstract state machines.* TIK-Report. Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zürich (ETH), 1998.

[37] Harald Fecher Jens Schonborn, Willem Paul de Roever and Marcel Kyas. Formal semantics of UML 2.0 behavioral state machines. Technical report, Institute of Computer Science and Applied Mathematics, Technical Faculty, Christian-Albrechts-University of Kiel, 2005.

[38] K. Jensen and L.M. Kristensen. *Coloured Petri nets: modeling and validation of concurrent systems.* Springer-Verlag New York Inc, 2009.

[39] Y. Jin, R. Esser, and J.W. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and Systems Modeling*, 3(2):150–163, 2004.

[40] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model checking dynamic and hierarchical uml state machines. *Proc. MoDeV2a: Model Development, Validation and Verification*, pages 94–110, 2006.

[41] A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *Proceedings of 5th Workshop on Tools for System Design and Verification, Technical Report*, volume 11, pages 59–64, 2002.

[42] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking - timed uml state machines and collaborations. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems: Co-sponsored by IFIP WG 2.2*, FTRTFT '02, pages 395–416, London, UK, UK, 2002. Springer-Verlag.

[43] Gihwon Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In *Proceedings of the 3rd international conference on The unified modeling language: advancing the standard*, UML'00, pages 528–540, Berlin, Heidelberg, 2000. Springer-Verlag.

[44] D. Latella, I. Majzik, M. Massink, et al. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the IFIP TC6/WG6*, volume 1, page 465, 1999.

[45] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Aspects of Computing*, 11(6):637–664, December 1999.

[46] Johan Lilius and Iván Porres Paltor. Formalising UML state machines for model checking. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 430–444, Berlin, Heidelberg, 1999. Springer-Verlag.

[47] Johan Lilius and Ivn Porres Paltor. The semantics of UML state machines. Technical report, 1999.

[48] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

[49] Wuwei Shen, Kevin Compton, and James Huggins. A toolset for supporting UML static and dynamic model checking. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 147–152, Washington, DC, USA, 2002. IEEE Computer Society.

[50] J. Sun, Y. Liu, J. Dong, and J. Pang. Pat: Towards flexible verification under fairness. In *Computer Aided Verification*, pages 709–714. Springer, 2009.

[51] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: bugs or bad comments?*/. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 145–158, New York, NY, USA, 2007. ACM.

[52] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 11–20, New York, NY, USA, 2011. ACM.

[53] D. Varró. A formal semantics of uml statecharts by model transition systems. *Graph Transformation*, pages 378–392, 2002.

[54] M. Von Der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.

[55] S.J. Zhang and Y. Liu. An automatic approach to model checking uml state machines. In *Secure Software Integration and Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference on*, pages 1–6. IEEE, 2010.

[56] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 307–318, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

Limitation of EHA

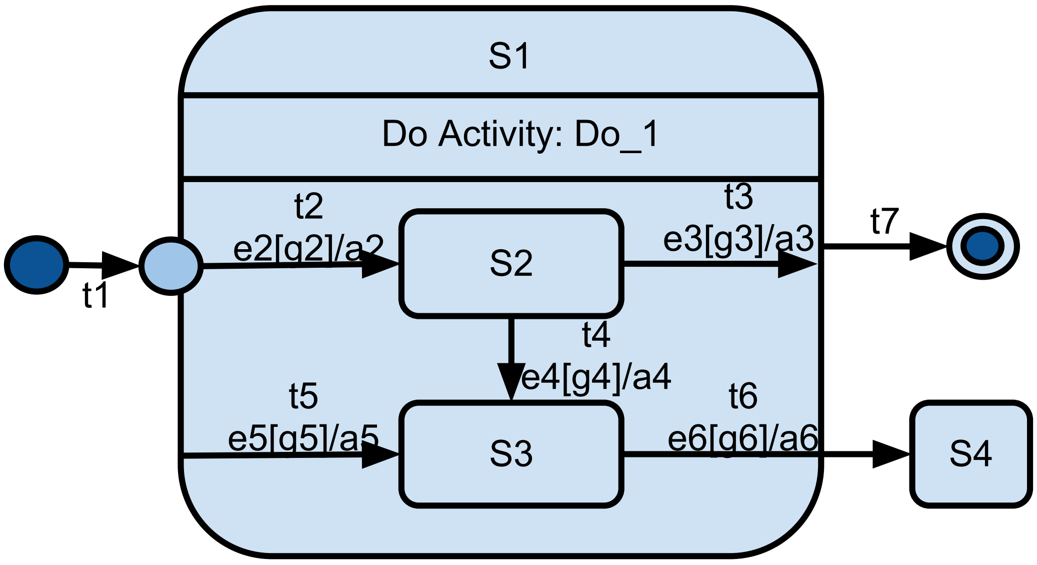


Figure A.1: Transitions cannot be expressed in EHA

We show in this section that EHA cannot represent local transitions.

> "A transition with kind local must have a composite state or an entry point as its source."
>
> [6, Chapter 15.3.15, Constraints, p.590]

In Figure A.1, transiton $t2$ and $t5$ are all local transitions, which emanating from a composite state to its containing substate.

In an EHA, internal transitions, i.e. transitions which do not cross state boundaries, are translated to the Sequential Automata which represent is containing composite s-tate. For example in Figure A.1, transtion $t4$ should be translated into a transition which belongs to a Sequential Automata that represent composite state $S1$. Interlevel transitions such as $t6$ in Figure A.1 should be translated into a transition in a Sequential

Automata which is the direct container composite state of its main source/target state. In the example shown in Figure A.1, transition $t6$ should be translated into a transition of a Sequential Automata which represents the outermost region, i.e. the state machine itself. With certain extra information recorded(source restriction and target determinator [44]), interlevel transitions can be recognized correctly.

But transition $t3$, $t5$ are out of the capability of EHA. Hierarchical Automata requires a strict hierarchical structure. The existence of interlevel transitions and local transitions break the hierarchical structure. EHA just extend the Hierarchical Automata to deal with interlevel transitions. But transitions like $t3, t5$ are from a state to its container state, or from composite states to its containing states. It is impossible to represent such transitions in an EHA since states in different hierarchies cannot appear in a single Sequence Automata. This may threat the usage of EHA in formalizing UML state machines.

Appendix B

List of Symbols

Symbols

$\frown$	concatenation operator
$\|\|$	parallel operator
;	sequencing operator
$\smile$	merge operator
!	output of an operation
$\uparrow$	container operation
$\Uparrow$	direct container operation
$\downarrow$	substate operation
$\Downarrow$	direct substate operation
$\upharpoonleft$	substate difference operation

Plain letters

B	The behavior type
C	choice psuedostate type
$Deffer$	deferred event of a state
do	do activity of a state
En	entry point pseudostate type
Ex	exit point pseudostate type
$entry$	entry behavior of a state
$exit$	exit behavior of a state
F	fork pseudostate type
G	guard constraint type
H^*	deep history pseudostate type
H	shallow history pseudostate type
I	initial pseudostate type
ID	The unique identifier of a state machine construct
J	join pseudostate type
$Junc$	junction pseudostate type
$\mathcal{K}$	configuration
$\mathcal{K}_c$	current configuration
$\mathcal{K}_x$	next configuration
$\mathcal{L}$	Labeled transition system
LCA	Least Common Ancestor operation
M	state machine

$\mathbb{N}$	set of natural numbers
$\mathbb{P}$	power set operation
PS	Set of Pseudostates
$\mathcal{PSK}$	set of pseudo configuration
$\mathcal{PSK}_c$	the current pseudo configuration
R	Set of region names
S	Set of state names
$\mathbb{S}$	states in LTS
SAP	source action pair, component of a transition
$\mathcal{S}_{init}$	initial state of LTS
S_f	Set of final state names
Src	source state set of transitions
STr	sequence transition set
T	terminate pseudostate type
TAP	target action pair, component of a transition
Trg	target state set of transitions
Tr	set of transitions
$Trigger$	The triggering event type

Greek letters

ι	Scope of a transition
ϵ	empty items
φ	guard constraint of a transition
α	effect behavior of a transition