A Formal Semantics for the Complete Syntax of UML State Machines with Communications

Shuang Liu[1], Yang Liu[2], Étienne André[3], Christine Choppy[3], Jun Sun[4], Bimlesh Wadhwa[1] and Jin Song Dong[1]

[1] School of Computing, National University of Singapore, Singapore
[2] Nanyang Technology University, Singapore
[3] LIPN, Université Paris 13, Sorbonne Paris Cité, France
[4] Singapore University of Design and Technology, Singapore

Abstract. UML is a widely used notation introduced by the Object Management Group (OMG), and formalizing its semantics is an important issue. In this work, we concentrate on formalizing UML state machines which are used to express the dynamic behavior of software systems. We propose a formal operational semantics covering all features of the latest version (2.4.1) of UML state machine specification. We use Labeled Transition System (LTS) as the semantic model of UML state machines, which is subject to automatic verification techniques like model checking. Furthermore, our proposed semantics includes synchronous and asynchronous communications between state machines. We implement our approach in USM^{2}C, a model checker supporting editing, simulation and automatic verification of UML state machines. Experiments show the effectiveness of our approach.

1 Introduction

UML diagrams [1] have become the *de facto* modeling language, and UML state machine diagrams are widely used to model the dynamic behavior of an object. Since UML specification is documented in natural language, inconsistencies and ambiguities arise, and it is thus important to provide a formal semantics for UML. Indeed, a formal UML semantics (1) allows more precise and efficient communication between engineers, (2) yields more consistent and rigorous models, and (3) lastly and most importantly, enables automatic formal verification of UML state machine models through techniques like model checking, which guarantees important properties of a system in the early development stage.

Some existing works provide formal semantics for a subset of UML state machine features, leaving some important issues unaddressed. First, none of the existing formalization approaches achieve a full coverage of UML state machine features, and only a few [5,2] consider UML 2.x specifications. To the best of our knowledge, only [11] considered the non-determinisms in the presence of orthogonal composite states. However, the work in [11] supports only a limited set of syntax features, e.g., no pseudostates except for initial and history are supported. We believe that all the features provided by UML state machine specification should be considered, since each of them has its specific usage, especially choice, fork, join pseudostates, completion transitions and event deferral, which are commonly used but often left out in existing formalizations.

Secondly, in the existing approaches, communications between state machines are not formally defined. UML state machines are used to model the behavior of objects, which are components of a system. The whole system may include several state machines interacting with each other synchronously or asynchronously. The dynamic behavior of those state machines constitute the dynamic behavior of the whole system. From the viewpoint of the overall system behavior, especially due to synchronizations among different components of the system, the verification of the entire system is more meaningful than its subparts, which are in turn modeled by respective state machines.

Lastly, the unclarities (that is, inconsistencies and ambiguities) in the UML state machine specifications are not thoroughly checked and discussed. Fecher et al. [6] discussed 29 unclarities in UML 2.0 state machines. But there are still some unclarities, such as the granularity of a transition execution sequence and container of a transition etc, which are not covered in [6] but will be discussed by our approach (Section 2.2).

In order to bridge the gaps in the current approaches, we provide a formal operational semantics for the complete set of UML state machine features, which includes formal definition of state machine level and orthogonal composite state level non-determinism. We also consider the communication mechanisms between different state machines. The contributions of this paper are summarized as follows. (1) We provide a formal operational semantics for UML 2.4.1 state machines covering the complete set of UML state machine features. In particular, our formalization considers state machine level and orthogonal composite state level non-determinism as well as synchronous and asynchronous communications between state machines. (2) We explicitly discuss the event pool mechanisms in UML state machines and consider deferral events as well as completion events. (3) We exhibit 6 new unclarities in UML 2.4.1 state machine semantics specifications. (4) We develop a self-contained tool USM^{2}C based on the semantics we have defined; it is able to model check various properties such as deadlock-freeness and linear temporal logic (LTL) properties. We conduct experiments on our tool and results show the effectiveness of our tool.

Related Works Due to limited space, we only discuss the most related works; in particular, we do not mention work focusing on the 1.x UML specification. Fecher [5] provided a formal semantics for a subset of UML state machine features. The remaining subset of UML state machine features are informally transformed to the defined subset of features. The semantics defined in [5] blurs the run to completion (RTC) step, which is the basic semantic step of UML state machine. Moreover, the informal transformation procedure as well as the extra costs it introduces might make it infeasible for automatic tool developing. Another work [11] by Schönborn considered non-determinism in orthogonal composite states. But in terms of pseudostates, only initial and history pseudostates are covered. Moreover, this approach does not define a complete RTC step semantics. A subset of UML state machine features is also covered in works like, e.g., [12] that adopts Labelled Transition System (LTS) as a semantic domain to formalize UML state machine semantics. But constructs such as junction, choice, fork and join pseudostates, submachine state etc. are not supported. Jin et al. [7] use Abstract State Machines (ASM) as the semantic domain and do cover more features, but choice pseudostate is not considered. Moreover, some of their formalizations, such as deciding conflicts in the presence of deferred events, do not respect UML2.x specifications. Recently, a few proposals have been made to formalize UML state machine semantics into Petri nets [3,2]. These approaches also do not support a number of pseudostates. The input languages to model checking tools (Spin, PAT, etc.) are used in other approaches, e.g., [9,13]. Due to the limitation of the translated language, only a small subset of UML state machine features are supported. It is also hard to link back to the original model when a counterexample is detected.

The rest of this paper is organized as follows. Section 2 provides the preliminaries of UML state machines, exhibits new unclarities, and defines basic assumptions for our work. Section 3 defines the syntax for UML state machines, including the event pool formalization. Section 4 defines the formal semantics for UML state machines with communications. Section 5 provides the implementation details and evaluation results. Section 6 concludes the paper and discusses future directions of research.

2 UML Features, Unclarities and Our Assumptions

In this section, we introduce the preliminary knowledge about UML state machine. Then, we exhibit unclarities that we found out in the UML 2.4.1 specification. Finally, we provide basic assumptions for our approach.

2.1 Introduction of Basic Features of UML State Machines

We briefly introduce basic features of UML state machines in this section. We use the *RailCar* system Fig. 1 (a modified version of the example used in [4]) as a running example. The *RailCar* system is composed of *Car* state machine and *Handler* state machine. They communicate with each other through synchronous event calls. The *Handler* state machine models a part of a terminal behavior, which is responsible of communicating with the *Car* state machine when the car is approaching and departing the terminal.

Vertices and Transitions. A vertex is a node, which refers to a state, a pseudostate, a final state or a connection point reference. A transition is a relation between a source vertex and a target vertex. It may have a guard, a trigger and an effect (a sequence of actions). The container of a transition is the region which owns the transition. A compound transition is composed of a multiple transitions joined via choice, junction, fork and join pseudostates.

Regions. It is container of vertices and transitions, and represents an orthogonal parts of a composite state or a state machine. In Fig. 1, the areas [R1] and [R2] are regions.

States. There are three kinds of states, viz., simple state (e.g., in Fig. 1: *Idle*), composite state (*Departure*) and submachine state. An orthogonal composite state (*WaitArrivalOK*) has more than one region. States can have optional entry/exit/do behaviors. A do behavior can be interrupted by an event. A state can also define a set of deferred events. A final state (*Final1*) is a special kind of state which indicates finishing of its enclosing region. It does not have regions, nor entry/exit/do behaviors.

Pseudostates. Pseudostates are introduced to connect multiple transitions to form complex transition paths. There are 10 kinds of pseudostates: initial, join, fork, junction, choice, entry point, exit point, shallow history, deep history, terminate. Due to lack

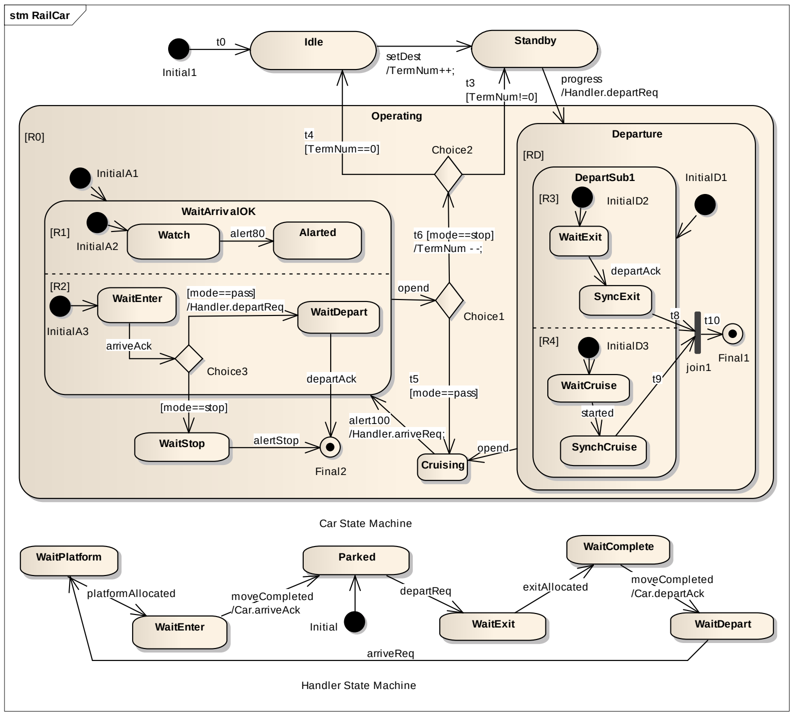

Fig. 1. The RailCar State Machine

of space, details are given in [10], while some commonly used features are discussed below. Join pseudostate (*join1*) is used to merge transitions from states in orthogonal regions. Fork pseudostate is used to split transitions targeting states in orthogonal regions. Choice pseudostates (*Choice1*) represent a dynamic branching point. When a choice pseudostate is encountered, the transition path emanating from it should be evaluated in the environment when the choice pseudostate is reached (and not in the beginning of the compound transition). Initial Pseudostate (*Initial1*) indicates the default initial state of a region.

Connection Point Reference. It is an entry/exit point of a submachine state. It refers to the entry/exit pseudostate of the state machine that the submachine state refers to.

Active State Configuration. It is a set of active states of a state machine when it is in a stable status[1], e.g., {*Idle*} or {*Operating, Final2*} in Fig. 1.

Run to Completion Step (RTC). It captures the semantics of processing one event occurrence, i.e., executing a set of compound transitions (fired by the dispatched event), which may cause the state machine to move to the next active state configuration, together with behavior executions. This is the basic semantic step in UML state machines. For example in Figure 1, if the current active state configuration is {*Standby*} and the transition emanating from it is fired, the RTC step will lead the state machine to

[1]The state machine is waiting for event occurrences.

the next active state configuration, i.e., $\{$*Operating, Departure, DepartSub1, WaitExit , WaitCruise*$\}$, accompanied by the behavior execution to call the *departReq* behavior of *Handler* state machine. The RTC step does not finish until the call event returns from *Handler* state machine.

2.2 Unclarities in UML 2.4.1 State Machine Specification

Due to lack of space, we briefly sketch below some new unclarities we found in the UML state machine specification (detailed discussions can be found in [10]).

Transition Execution Sequence. Transitions and compound transitions are used interleavinglly in the descriptions of transition execution sequence, which makes it unclear whether some rule is applied to a transition or to a compound transition. The transition execution ordering is important since different execution orders may lead to different results.

Basic Interleave Execution Step. If multiple compound transitions in orthogonal regions are fired by the same event, it is unclear in what granularity should the interleaving execution be conducted, either on transition or on compound transition level.

Order issue of entering orthogonal composite states. When entering orthogonal composite states, no interleaving order is specified.

There are three other unclarities, viz., the container of a transition, the Least Common Ancestor (LCA) of a compound transition and the conflict resolutions in the presence of choice pseudostates are not clearly defined. We provide the detailed discussions in [10] due to space limits.

2.3 Basic Assumptions on UML State Machine Semantics

In this section, we try to address the unclarities discussed in Section 2.2, and we provide the basic assumptions for our semantics definition.

Transition Execution Sequence. Whether the transition execution sequence is defined on a single transition or on a compound transition is not clearly stated. We define the transition execution sequence based on a transition instead of compound transitions. In this way, we can guarantee that behaviors are executed in a correct order.

Basic Interleave Execution Step. For interleaving execution of compound transitions in orthogonal regions, we decide to regard a compound transition as the interleaving execution step since a compound transition is a semantically complete path.

Order issues of entering orthogonal composite states. On entering an orthogonal composite state, all possible interleaving orders among its substates to be entered are allowed, as long as the hierarchical order is preserved, i.e., composite states are entered before their substates.

3 Syntax of UML State Machines

In this section, we provide formal syntax definitions for UML state machine features and abstractions of event pools. We define a self-contained model which includes multiple state machines. Table 1 lists the basic notations of all types defined in the work.

Table 1. Type Notations

Symbol	Type	Symbol	Type	Symbol	Pseudostate type
$\mathcal{K}_\mathcal{S}$	active state configuration	$\mathbb{B}$	boolean	SH_{ps}	deep history
$\widetilde{T}$	compound transition	C	constraints	I_{ps}	initial
$\mathcal{K}$	configurations	S_f	final state	C_{ps}	choice
$\langle\widetilde{T}\rangle$	compound transition list	S	state	Jo_{ps}	join
V	vertex	$Trig$	triggers	Ju_{ps}	junction
$\mathcal{K}_\mathcal{V}$	active vertex configuration	T	transition	T_{ps}	terminate
CPR	connection point reference	E	event	En_{ps}	entry point
SM	state machine	R	region	F_{ps}	fork
B	behaviors	PS	pseudostate	SH_{ps}	shallow history
$\langle B\rangle$	behavior list	$\mathbb{N}$	natural number	Ex_{ps}	exit point

3.1 Syntax Formalization

We use tuples as syntax domain and refer to [1] as the basis for syntax definition.

Definition 1 (State). *A state is defined as a tuple $s = (\widehat{r}, \widehat{t_{def}}, \alpha_{en}, \alpha_{ex}, \alpha_{do}, \widehat{en}, \widehat{ex}, \widehat{cpr}, sm, \widehat{t})$ where:*

- *$\widehat{r} \subset R$ is the set of regions directly contained in this state.*
- *$\widehat{t_{def}} \subset Trig$ is the set of deferral triggers associated with this state.*
- *$\alpha_{en} \in B$,$\alpha_{ex} \in B$ and $\alpha_{do} \in B$ represent the entry, exit and do behaviors associated with the state respectively.*
- *$\widehat{en} \in PS$ and $\widehat{ex} \in PS$ are the entry point reference and exit point reference associated with the state.*
- *$\widehat{cpr} \subset CPR$ is a set of connection point references belonging to a submachine state. This field is used only when the state is a submachine state.*
- *$sm \in SM$ is the state machine referenced by this state. This is used only when the state is a submachine state.*
- *$\widehat{t} \subset T$ is the set of internal transitions defined in the state.*

There are four kinds of state types S_s, S_c, S_o and S_m, that represent simple state, composite state, orthogonal composite state and submachine state, respectively.

Definition 2 (Pseudostate). *A* pseudostate *is defined as a tuple $ps = (\iota, \widehat{h})$, where $\iota \in R$ is the region in which the pseudostate is defined, and $\widehat{h} \in S$ is an optional field which is used to record the last active set of states. This latter field is only used when the pseudostate is a shallow history or deep history pseudostate.*

There are ten kinds of pseudostates defined in UML 2.4.1 state machine specifications. The last column of Table 1 shows the notations of different kinds of pseudostates. We use PS to represent all kinds of pseudostates.

Definition 3 (Final state). *A* final state *is a special kind of state, which is defined as a tuple $fs = (\iota)$ where: $\iota \in S_o \cup S_c$ is the composite state which is the direct ancestor of the container of the Final State.*

Definition 4 (Connection Point Reference). *A* Connection Point Reference *is defined as a tuple $(\widehat{en}, \widehat{ex}, s)$ where $\widehat{en} \subset En_{ps}$ and $\widehat{ex} \subset Ex_{ps}$ are the entry point and exit point kind pseudostates corresponding to this connection point, and s is the state in which the connection point reference is defined.*

Vertex $V \triangleq S \cup S_f \cup PS \cup CPR$ is an abstraction of all nodes. It is the superclass of State, Final state, Pseudostate and Connection Point Reference.

Definition 5 (Transition). *A* transition *is a tuple* $t = (sv,\ tv,\ \widehat{tg},\ g,\ \alpha\ ,\ \iota,\ \widehat{tc})$ *where:*

- $sv \in V$, $tv \in V$ *are the source and target vertex of the transition respectively.*
- $\widehat{tg} \subset Trig$, $g \in C$ *and* $\alpha \in B$ *are the set of triggers, the guard and the effect behavior associated with the transition respectively.*
- $\iota \in R$ *is the container of the transition.*
- $\widehat{tc}$ *is a set of tuples of the form* $segt = (ss, \alpha_{st}, \iota_{st})$. *It represents the special situation that a join or fork pseudostate connects multiple transitions to form a compound transition. Each tuple represents a segment transition which ends in the join (resp. emanates from the fork) pseudostate.* $ss \in S$ *is the non-fork (resp. non-join) end of the segment transition* [2], $\alpha_{st} \in B$ *is the behavior associated with the segment transition.* $\iota_{st} \in R$ *is the container of the segment transition.*

We treat exit point pseudostate the same way with join pseudostate and entry point pseudostate the same way with fork pseudostate.

We define the following functions on transitions for the benefit of a clear notation. Function $isFork(t)$ and $isJoin(t)$ decides whether the transition t is a fork transition and join transition respectively. We use $t.\widetilde{\alpha}$ to represent all possible action execution sequences of t. Formal definition of $t.\widetilde{\alpha}$ is in [10].

Definition 6 (Region). *A* region *is defined as a tuple* $r \triangleq (\widehat{v}, \widehat{t})$ *where:* $\widehat{v} \subset (S \cup PS)$ *is the set of vertices directly contained in this region.* $\widehat{t} \subset T$ *is the set of transitions owned by this region.*

Definition 7 (State Machine). *A UML* state machine *is defined by a tuple* $sm \triangleq (\widehat{r}, \widehat{cp})$, *where* $\widehat{r}$ *is top most region which is directly contained by* sm, *and* $\widehat{cp}$ *are the connection points, i.e., entry/exit point pseudostates defined for this state machine.*

Definition 8 (Compound Transition). *A* compound transition $\tilde{t}$ *is a "semantically complete" path composed of one or multiple transitions connected by pseudostates. The set of compound transition* $\widetilde{T} = \{\tilde{t} \mid \tilde{t} \in ST \wedge \tilde{t}.\widehat{sv} \in S \wedge \tilde{t}.\widehat{tv} \in S\}$. *where* $st \in ST \equiv st \in T \vee \exists\, st_i, st_j \in ST : last(st_i) = first(st_j) \wedge st = st_i \frown st_j$.

The operator $\frown$ represents the operation of connecting transitions in order. We define function $len(\tilde{t})$ to compute the total number of segment transitions the compound transition is composed of. And $seg(\tilde{t}, i)$ returns the ith segment specified by the natural number index (i) of a given compound transition. We use $first(\tilde{t})$ and $last(\tilde{t})$ to represent the first and last segment of $\tilde{t}$ (formal definitions are in [10]).

Compositional Operators. The operator "; " is used to represent a sequential composition. Interleave operation ($|||$) represents a non-determinism in the execution orders of all the involved objects. Interleaving composition with synchronous communications ($|||^C$) is a special case of interleaving: it requires the state machine to synchronize on the specified event indicated by C. Interruption (∇) is used to represent interruption of a do activity by some event occurrence. Parallel composition ($||$) represents a real concurrency, i.e., execute at the same time.

[2]The other end is the fork (resp. join) pseudostate.

Definition 9 (System). *A* system *is a set of state machines executing interleavingly (with synchronous communications).* $sys \triangleq |||_{i \in [1,n]}^{C} Sm_i$ *where* $Sm \triangleq (sm, EP, GV)$*. Fields of* Sm *represent state machine* (sm)*, event pool associated with* sm *and global shared variables of* sm *respectively.* n *is the number of state machines within* sys*.*

Event Pool Abstraction. Events of different types, such as change events and signal events, are processed differently. Events with same type but appearing in different places, such as the trigger of a transition and in the deferred event set of a state, are also processed differently. Change events have the highest priority during event dispatching. A deferred event should always be checked to decide its status in each active state configuration. We provide for this purpose three separate event pools, viz., completion event pool (CEP), deferred event pool (DEP), and normal event pool (NEP). But the event dispatching order in each pool is not specified. We use $EP \triangleq (CEP, DEP, NEP)$ to represent the event pool of a state machine and define the two basic operations on EP. $Merge(e, P)$ represents merge event e into the corresponding event pool represented by P, and $!EP$ represents dispatch an event from EP. (Formal definitions are in [10].)

4 A Formal Semantics for UML State Machines

This section devotes to a self-contained formal semantics for all UML state machine features. We have adopted the semantic model of Labeled Transition Systems (LTS). The dynamic semantics of a UML state machine is captured by the execution of RTC steps, which have two kinds of effects, viz., changing active states and executing behaviors. We formally define the two kinds of changes separately. Then the semantics of the RTC step is defined formally. At last, we define the semantics of the system. Remind that for all the following definitions, we shall assume the notations in Table 1.

4.1 Active State Configuration Changes

An active state configuration $\mathcal{K_S}$ is a set of states which are in active status at the same time. It describes a stable state status of a UML state machine execution, i.e., the status when the previous RTC step finishes. Remind that we define the transition execution sequence based on transitions, which may emanate from or target pseudostates. So we use Active Vertex Configuration $\mathcal{K_V}$ to represent the snapshots of a UML state machine during an RTC execution. An active vertex configuration is a set of vertices that are in active status at the same time. $\mathcal{K_S}$ and $\mathcal{K_V}$ are defined formally in [10].

Next Active State Configuration. $NextK : \mathcal{K_S} \times \langle \widetilde{T} \rangle \rightarrow \mathcal{K_S}$ is a function that computes the next active state configuration after executing the list of compound transitions. Formally: $NextK(ks, (\tilde{t}_1;\ \ldots;\ \tilde{t}_n)) \triangleq NxK(ks_n, \tilde{t}_n)$, where $\forall\, i \in [2, n], ks_i = NxK(ks_{i-1}, \tilde{t}_{i-1}) \wedge ks_1 = ks$. Function $NxK : \mathcal{K_S} \times \widetilde{T} \rightarrow \mathcal{K_S}$ computes the next active state configuration after executing a compound transition. Formally, we have: $NxK(ks, \tilde{t}) \triangleq NxPK(kv_n, seg(\tilde{t}, n))$, where $n = len(\tilde{t})$, $kv_1 = ks$, and $\forall\, i \in [2, n], kv_i = NxPK(kv_{i-1}, seg(\tilde{t}, i-1))$. Function $NxPK : \mathcal{K_V} \times T \rightarrow \mathcal{K_V}$ computes the next active vertex configuration after executing a transition. Formally: $NxPK(kv, t) \triangleq kv \backslash Leave(kv, t) \cup Enter(t)$. Functions $Leave$ and $Enter$ represent the set of states left and entered after executing a transition and are formally defined in [10].

4.2 Behavior Execution

Another effect of executing an RTC step is to cause behaviors to be executed. All the behaviors should be collected in the correct order. We define the following functions to collect the behavior execution sequence.

Exit Behavior. $\mathit{ExitBehavior} : \mathcal{K}_{\mathcal{V}} \times T \to \langle B \rangle$ collects the ordered exit behaviors of states that a given transition leaves in the current vertex configuration. Formally:

$$\mathit{ExitBehavior}(kv, t) = \mathit{ExitV}(kv, \mathit{MainSource}(t), t)$$

$$\mathit{ExitV}(kv, v, t) \triangleq \begin{cases} |||^{C}_{r \in v.\widehat{r}} \mathit{ExitR}(kv, r, t);\ v.\alpha_{do} \nabla v.\alpha_{ex} & \text{if } v \in S_o \\ \mathit{ExitR}(kv, r, t);\ v.\alpha_{do} \nabla v.\alpha_{ex} & \text{if } v \in S_c \\ v.\alpha_{do} \nabla v.\alpha_{ex} & \text{if } v \in S_s \\ \epsilon & \text{otherwise} \end{cases}$$

$$\mathit{ExitR}(kv, r, t) \triangleq \begin{cases} \mathit{SetSH}(h, v);\ \mathit{ExitV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge v \in S \\ & \wedge\, \exists\, h \in \mathit{SH}_{ps} : h \in r.\widehat{v} \\ & \\ \mathit{SetDH}(h, v);\ \mathit{ExitV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge v \in S \\ & \wedge\, \exists\, h \in \mathit{DH}_{ps} : \mathit{isAncestor}(h.\iota, r) \\ & \wedge\, \mathit{isAncestor}(t.\iota, h.\iota) \\ & \\ \mathit{ExitV}(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \\ & \wedge\, \forall\, s' \in r.\widehat{v}, s' \notin \mathit{SH}_{ps} \\ & \wedge\, \nexists h \in \mathit{DH}_{ps} : \mathit{isAncestor}(h.\iota, r) \\ & \wedge\, \mathit{isAncestor}(t.\iota, h.\iota) \end{cases}$$

The exit behaviors of executing a transition are collected recursively starting from its main source state (computed by function $\mathit{MainSource}(t)$). Exit behaviors should be collected in an innermost-out order. We define function ExitV and ExitR to recursively collect exit behaviors (from vertices and regions respectively). For orthogonal and composite states, all their orthogonal regions should be exited before it. If the region to be exited contains a shallow history or deep history pseudostate, the content of the history pseudostate should be set properly (by functions SetSH and SetDH respectively) before exiting the region. Exiting simple states means terminates the do behavior (if any) and executes the exit behavior, as defined by function exit. Otherwise, a pseudostate must be encountered and no behavior is collected (denoted by ϵ).

Entry Behavior. $\mathit{EntryBehavior} : T \to \langle B \rangle$ collects the ordered entry behaviors of states a given transition enters. Formally:

$$\mathit{EntryBehavior}(t) = \mathit{EntryV}(\mathit{MainTarget}(t), \mathit{Enter}(t))$$

$$\mathit{EntryV}(v, \widehat{V}) \triangleq \begin{cases} v.\alpha_{en};\ (|||^{C}_{r \in v.\widehat{r}} \mathit{EntryR}(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_o \\ v.\alpha_{en};\ (\mathit{EntryR}(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_c \\ v.\alpha_{en};\ v.\alpha_{do} & \text{if } v \in S_s \\ \mathit{GenEvent}(v.\iota) & \text{if } v \in S_f \wedge \forall\, r \in v.\iota.\widehat{r}, \\ & \exists\, s' \in r.\widehat{v} : s' \in kv \Rightarrow s' \in S_f \\ & \\ \epsilon & \text{otherwise} \end{cases}$$

$$EntryR(r, \widehat{V}) \triangleq EntryV(s', \widehat{V}) \textit{ where } r \in R \land s' \in r.\widehat{v} \land s' \in \widehat{V}$$

Entry behaviors are collected in a similar manner to exit behaviors, except that the order should be outermost-in. We define the function $EntryV$ and $EntryR$ to recursively collect the entry behaviors of all the vertices in $\widehat{V}$ in order. All the states entered by firing the transition t are computed by function $Enter(t)$. Starting from the main target state of a transition, if the state is an orthogonal composite state, then all its orthogonal regions are entered interleavingly. Entering each state means executing its entry behavior followed by its do activities ($s.\alpha_{en};\ s.\alpha_{do}$) if any. Do activities of a composite state should be executed in parallel ($\|$) with all the behaviors of its containing states. Function $GenEvent(s)$ generates a completion event for state $s.\iota$ (the container state of final state s) and merges the generated event in the completion event queue (CEQ). For orthogonal composite states, we can only generate a completion event when active states in all its regions are final states .

Collect Actions. $CollectAct : \mathcal{K_S} \times \widetilde{T} \rightarrow \langle B \rangle$ collects the ordered sequence of behaviors, associated with the execution of the given compound transition. Formally:

$$CollectAct(ks, \tilde{t}) \triangleq Act(kv_1, seg(\tilde{t}, 1));\ \ldots;\ Act(kv_i, seg(\tilde{t}, i));\ \ldots;\ Act(kv_n, seg(\tilde{t}, n))$$

$$Act(kv, t) \triangleq ExitBehavior(kv, t);\ t.\widetilde{\alpha};\ EntryBehavior(t)$$

where $n = len(\tilde{t})$, $kv_1 = ks$ and $kv_i = NxPK(kv_{i-1}, \tilde{t}_{i-1})$ for $i \in [2, n]$.

4.3 The Run to Completion Semantics

The effects of an RTC step execution include both active state changes and behavior executions which may cause the event pool and global shared variables to be updated. We use the term *configuration* to capture the stable status (including states, event pool and global shared variables) of a UML state machine.

Definition 10 (Configuration). *A* configuration *is a tuple $k = (ks, EP, GV)$ where ks is the active state configuration, EP is the event pool and GV is the set of global shared variables. Configurations describe the stable status of a UML state machine.*

A configuration can be considered as a (stable) snapshot of the current UML state machine. The execution of an RTC step can be depicted as moving from one configuration to the next configuration. Based on the above definition, we provide the following (inference) rules to formalize the procedure of an RTC step.

Wandering Rule. This rule captures the case where a dispatched event e is neither consumed nor delayed. As a result, it is discarded, i.e., removed from the event pool without causing any other effect.

$$\frac{\begin{array}{l} e = !EP, EP' = EP \backslash \{e\}, \\ \forall\, s \in ks, e \notin s.\widehat{t_{def}}, Enable((ks, EP', V), e) = \varnothing \end{array}}{(ks, EP, V) \xrightarrow{e} (ks, EP', V)} [\ Wandering\ ]$$

Event e is dispatched from event pool ($!EP$), but no transition is triggered by e (i.e., $Enable((ks, EP', V), e) = \varnothing$), and no deferred event in the current configuration matches the event e (i.e., $\forall\, s \in ks, e \notin s.\widehat{t_{def}}$). Event pool EP' is the the event pool EP after dispatching event e. After executing this RTC step, only the event pool of the state machine configuration changes.

Deferral Rule 1. This rule captures the case where a dispatched event is deferred by some states in the current active state configuration, but does not trigger any transitions.

$$\frac{\begin{array}{l} e = !EP, EP' = EP \backslash \{e\}, \\ \exists\, s \in ks : e \in s.\widehat{t_{def}}, \mathit{Enable}((ks, EP', V), e) = \varnothing, \\ EP'' = \mathit{Merge}(e, EP'.DEP), \end{array}}{(ks, EP, V) \xrightarrow{e} (ks, EP'', V)} \; [\; \mathit{Deferral1} \;]$$

Since event e is deferred, it should not be discarded but merged back to the deferred event pool ($\mathit{Merge}(e, EP'.DEP)$). So after the RTC execution, only the event pool EP'' is changed.

Deferral Rule2. This rule captures the case where the dispatched event e triggers some transitions and it is also deferred by some states in the current active state configuration. But there exists at least one state, which defines the deferral event, that has higher priority than the source states of the enabled transitions.

$$\frac{\begin{array}{l} e = !EP, EP' = EP \backslash \{e\}, \\ \exists\, s \in ks : e \in s.\widehat{t_{def}}, \widehat{T} = \mathit{Enable}((ks, EP', V), e), \widehat{T} \neq \varnothing, \\ \forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (ks, EP', V), e) \\ EP'' = \mathit{Merge}(e, EP'.DEP) \end{array}}{(ks, EP, V) \xrightarrow{e} (ks, EP'', V)} \; [\; \mathit{Deferral2} \;]$$

$\widehat{T}$ is the set of transitions enabled by the dispatched event e. Event e is also deferred by some states in the current active state configuration and the event deferral has higher priority over transition firing ($\forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (ks, EP', V), e)$. Function $\mathit{deferralConflict}$ is used to solve deferral conflicts and is formally defined in [10]. As a consequence, only the event pool of the state machine changed.

To increase the readability of the rules, we use the following brief representations in all the following RTC rules. $\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_n) = \mathit{CollectAct}(\tilde{t}_1);\ , \ldots, ;\ \mathit{CollectAct}(\tilde{t}_n)$ represents the execution of the behaviors along $\tilde{t}_1, \ldots, \tilde{t}_n$ (i.e., a list of compound transitions). $\mathit{Merge}(\mathcal{A}(\langle \tilde{t} \rangle), EP)$ represents merging the event generated by actions in $\mathcal{A}(\langle \tilde{t} \rangle)$ if any into event pool EP. $\mathit{UpdateV}(\mathcal{A}(\langle \tilde{t} \rangle), GV)$ represents updating of global shared variables GV by actions in $\mathcal{A}(\langle \tilde{t} \rangle)$.

Progress Rule. This rule captures the case where a set of compound transitions are triggered by a dispatched event e. There is no event deferral or the fired transitions have higher priority over event deferral.

$$\frac{\begin{array}{l} e = !EP, EP' = EP \backslash \{e\}, \\ \widehat{T} = \mathit{Firable}((ks, EP', V), e), \mid \widehat{T} \mid = n, \langle \tilde{t} \rangle \in \mathit{Permutation}(\widehat{T}), \\ EP'' = \mathit{MergeA}(\mathcal{A}(\langle \tilde{t} \rangle), EP'), V' = \mathit{UpdateV}(\mathcal{A}(\langle \tilde{t} \rangle), V) \end{array}}{(ks, EP, V) \xrightarrow{e} (\mathit{NextK}(ks, \langle \tilde{t} \rangle), EP'', V')} \; [\; \mathit{Progress} \;]$$

Function $\mathit{Firable}((ks, EP', V), e)$ (defined in [10]) returns a set of compound transitions which is the maximal non-conflicting subset of enabled transitions. As a result, the firable set of transitions will be executed in an order specified by $\langle \tilde{t} \rangle$, which is an ordered list of compound transitions. Function $Permutation$ (defined in [10]) computes all possible total orders on the set of compound transitions $\widehat{T}$. This function captures the orthogonal composite state level non-determinism, i.e., when multiple compound transitions are fired. Behaviors are collected along the transition execution sequence following the permutation order (indicated by $\mathcal{A}(\langle \tilde{t} \rangle)$). Active state configuration is changed as computed by function $NextK(ks, \langle \tilde{t} \rangle)$.

ProgressC Rule. This rule captures the case where choice pseudostates are encountered during an RTC execution. Different from the RTC Progress rule, dynamic evaluation would be conducted at the point where a choice pseudostate is reached.

$$\frac{\begin{array}{l} e =!EP, EP' = EP \backslash \{e\}, \\ \widehat{T} =\in Firable((ks, EP', V), e), \mid \widehat{T} \mid= n, \\ \tilde{t}_i^1 \in \widehat{T}, \langle \tilde{t} \rangle = (\tilde{t}_1, \ldots \tilde{t}_i^1, \ldots, \tilde{t}_n) \in Permutation(\widehat{T}) \\ V' = UpdateV(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), V), \\ EP'' = MergeA(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), EP') \\ \tilde{t}_i^2 \in Firable((\{last(\tilde{t}_i^1).tv\}, EP'', V'), e), \\ EP''' = MergeA(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), EP''), \\ V'' = UpdateV(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), V') \end{array}}{(ks, EP, V) \xrightarrow{e} (NextK(ks, \langle \tilde{t} \rangle), EP''', V'')} \; [\; ProgressC \;]$$

The RTC ProgressC rule captures the same situation as the RTC Progress rule except that choice pseudostates are encountered in a compound transition. Compound transition t_i is splitted by a choice pseudostate into t_i^1 and t_i^2. The second half of t_i is evaluated based on the current environment V'.

4.4 System Semantics

A UML state machine models the dynamic behavior of one object within a system. But multiple state machines representing different components of a system may interact with each other synchronously or asynchronously. The interactions between state machines together with the dynamic behavior of each single state machine compose the dynamic behavior of the whole system. In order to verify the correctness of the overall system behaviors, we need to capture the message passing sequences between all state machines in the system.

Definition 11 (Semantics of a system). *The* semantics of a system *is defined as a Labeled Transition System (LTS) $\mathcal{L} \triangleq (\mathbb{S}, \mathcal{S}_{init}, \rightsquigarrow)$, with:*

- $\mathbb{S}$ *is the set of states of $\mathcal{L}$. Each LTS state is a tuple $(k_1, \ldots, k_n)$ where k_i is the configuration of the state machine Sm_i within the system;*
- $\mathcal{S}_{init}$ *is the initial state of $\mathcal{L}$.*
- $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$ *is the transition relation of $\mathcal{L}$;*

The LTS transition relations are defined as follows.

$$\frac{|||_{i \in [1,n]}^{C} Sm_i, k_j \to k_j'}{(k_1, \ldots, k_j, \ldots, k_n) \rightsquigarrow (k_1, \ldots, k_j', \ldots, k_n)} \; [\; LTS1 \;]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \to k'_j, e = \mathit{SendSignal}(j,k), \mathit{Merge}(e, EP_k)}{(k_1,\ldots,k_k,\ldots,k_j,\ldots,k_n,) \rightsquigarrow (k_1,,\ldots,k'_k,\ldots,k'_j,\ldots,k_n)} \; [\; LTS2 \;]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \to k'_j, e = \mathit{Call}(j,k), e \in C, k_k \xrightarrow{e} k'_k}{(k_1,\ldots,k_k,\ldots,k_j,\ldots,k_n) \rightsquigarrow (k_1,\ldots,k'_k,\ldots,k'_j,\ldots,k_n)} \; [\; LTS3 \;]$$

All the state machines in the system are executed non-deterministically. If the event pool of one state machine dispatches an event, all the effects caused by the dispatched event must be fulfilled before the RTC step completes. Specially, if a call action is invoked by the effects of the current RTC step, the RTC does not complete until the call action returns. Rule LTS1 captures the normal situation that a single state machine is executed without communicating with other state machines. Rule LTS2 captures the case where asynchronous communication is involved, i.e., the executing state machine sends an asynchronous message to another state machine. The state machine receiving the message merges the message into its own event pool. Rule LTS3 captures the case where synchronous communication is involved. In this case, the callee state machine is triggered by the call event. As a consequence, more than two state machines are triggered to execute. The caller state machine can not finish its RTC step until the callee has finished execution. Function $\mathit{SendSignal}(j,k)$ and $\mathit{Call}(j,k)$ represent the jth state machine sends an asynchronous and a synchronous message to the kth state machine, respectively.

5 Implementation and Evaluation

We have implemented the formal semantics defined in Section 4 in a self-contained tool USM^{2}C. This tool supports model checking of deadlock-freeness and LTL properties, as well as step-wise simulation of state machine executions. Counterexamples are reported in terms of state machine execution traces. Due to space limitation, we report part of the experiment here. The full set of experiments can be found in [10].

The first experiment is a comparison on the *BankATM*[3] state machine provided in [8] with the off-the-shelf tool HUGO [8][4]. The *BankATM* system contains Bank state machine and ATM state machine, which communicate with each other via both synchronous and asynchronous events. HUGO translates UML state machine models into Promela and uses Spin as the underlying model checker to do the verification. Due to its comparability problem with Spin, we manually inspect the Promela code generated by HUGO, write LTL properties accordingly and invoke Spin. The property we checked is $\Box(\mathit{retain} \to ((!\mathit{cardValid} \wedge \mathit{numIncorrect} \geq \mathit{maxNumIncorrect}))$. It guarantees that when a card is retained, it must be the case that at least $\mathit{maxNumIncorrect}$ times of wrong pin are entered. This property should hold for the *BankATM* system. Both

[3]We did modifications on the *BankATM* system to comply with UML 2.4.1 state machine specifications. The modified *BankATM* system is available in [10].

[4]This is the only tool that model checks UML state machines available to public downloading we are aware of. The latest version of HUGO is based on Spin4.3.0, which is currently unavailable, and HUGO has compatibility problems with Spin5.x and Spin6.x.

Table 2. Scalability Evaluation Result

N	Time (s)	States	Transitions	Memory (KiB)
2	0.06	63	105	$8,701$
3	0.11	598	$1,397$	$10,970$
4	1.1	$5,560$	$17,448$	$26,726$
5	13.1	$50,737$	$199,513$	$163,947$
6	163	$447,895$	$2,237,563$	$734,510$

Spin and USM^{2}C report a valid verification result. Spin reported 34.3 MiB memory usage, 61 stored states and 106 transitions verifying the above property on the generated Promela code. Our tool USM^{2}C reported 9.8 MiB total memory usage, 28 states and 31 transitions visited. By manually inspecting the Promela code generated by HUGO, we found that an RTC step semantics is implemented as multiple steps in the presence of orthogonal regions in their translation. This may lead to redundant copies of variables and propositions, which cause more memory usage.

The second experiment is on the example in Fig. 1, which modifies the example provided in [4][5] by manually introducing bugs. The example contains transitions which emanate and enter orthogonal composite states, such as the transition from *Cruising* state to *WaitArrivalOK* state, which is not supported by HUGO.

We checked the LTL property $\Box(alert100 \rightarrow \Diamond arriveAck)$, which depicts the situation when a car approaches a terminal and is 100 yards from it; the car will finally receive the *arriveAck* event from the *Handler*. This property guarantees that the car will not wait on the rail forever and it should hold globally in the *RailCar* system. But it is reported to be violated and our tool finds the loop (*opend*→*alert100*)*, indicating that the event *opend* caused the problem. The reason is that the *opend* event is immediately available on entering state *WaitArrivalOK*; thus it got a chance to be dispatched by the event pool in the next RTC step and causes the problem.

The third experiment is to evaluate the scalability of our tool. We modeled the dinning philosopher problem with UML state machines and conducted model checking with our tool. Table 2 shows the result of this experiment.

The data listed in Table 2 is the result of checking deadlock free property with our *Shortest Witness Trace using Breadth First Search* search engine, which forces breadth first search. The state space we get is quite close to the real state space generated. We can see from the result that our tool can handle large state spaces caused by non-determinism. In addition, we can further reduce the state space through techniques like partial order reduction. We are considering this as one of our future work.

We believe that communications between objects are error-prone and hard to find manually. The experiment results show that our method is effective in finding design errors in the presence of both synchronous and asynchronous communications. Our tool is also more efficient and can deal with more features of UML state machines.

6 Conclusion

In this paper, we provided a formal semantics for the complete set of UML behavioral state machine features. Our semantics considers state machine level and orthogonal

[5] We remove the *Arrival* state and the completion transition emanating from *Operating* state

composite state level non-determinisms as well as the communication aspect between UML state machines which bridge the gap of current approaches. To the best of our knowledge, this is the first attempt of full formalization of the latest UML state machines specification [1]. We have implemented a self-contained tool for model checking various properties for UML behavior state machine. The experiments show that our tool is effective in finding bugs with both synchronous and asynchronous communications between different state machines. Several issues linked with UML state machines remain unaddressed. In future work, we aim at considering the real-time aspects and object-oriented issues, such as dynamic invoking and destroying objects.

References

1. OMG unified language superstructure specification(formal). version 2.4.1, 2011-08-06. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`.
2. É. André, C. Choppy, and K. Klai. Formalizing non-concurrent UML state machines using colored Petri nets. *ACM SIGSOFT Software Engineering Notes*, 37(4):1–8, 2012.
3. 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.
4. D.Harel and E.Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:31–42, 1997.
5. 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.
6. H. Fecher, J. Schönborn, M. Kyas, and W. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. *Formal Methods and Software Engineering*, pages 52–65, 2005.
7. Y. Jin, R. Esser, and J. Janneck. A method for describing the syntax and semantics of UML statecharts. *Software and Systems Modeling*, 3(2):150–163, 2004.
8. A. Knapp and S. Merz. Model checking and code generation for UML state machines and collaborations. In *Proc. 5th W. Tools System Design & Verif.* , volume 11, pages 59–64, 2002.
9. A. Knapp, S. Merz, and C. Rauh. Model checking - timed UML state machines and collaborations. In *FTRTFT'02* , pages 395–416. Springer-Verlag, 2002.
10. S. Liu, Y. Liu, É. André, C. Choppy, J. Sun, B. Wadhda, and J. S. Dong. A Formal Semantics for the Complete Syntax of UML State Machines with Communications (Report). Technical report, National University of Singapore, 2013. `http://comp.nus.edu.sg/~lius87/uml/techreport/uml_sm_semantics.pdf`.
11. J. Schönborn. Formal semantics of UML 2.0 behavioral state machines. Technical report, Inst. Computer Science and Applied Mathematics, Christian-Albrechts-Univ. of Kiel , 2005.
12. M. Von Der Beeck. A structured operational semantics for UML-statecharts. *Software and Systems Modeling*, 1(2):130–141, 2002.
13. S. Zhang and Y. Liu. An automatic approach to model checking UML state machines. In *4th Int. Conf. Secure Software Integration & Reliability etc. (SSIRI-C)* , pages 1–6. IEEE, 2010.