6-2013

A formal semantics for complete UML state machines with communications

Shuang LIU

Yang LIU

Étienne ANDRÉ

Christine CHOPPY

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Citation

LIU, Shuang; LIU, Yang; ANDRÉ, Étienne; CHOPPY, Christine; SUN, Jun; WADHWA, Bimlesh; and DONG, Jin Song. A formal semantics for complete UML state machines with communications. (2013). *Proceedings of the 10th International Conference, IFM 2013 Turku, Finland, June 10-14*. 331-346.
Available at: https://ink.library.smu.edu.sg/sis_research/5003

A Formal Semantics for Complete 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] Université Paris 13, Sorbonne Paris Cité, LIPN, F-93430, Villetaneuse, France
[4] Singapore University of Design and Technology, Singapore

Abstract. UML is a widely used notation, and formalizing its semantics is an important issue. Here, we concentrate on formalizing UML state machines, used to express the dynamic behaviour of software systems. We propose a formal operational semantics covering all features of the latest version (2.4.1) of UML state machines specification. We use labelled transition systems as the semantic model, so as to use 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 state machines are widely used to model the dynamic behaviour of an object. Since the UML specification is documented in natural language, inconsistencies and ambiguities arise, and it is thus important to provide a formal semantics for UML state machines. A formal 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 analysis of UML state machines.

However, existing works only provide formal semantics for a subset of UML state machines features, leaving some important issues unaddressed. A few approaches [19,22] consider the non-determinism in the presence of orthogonal composite states, which is an important modelling concept. Although extensibility of the syntax structure is important due to the refinement operations on UML state machines, the syntax formats defined in those works does not extend well. A semantics able to support the full set of syntax features will help to bring the expressive power of UML state machines to life.

Secondly, in the existing approaches, the event pool mechanism and the communications between state machines are not thoroughly addressed. UML state machines are used to model the behaviour of objects. The whole system may include several state

* This work is supported by project 9.10.11 "Software Verification from Design to Implementation" of Programme Merlion (official collaborative grant co-funded by France and Singapore).

E.B. Johnsen and L. Petre (Eds.): IFM 2013, LNCS 7940, pp. 331–346, 2013.

machines interacting with each other synchronously or asynchronously. Enabling the verification of the entire system is quite important, especially in the presence of synchronous communications, which are more likely to cause deadlock situations.

Lastly, the unclarities (that is, inconsistencies and ambiguities) in the UML state machines specifications are not thoroughly checked and discussed. Fecher et al. [8] discussed 29 unclarities in UML 2.0 state machines. But there are still some unclarities (such as the granularity of a transition execution sequence) that are not covered in [8] but will be discussed in this work.

This work aims at bridging the gaps in the existing approaches with the following contributions. (1) We provide a formal operational semantics for UML 2.4.1 state machines covering the complete set of UML state machines features. In particular, our syntax structure is extensible to state machine refinement and future changes. Our semantics formalization considers non-determinism as well as synchronous and asynchronous communications between state machines. (2) We explicitly discuss the event pool mechanisms and consider deferral events as well as completion events. (3) We report new unclarities in UML 2.4.1 state machines specifications. (4) We develop a self-contained tool USM^{2}C based on the semantics we have defined; it model checks various properties such as deadlock-freeness and linear temporal logic (LTL) properties. We conduct experiments on our tool and results show its effectiveness.

The rest of this paper is organized as follows. Section 2 provides the preliminaries of UML state machines. Section 3 and Section 4 define the syntax and semantics for UML state machines, respectively. Section 5 provides the implementation and evaluation results. Related work is discussed in Section 6. Section 7 addresses the limitations of our work, and concludes the paper with future works.

2 UML State Machines Features and Our Assumptions

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 in Fig. 1 (a modified version of the example used in [10]) as a running example. The *RailCar* system is composed of 3 state machines: *Car*, *Handler* and *DepartureSM* (referenced by the *Departure* submachine state in the *Car* state machine). They communicate with each other through synchronous event calls.

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. The container of a transition is the region which owns the transition. A compound transition is composed of multiple transitions joined via choice, junction, fork and join pseudostates.

Regions. It is a container of vertices and transitions, and represents the orthogonal parts of a composite state or a state machine. In Fig. 1, the area *[R1]* is a region.

States. There are three kinds of states, viz., simple state (*Idle*), composite state (*Operating*) and submachine state (*Departure*). An orthogonal composite state (*WaitArrivalOK*) has more than one region. States can have optional entry/exit/do behaviours. A do behaviour (*PlaySound* in state *Alerted*) can be interrupted by an event. A state can also

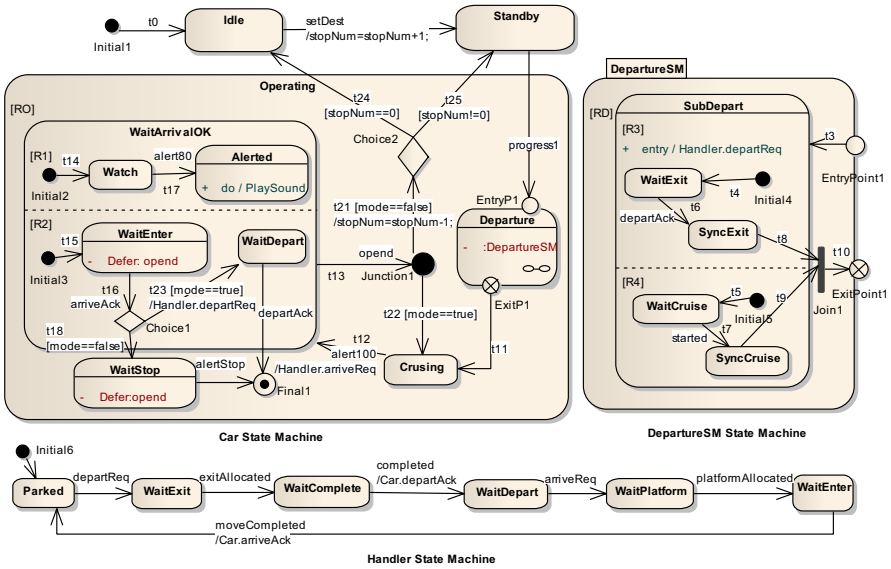

Fig. 1. The RailCar state machine

define a set of deferred events ($\{opend\}$ in state *WaitEnter*). A final state (*Final1*) is a special kind of state which indicates finishing of its enclosing region.

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. A join pseudostate (*join1*) is used to merge transitions from states in orthogonal regions. A fork pseudostate is used to split transitions targeting states in orthogonal regions. Junction pseudostates (*Junction1*) represent static branching points. Choice pseudostates (*Choice1*) represent dynamic branching points, i.e., the evaluation of enabled transitions is based on the environment when the choice pseudostate is reached.

Connection Point Reference. It is an entry/exit point of a submachine state and refers to the entry/exit pseudostate of the state machine that the submachine state refers to. In Fig. 1, *EntryP1* and *ExitP1* in *Departure* state are connection point references.

Active State Configuration. It is a set of active states of a state machine when it is in a stable status[1]. In Fig. 1, $\{Operating, Crusing\}$ is an active state configurations.

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 event), which may cause the state machine to move to the next active state configuration, accompanied by behaviour executions. It is the basic semantic step in UML state machines. For example in Fig. 1, $\{Operating, WaitArrivalOK, Watch, WaitDepart,\} \xrightarrow{opend} \{Idle\}$ is an RTC step.

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

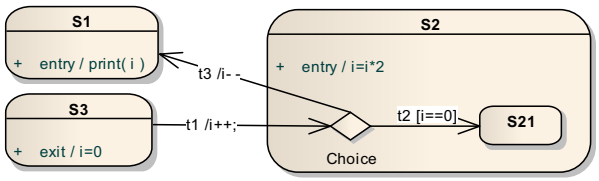

Fig. 2. Illustration of transition execution sequence

2.2 Basic Assumptions on UML State Machines Semantics

We briefly sketch below some new unclarities (detailed in [17]) we found in the UML 2.4.1 state machines specification, as well as our assumptions in this work.

Transition Execution Sequence. Transitions and compound transitions are used in interleaving in the descriptions of transition execution sequence, which raises confusions. The transition execution ordering is important since different execution orders may lead to different results. For example in Fig. 2, Suppose S_3 is active and transition t_1 is fired. If we define the transition execution sequence based on the compound transition, the behaviour execution sequence is "$i = 0;\ i++;\ i--;\ print(i)$ " and 0 should be printed. If we define the transition execution sequence based on a single transition, the behaviour execution sequence should be "$i = 0;\ i++;\ i = i * 2;\ i--;\ print(i)$" and 1 should be printed. In the first case, the entry behaviour of state *S2* is not executed, which contradicts the semantics of entry behaviours. We define the transition execution sequence based on a transition to keep the semantics consistent with entry behaviours.

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. The execution order of the (behaviours associated with the) fired transitions may affect the value of global shared variables. 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.

3 Syntax of UML State Machines

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

Our syntax definition preserves the structure specified by [1], which makes it suitable to support refinement as well as future changes of UML state machines.

Table 1. Type notations

Symbol	Type	Symbol	Type	Symbol	Pseudostate type
$\mathcal{K}_{\mathcal{S}}$	active state configuration	$\mathbb{B}$	boolean	DH_{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
CR	connection point reference	E	event	En_{ps}	entry point
SM	state machine	R	region	F_{ps}	fork
B	behaviours	PS	pseudostate	SH_{ps}	shallow history
$\langle B \rangle$	behaviour list	$\mathbb{N}$	natural number	Ex_{ps}	exit point

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

- $\widehat{r} \subset R$ *is the set of regions directly contained in this state,*
- $\widehat{t_{def}} \subset Trig$, $\alpha_{en} \in B$, $\alpha_{ex} \in B$ *and* $\alpha_{do} \in B$ *are the set of deferred events, the entry, exit and do behaviours defined in the state, respectively.*
- $\widehat{en} \subset En_{ps}$ *and* $\widehat{ex} \subset Ex_{ps}$ *are the set of entry point and exit point pseudostates associated with the state.*
- $\widehat{cr} \subset CR$ *is the set of connection point references belonging to the state.* $sm \in SM$ *is the state machine referenced by this state; the two fields are 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 states, viz., simple state (S_s), composite state (S_c), orthogonal composite state (S_o) and submachine state (S_m). In Fig. 1, the submachine state *Departure* is denoted as $(\varnothing, \varnothing, \epsilon, \epsilon, \varnothing, \varnothing, \{\textit{EntryP1, ExitP1}\}, \textit{DepartureSM}, \varnothing)$, where ϵ and $\varnothing$ denote the empty element and the empty set, respectively.

Definition 2 (Pseudostate). *A* pseudostate *is a tuple* $ps = (\iota, \widehat{h})$*, where* $\iota \in R \cup SM$ *is the region or state machine 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.*

The last column of Table 1 shows the notations of the ten kinds of pseudostates PS.

Definition 3 (Final state). *A* final state *is a special kind of state, which is defined as a tuple* $s_f = (\iota)$ *where* $\iota \in S_o \cup S_c \cup SM$ *is the composite state or state machine 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 pseudostates corresponding to this connection point reference, and* s *is the submachine state in which the connection point reference is defined.*

For example, in Fig. 1, *EntryP1* is defined as $(\{\textit{EntryPoint1}\}, \varnothing, \textit{DepartureSM})$.
Vertex $V \triangleq S \cup S_f \cup PS \cup CR$ is an abstraction of all nodes.

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.*
- $\widehat{tg} \subset Trig$, $g \in C$, $\alpha \in B$ *and* $\iota \in R$ *are the set of triggers, the guard, the associated behaviour and the container of the transition, respectively.*
- $\widehat{tc}$ *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*[2] *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,* $\alpha_{st} \in B$ *is the behaviour associated with the segment transition.* $\iota_{st} \in R$ *is the container of the segment transition.*

We define the following functions on transitions for clarity sake. Functions $isFork(t)$ and $isJoin(t)$ decide whether transition t is a fork transition and join transition, respectively. For example, in Fig. 1, the join transition *t10* is $(\{$*Join1*$\}, \{$*ExitPoint1*$\}, \varnothing, \epsilon, \epsilon,$ *RD*, $\{($*SyncExit*$, \epsilon,$ *RD*$), ($*SyncCruise*$, \epsilon,$ *RD*$)\})$. We use $t.\widetilde{\alpha}$ to represent all possible action execution sequences of t. Formal definition of $t.\widetilde{\alpha}$ is in [17].

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

Definition 7 (State Machine). *A* state machine *is defined as* $sm \triangleq (\widehat{r},\ \widehat{cp})$, *where* $\widehat{r} \subset R$, $\widehat{cp} \subset En_{ps} \cup Ex_{ps}$ *are the set of (directly owned) regions and the set of entry/exit point pseudostates defined for this state machine.*

For example in Fig. 1, state machine *DepartureSM* is $(\{$*RD*$\}, \{$*EntryPoint1*$, $*ExitPoint1*$\})$.

Definition 8 (Compound Transition). *A* compound transition *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* $st \in ST \equiv (len(st) = 1 \wedge seg(st, 0) \in T) \vee \exists\, st_i, st_j \in ST : last(st_i) = first(st_j) \wedge st = st_i \frown st_j$.

The operator $\frown$ denotes the operation of connecting transitions in order. Notation $len(\tilde{t})$ denotes the total number of segment transitions the compound transition is composed of. $seg(\tilde{t}, i)$ denotes 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 denote the first and last segment of $\tilde{t}$. We define $\tilde{t}.\widehat{sv} = first(\tilde{t}).\widehat{sv}$, $\tilde{t}.\widehat{tv} = last(\tilde{t}).\widehat{tv}$ for convenience sake.

Compositional Operators. The operator "; " represents a sequential composition. Interleave operation ($|||$) represents a non-determinism in the execution orders. Interleave with synchronous communications ($|||^C$) is a special case of interleaving: it requires the state machines to synchronize on the specified event in 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.

Definition 9 (System). *A* system *is a set of state machines executing in interleaving (with synchronous communications).* $sys \triangleq |||^C_{i\in[1,n]} Sm_i$ *where* $Sm \triangleq (sm, P, GV)$. *In* Sm, sm *denotes the state machine,* P *the event pool associated with* sm, *and* GV *the shared variables of* sm. *And* n *is the number of state machines within the system* sys.

[2] We treat exit (resp. entry) point pseudostate the same way with join (resp. fork) pseudostate.

For example, the *RailCar* system in Fig. 1 is defined by $|||^C(\mathit{Car}, \mathit{Handler})$, where $C = \{\mathit{departReq}, \mathit{departAck}, \mathit{arriveReq}, \mathit{arriveAck}\}$.[3]

Event Pool Abstraction. Change events, signal events, and deferred events are processed differently in UML state machines. We provide for this purpose 3 separate event pools, viz., completion event pool (CP), deferred event pool (DP), and normal event pool (NP). $P \triangleq (CP, DP, NP)$ represents the event pool, and we define two basic operations on P. $Merge(e, EP)$ merges an event e into the corresponding event pool represented by EP, and $Disp(P)$ dispatches an event from P. Since function $Merge$ (formally defined in [17]) is straightforward, we focus here on Function $Disp$.

Definition 10. *The following function formally defines the event dispatch mechanism.*

$$Disp(P, ks) \triangleq \begin{cases} CP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP \neq \varnothing \wedge HighestPriority(e, CP) \\ DP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP = \varnothing \wedge DP \neq \varnothing \wedge !isDeferred(e, ks) \\ NP\backslash\{e\};\ CheckDP(P, ks) & \text{if } CP = \varnothing \wedge allDefer(DP, ks) \wedge NP \neq \varnothing \\ \epsilon & \text{otherwise} \end{cases}$$

$$CheckDP(P, ks) \triangleq DP\backslash E;\ NP \cup E,\ \textit{where } E \triangleq \{e \mid e \in DP \wedge !isDeferred(e, ks)\}.$$

The function guarantees that the precedence order $CP \prec DP \prec NP$ is preserved ($\prec$ denotes the preceding partial order). But the order within each event pool is not specified. The macro $HighestPriority(e, CP)$ denotes that event e has the highest priority in CP, which preserves the priority order of a nested state over its ancestor states. In the deferred event pool, only events that are not deferred in the current active state configuration ($!isDeferred(e, ks)$) can be dispatched. The macro $allDefer(DP, ks) \Leftrightarrow \forall\, e \in DP, isDeferred(e, ks)$ guarantees the priority of deferred events over normal events. When an event is dispatched, we check all the deferred events defined in the states of the current active state configuration, and remove those events that are not deferred any more from DP to NP; this is accomplished by $CheckDP$.

4 A Formal Semantics for UML State Machines

This section devotes to a self-contained formal semantics for all UML state machines features. We have adopted the semantic model of Labelled Transition Systems (LTS). The dynamic semantics of a state machine is captured by the execution of RTC steps, which have two kinds of effects, viz., changing active states and executing behaviours. We formally define the two kinds of effects separately. Then the semantics of the RTC step is defined formally. At last, we define the semantics of the system.

4.1 Active State Configuration Changes

An active state configuration $\mathcal{K}_\mathcal{S}$ is a set of states which are active at the same time. It describes a stable state status when the previous RTC step finishes. We use Active

[3] We treat the state machine (*DepartureSM*) that is referenced by a submachine state (*Departure*) the same way as a composite state.

Vertex Configuration $\mathcal{K}_{\mathcal{V}}$ (a set of vertices that are active at the same time) to represent the snapshot of a state machine during an RTC execution. For example, in Fig. 1, $\{$*Operating, Choice2*$\}$ is an active vertex configuration. $\mathcal{K}_{\mathcal{S}}$ and $\mathcal{K}_{\mathcal{V}}$ are defined in [17].

Next Active State Configuration. $NextK : \mathcal{K}_{\mathcal{S}} \times \langle \widetilde{T} \rangle \rightarrow \mathcal{K}_{\mathcal{S}}$ computes the next active state configuration after executing the compound transition list indicated by $\langle \widetilde{T} \rangle$. 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 $NxK : \mathcal{K}_{\mathcal{S}} \times \widetilde{T} \rightarrow \mathcal{K}_{\mathcal{S}}$ computes the next active state configuration after executing a compound transition indicated by $\widetilde{T}$. Formally: $NxK(ks, \tilde{t}) \triangleq NxPK(kv_n, seg(\tilde{t}, n))$, 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 $NxPK : \mathcal{K}_{\mathcal{V}} \times T \rightarrow \mathcal{K}_{\mathcal{V}}$ 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 defined in [17].

4.2 Behaviour Execution

Another effect of executing an RTC step is to cause behaviours to be executed. We define the following functions to collect the behaviour execution sequence.

Exit Behaviour. $ExitBehaviour : \mathcal{K}_{\mathcal{V}} \times T \rightarrow \langle B \rangle$ collects the ordered exit behaviours of states that a given transition leaves in the current vertex configuration. Formally:

$$ExitBehaviour(kv, t) = ExV(kv, MainSource(t), t)$$

$$ExR(kv, r, t) \triangleq \begin{cases} SH(h, v);\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge \\ & \quad v \in S \wedge \exists\, h \in SH_{ps} : h \in r.\widehat{v} \\ DH(h, v);\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \wedge v \in S \\ & \quad \wedge \exists\, h \in DH_{ps} : isAncestor(h.\iota, r) \\ & \quad \wedge\ isAncestor(t.\iota, h.\iota) \\ ExV(kv, v, t) & \text{if } r \in R \wedge \exists\, v \in r.\widehat{v} : v \in kv \\ & \quad \wedge \forall\, s' \in r.\widehat{v}, s' \notin SH_{ps} \\ & \quad \wedge\ \nexists h \in DH_{ps} : isAncestor(h.\iota, r) \\ & \quad \wedge\ isAncestor(t.\iota, h.\iota) \end{cases}$$

$$ExV(kv, v, t) \triangleq \begin{cases} |||^{C}_{r \in v.\widehat{r}} ExR(kv, r, t);\ exit(v) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ ExR(kv, r, t);\ exit(v) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ exit(v) & \text{if } v \in S_s \\ ExV(kv, cr, t) & \text{if } v \in Ex_{ps} \wedge \\ & \quad \exists\, cr \in CR : v \in cr.\widehat{ex} \\ ExV(kv, v.s, t) & \text{if } v \in CR \\ Agn(v.\widehat{r}, v.sm.\widehat{r});\ ExV(kv, v, t) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ \epsilon & \text{otherwise} \end{cases}$$

The exit behaviours of executing a transition are collected recursively starting from the innermost state. We define functions ExV and ExR to recursively collect exit behaviours. All the regions of a composite state should be exited before it. If the region contains a (shallow/deep) history pseudostate, the content of the history pseudostate should be set properly (by functions SH and DH respectively) before exiting the region. Exiting simple states means terminating the do behaviour (if any) and executing

the exit behaviour, as defined by $exit(v) = v.\alpha_{do} \nabla v.\alpha_{ex}$. If an exit point pseudostate is encountered, the associated connection point reference is exited, which means the state defining the connection point reference is exited. Exiting a submachine state means exiting all the regions in the state machine it refers to. Function $Agn(v.\widehat{r}, v.sm.\widehat{r})$ assigns the set of regions of a state machine to the the of regions of a submachine state.

Entry Behaviour. $EntryBehaviour : T \rightarrow \langle B \rangle$ collects the ordered entry behaviours of the states a given transition enters. Formally:
$EntryBehaviour(t) = EnV(MainTarget(t), Enter(t))$
$EnR(r, \widehat{V}) \triangleq EnV(s', \widehat{V})$ where $r \in R \wedge s' \in r.\widehat{v} \wedge s' \in \widehat{V}$

$$EnV(v, \widehat{V}) \triangleq \begin{cases} v.\alpha_{en};\ (|||^{C}_{r \in v.\widehat{r}} EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_o \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ (EnR(r, \widehat{V}) \parallel v.\alpha_{do}) & \text{if } v \in S_c \vee (v \in S_m \wedge v.\widehat{r} \neq \varnothing) \\ v.\alpha_{en};\ v.\alpha_{do} & \text{if } v \in S_s \\ GenEvent(v.\iota) & \text{if } v \in S_f \wedge \forall\, r \in v.\iota.\widehat{r}, \\ & \exists\, s' \in r.\widehat{v} : s' \in kv \Rightarrow s' \in S_f \\ Agn(v.\widehat{r}, v.sm.\widehat{r});\ EnV(v, \widehat{V}) & \text{if } v \in S_m \wedge v.\widehat{r} = \varnothing \\ EnV(v.s, \widehat{V}) & \text{if } v \in CR \\ EnV(cr, \widehat{V}) & \text{if } v \in En_{ps} \wedge \exists\, cr \in CR : v \in cr.\widehat{en} \\ \epsilon & \text{otherwise} \end{cases}$$

Entry behaviours are collected in a similar manner to exit behaviours, except that the collect starts from the outermost state. We define functions EnV and EnR to recursively collect the entry behaviours of all the vertices in $\widehat{V}$ in order. States entered by firing transition t are computed by function $Enter(t)$. Starting from the main target state of a transition, all regions of a composite state are entered in interleaving. Entering each state means executing its entry behaviour followed by its do activities ($s.\alpha_{en};\ s.\alpha_{do}$). Do activities of a composite state should be executed in parallel ($\parallel$) with all the behaviours of its containing states. Function $GenEvent(s)$ generates a completion event for state $s.\iota$ and merges the generated event in the completion event pool (CP).

Collect Actions. $CollectAct : \mathcal{K}_{\mathcal{S}} \times \widetilde{T} \rightarrow \langle B \rangle$ collects the ordered sequence of behaviours associated with the execution of the given compound transition. Formally: $CollectAct(ks, \tilde{t}) \triangleq Act(kv_1, seg(\tilde{t}, 1));\ \ldots;\ Act(kv_i, seg(\tilde{t}, i));\ \ldots;\ Act(kv_n, seg(\tilde{t}, n))$, $andAct(kv, t) \triangleq ExitBehaviour(kv, t);\ t.\tilde{\alpha};\ EntryBehaviour(t)$ where $n = len(\tilde{t})$, $kv_1 = ks$ and $kv_i = NxPK(kv_{i-1}, seg(\tilde{t}, i-1))$ 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 behaviour executions which may cause the event pool and global shared variables to be updated. We use the term *configuration* to capture the stable status of a state machine.

Definition 11. *A* configuration *is a tuple $k = (ks, P, GV)$ where ks is the active state configuration, P is the event pool and GV is the set of valuation of global variables.*

For example, $(\{Idle\}, (\varnothing, \varnothing, \{setDest\}), \{stopNum = 0, mode = false\})$ is a configuration. The execution of an RTC step can be depicted as moving from one configuration

to the next configuration. We provide the following rules to formalize an RTC step. We use the *RailCar* system in Fig. 1 to illustrate the following RTC step rules.

Wandering Rule. This rule captures the case where a dispatched event e is neither consumed nor delayed. As a result, it is discarded.

$$\frac{e = \mathit{Disp}(P), P' = P \backslash \{e\}, \forall\, s \in ks, e \notin s.\widehat{t_{def}}, \mathit{Enable}((ks, P', GV), e) = \varnothing}{(ks, P, GV) \xrightarrow{e} (ks, P', GV)}$$

Event e is dispatched from event pool ($\mathit{Disp}(P)$), but no transition is triggered by e (i.e., $\mathit{Enable}((ks, P', GV), e) = \varnothing$), and no deferred event in the current configuration matches the event e (i.e., $\forall\, s \in ks, e \notin s.\widehat{t_{def}}$).

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.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P \backslash \{e\}, \exists\, s \in ks : e \in s.\widehat{t_{def}}, \mathit{Enable}((ks, P', GV, e) = \varnothing, \\ P'' = \mathit{Merge}(e, P'.DP) \end{array}}{(ks, P, GV) \xrightarrow{e} (ks, P'', GV)}$$

Since event e is deferred, it should be merged back to the deferred event pool (i.e., $\mathit{Merge}(e, P'.DP)$). So after the RTC execution, only the event pool is changed to P''.

Deferral Rule 2. 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 deferred event, that has higher priority than the source states of the enabled transitions.

$$\frac{\begin{array}{l} e = \mathit{Disp}(P), P' = P \backslash \{e\}, \exists\, s \in ks : e \in s.\widehat{t_{def}}, \widehat{T} = \mathit{Enable}((ks, P', GV, e), \widehat{T} \neq \varnothing, \\ \forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (ks, P', GV), e), P'' = \mathit{Merge}(e, P'.DP) \end{array}}{(ks, P, GV) \xrightarrow{e} (ks, P'', GV)}$$

$\widehat{T}$ 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 ($\forall\, \tilde{t} \in \widehat{T} \Rightarrow \mathit{deferralConflict}(\tilde{t}, (ks, P', GV), e)$)). As a consequence, only the event pool of the state machine changes. For example, $(\{\textit{Operating, WaitArriveOK, Watch, WaitEnter}\}, (\varnothing, \varnothing, \{\textit{opend}\}), \mathit{Env}1) \xrightarrow{\textit{opend}}$ $(\{\textit{Operating}, \textit{WaitArriveOK, Watch, WaitEnter}\}, (\varnothing, \{\textit{opend}\}, \varnothing), \mathit{Env}1)$ illustrates the application of this rule, where $\mathit{Env}1$ denotes $\{stopNum = 1, mode = \mathit{false}\}$.

To increase the rules readability, we use the following notations. $\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_n) = \mathit{CollectAct}(\tilde{t}_1);\ , \ldots, ;\ \mathit{CollectAct}(\tilde{t}_n)$ denotes the behaviours collection along transitions $\tilde{t}_1, \ldots, \tilde{t}_n$. $\mathit{Merge}(\mathcal{A}(\langle \tilde{t} \rangle), P)$ merges all events generated by actions in $\mathcal{A}(\langle \tilde{t} \rangle)$ into event pool P. Function $\mathit{UpdateV}(\mathcal{A}(\langle \tilde{t} \rangle), GV)$ updates global 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 deferred, or the fired transitions

have higher priority over event deferral.

$$\frac{e = Disp(P), P' = P \backslash \{e\}, \widehat{T} \in Firable((ks, P', GV), e), \mid \widehat{T} \mid = n, \\ \langle \tilde{t} \rangle \in Permutation(\widehat{T}), P'' = MergeA(\mathcal{A}(\langle \tilde{t} \rangle), P'), V' = UpdateV(\mathcal{A}(\langle \tilde{t} \rangle), GV)}{(ks, P, GV) \xrightarrow{e} (NextK(ks, \langle \tilde{t} \rangle), P'', GV')}$$

Function $Firable((ks, P', GV), e)$ (defined in [17]) returns a set of maximal non-conflicting subset of enabled transitions. The firable set of transitions[4] will be executed in an order specified by $\langle \tilde{t} \rangle$. Function $Permutation$ (defined in [17]) computes all possible total orders on the set of compound transitions $\widehat{T}$. Behaviours 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.

$$\frac{e = Disp(P), P' = P \backslash \{e\}, \widehat{T} \in Firable((ks, P', GV), e), \mid \widehat{T} \mid = n, \\ \tilde{t}_i^1 \in \widehat{T}, \tilde{t}_i^1.tv \in C_{ps}, \langle \tilde{t} \rangle = (\tilde{t}_1, \ldots \tilde{t}_i^1, \ldots, \tilde{t}_n) \in Permutation(\widehat{T}), \\ GV' = UpdateV(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), GV), P'' = MergeA(\mathcal{A}(\tilde{t}_1, \ldots, \tilde{t}_i^1)), P'), \\ \tilde{t}_i^2 \in Firable((\{last(\tilde{t}_i^1).tv\}, P'', GV'), e), P''' = MergeA(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), P''), \\ GV'' = UpdateV(\mathcal{A}(\tilde{t}_i^2 \ldots, \tilde{t}_n), GV')}{(ks, P, GV) \xrightarrow{e} (NextK(ks, \langle \tilde{t} \rangle), P''', GV'')}$$

Compound transition t_i is split by a choice pseudostate into t_i^1 and t_i^2. The second half of t_i is evaluated based on environment GV' . In Fig. 1, $(\{$*Operating*, *WaitArriveOK*, *Watch*, *WaitDepart*$\}$, $(\varnothing, \varnothing, \{$*opend*$\})$, $Env1)$ $\overset{opend}{\dashrightarrow}$ $(\{$*Operating*, *Choice2*$\}$, $(\varnothing, \varnothing, \varnothing)$, $Env0)$ $\dashrightarrow$ $(\{$*Idle*$\}, (\varnothing, \varnothing, \varnothing), Env0)$[5] illustrates the application of this rule.

4.4 System Semantics

A UML state machine models the dynamic behaviour of one object within a system. But state machines representing different components of a system may interact with each other. In order to verify the correctness of the overall system behaviours, we need to capture the message passing sequences between state machines in the system.

Definition 12 (Semantics of a system). *The* semantics of a system *is defined as a Labelled Transition System (LTS)* $\mathcal{L} \triangleq (\mathbb{S}, \mathcal{S}_{init}, \rightsquigarrow)$. *In this expression,* $\mathbb{S}$ *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* $\mathcal{L}$. *And* $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$ *is the transition relation of* $\mathcal{L}$, *defined below.*

[4] We assume the UML state machines obey well-formedness rules. If more than one non-conflicting sets of transitions are fiable, the choice of which set to execute is non-deterministic.

[5] We use $Env0$ to represent the set $\{stopNum = 0, mode = false\}$. The dashed arrow $\dashrightarrow$ represents an instant stop in a choice pseudostate.

Table 2. Evaluation results

Model	Property	Result	USM^{2}C				HUGO				
			Time(s)	State	Transition	Mem (KiB)	TTime(s)	ETime(s)	State	Transition	Mem (KiB)
RailCar	Prop1	not valid	0.013	30	34	43,342	-	-	-	-	-
RailCarO	Prop1	valid	0.011	44	54	43,058	-	-	-	-	-
BankATM	Prop2	valid	0.009	25	28	917.5	0.231	0.050	578	1,133	98,528
DP2	deadlock	not valid	0.005	39	65	2,318	0.196	0.111	12,766	42,081	98,918
TollGate	Prop3	valid	0.110	36	50	43,345	0.197	0.505	61,451	256,807	100,578

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

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \rightarrow k'_j, e = SendSignal(j,l), Merge(e, EP_l)}{(k_1,\ldots,k_l,\ldots,k_j,\ldots,k_n,) \rightsquigarrow (k_1,,\ldots,k'_l,\ldots,k'_j,\ldots,k_n)}\ [\ LTS2\ ]$$

$$\frac{|||^{C}_{i\in[1,n]} Sm_i, k_j \rightarrow k'_j, e = Call(j,l), e \in C, k_l \xrightarrow{e} k'_l}{(k_1,\ldots,k_l,\ldots,k_j,\ldots,k_n) \rightsquigarrow (k_1,\ldots,k'_l,\ldots,k'_j,\ldots,k_n)}\ [\ LTS3\ ]$$

All the state machines in the system are executed non-deterministically. Rule LTS1 captures the normal situation that a single state machine is executed without communicating with other state machines. The notation with prime, i.e., k'_j, represents the new configuration after executing an RTC step. Rule LTS2 defines asynchronous communication, i.e., the executing state machine (Sm_j) sends an asynchronous message ($e = SendSignal(j,l)$) to another state machine (Sm_l). The state machine receiving the message merges the message into its own event pool. Rule LTS3 defines synchronous communication. In this case, the callee state machine (Sm_l) is triggered by the call event ($e = Call(j,l), e \in C$). The caller state machine (Sm_j) cannot finish its RTC step until the callee has finished execution. For example in Fig. 1, if state machine *Car* and *Handler* are in configuration $(\{$*Operating, Crusing*$\}, (\varnothing,\varnothing,\{$*alert100*$\}, Env1)$, $(\{$*WaitDepart*$\}, (\varnothing,\varnothing,\varnothing), \varnothing)$ separately and event *alert100* is dispatched and fires transition t_{12}. The behaviour associated with t_{12} invokes a call event (that is *arriveReq* = *Call*(*Car*, *Handler*)) in *Handler* state machine. The *Handler* state machine consumes the call event and execute an RTC step. After applying rule LTS3, the system is $((\{$*Operating*, *WaitArriveOK, Watch, WaitEnter*$\}, (\varnothing,\varnothing,\varnothing), Env1), (\{$*WaitPlatform*$\}, (\varnothing,\varnothing,\varnothing), \varnothing))$.

5 Implementation and Evaluation

We have implemented the formal semantics in a self-contained tool USM^{2}C [2]. It supports model checking of deadlock, LTL properties, and step-wise simulation. . We compared USM^{2}C with HUGO [13] on 5 examples used in literature, viz., *RailCarO* [10],

Table 3. Scalability evaluation result

N	Time (s)	States	Transitions	Memory (KiB)	N	Time (s)	States	Transitions	Memory (KiB)
2	0.005	39	65	2,318	3	0.039	237	589	10,145
4	0.34	1,519	5,079	21,059	5	3.11	9,634	40,366	41,651
6	27.87	63,069	324,275	90,023	7	232.64	398,101	2,385,361	2,852,672

RailCar in Fig. 1 (modifies *RailCarO* to manually introduce bugs[6]), *BankATM* [13], dining philosopher ($n = 2$) and *TollGate* [15]. HUGO is a tool translating UML state machines into Promela models and using Spin to perform model checking.

Results are in Table 2, where Prop1=$\Box(alert100 \rightarrow \Diamond arriveAck)$, Prop2=$\Box(retain \rightarrow ((!cardValid \wedge numIncorrect \geq maxNumIncorrect))$, Prop3=$\Box(TurnGreen \rightarrow \Diamond carExit)$. Our tool finds the manually injected bugs in *RailCar* system, which is out of the capability of HUGO. The results also show that our tool is more efficient in execution time and memory consumption compared to HUGO[7]. The main reason is that the Promela code generated by HUGO has many local transitions, which introduce overheads. For example, in the generated *TollGate* promela code, 7 steps are conducted to move from a initial pseudostate to its target state, while in our model only one (implicit) step is taken. The effect is exponential in case of non-determinism.

We conducted another experiment on the dining philosophers problem to evaluate the scalability of USM^{2}C. Table 3 shows the result of checking deadlock free property (with breadth first search). We can see from the result that USM^{2}C can handle large state spaces caused by non-determinism. Reducing further the state space through techniques such partial-order reduction is the subject 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 can find design errors in the presence of both synchronous and asynchronous communications and is scalable.

6 Related Work

Existing approaches for formalizing UML state machines semantics fall into two major groups, viz., translation-based approaches and direct formalization approaches.

A large number of existing approaches translate UML state machines to an existing formal modelling language, such as Abstract State Machines [11,5,12], Petri nets [6,3], or the modelling language of some model checkers. The verification can be accomplished by relying on verification tools for the translated languages. For example, state machines have been translated to Promela [14], CSP [18], Event-B [21] and CSP# [23]; then Spin, FDR, ProB and PAT model checkers are used to perform the verification, respectively. The translation approaches suffer from the following defects: (1) Due to the semantic gaps, it may be hard to translate some syntactic features of UML state machines, introducing sometimes additional but undesired behaviours. For example

[6] Both examples contain transitions which emanate/enter orthogonal composite states, e.g., the transition from *Cruising* state to *WaitArrivalOK* state, which is not supported by HUGO.

[7] TTime represents the time used to translate UML state machines models into Promela. ETime represents the time used by Spin to do model checking.

in [23], extra events have to be added to each process so as to model exit behaviours of orthogonal composite states. (2) For the verification, translation approaches heavily depend on the tool support of the target formal languages. Furthermore, the additional behaviours introduced during the translation may significantly slow down the verification; and optimizations and reduction techniques (like partial order reduction) may not apply in order to preserve the semantics of the original model. (3) Lastly, when a counterexample is found by the verification tool, it is hard to map it to the original state machine execution, especially when state space reduction techniques are used.

Works directly provide operational semantics for UML state machines are more related to our approach. [22] provides an operational semantics for a subset of UML state machines. The approach uses terms to represent states and transitions are nested into Or-terms, which makes it hard to extend to support the other features. Fecher [7] defines a formal semantics for a subset of UML state machines features. The remaining features are informally transformed to the formalized features. The informal transformation procedure as well as the extra costs it introduces might make it infeasible for tool developing. The work in [19] considers non-determinism in orthogonal composite states. But it supports only a subset of features and neither event pool mechanisms nor RTC steps are discussed. In all those works [22,7,19], behaviours associated with states and transitions are explicitly represented with mapping functions. As a consequence, future changes to the state machines may cause modifications of multiple structures in their syntax definition and the consistencies between those structures need to be properly maintained. Conversely, our semantics preserves the syntax structure specified by the specification and should extend better to future changes and refinements of state machines. For example, if a simple state is refined into a composite state, only the definition of that simple state needs to be changed in our approach, whereas all the mappings related to that simple state need to be changed in their work.

A number of prototype tools were developed to support the verification of UML state machines in the literature. vUML [16] and HUGO [13] are tools that translate UML state machines to PROMELA and use Spin for the verification. TABU [4] and the tool proposed in [20] translate UML state machines to the input language of SMV. JACK [9] is an integrated environment containing an AMC component, which is able to conduct model checking. UML-B [21] is developed to support translation from UML state machines into Event-B model and ProB is invoked to conduct model checking. Among all the tools discussed here, only HUGO and UML-B are currently available. HUGO has compatibility problems with newer versions of Spin (Spin5.x, Spin6.x), thus manual efforts and knowledge of Spin are required for the verification. UML-B is a UML-like notation, which integrates with B.

7 Discussion and Perspectives

In this paper, we provide a formal semantics for the complete set of UML state machines features. Our semantics considers non-determinism as well as the communication aspects between UML state machines, which bridge the gap of current approaches. We have implemented a self-contained tool, USM^{2}C, for model checking various properties for UML behavioural state machines. The experiments show that our tool is effective in finding bugs with communications between different state machines.

We discuss in the following limitations related to our work. (1) We provide basic assumptions for those unclarities found in UML 2.4.1 state machines specifications based on our understanding, which may introduce thread to the validity of our work. (2) We did not formally define the constraint and action language in this work.

Several other issues linked with UML state machines remain unaddressed. As future work, we aim at considering the real-time aspects and object-oriented issues, such as dynamic invoking and destroying objects.

Acknowledgements. We thank the anonymous reviewers for their insightful comments.

References

1. OMG unified language superstructure specification (formal), Version 2.4.1 (August 06, 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.
2. USM^2C, a UML state machines model checker (April 05, 2013), http://www.comp.nus.edu.sg/~lius87
3. André, É., Choppy, C., Klai, K.: Formalizing non-concurrent UML state machines using colored Petri nets. ACM SIGSOFT Software Engineering Notes 37(4), 1–8 (2012)
4. Beato, M.E., Barrio-Solórzano, M., Cuesta, C.E., Fuente, P.: UML automatic verification tool with formal methods. Elec. N. in Th. Computer Sc. 127(4), 3–16 (2005)
5. Börger, E., Cavarra, A., Riccobene, E.: On formalizing UML state machines using ASMs. Information Software Technology 46(5), 287 (2004)
6. Choppy, C., Klai, K., Zidani, H.: Formal verification of UML state diagrams: a Petri net based approach. ACM SIGSOFT Software Engineering Notes 36(1), 1–8 (2011)
7. Fecher, H., Schönborn, J.: UML 2.0 state machines: Complete formal semantics via core state machine. Formal Methods: Applications and Technology, 244–260 (2007)
8. Fecher, H., Schönborn, J., Kyas, M., de Roever, W.: 29 new unclarities in the semantics of UML 2.0 state machines. Formal Methods and Software Engineering, 52–65 (2005)
9. Gnesi, S., Latella, D., Massink, M.: Model checking UML statechart diagrams using JACK. In: HASE 1999, pp. 46–55 (1999)
10. Harel, D., Gery, E.: Executable object modeling with statecharts. IEEE Computer 30, 31–42 (1997)
11. Jin, Y., Esser, R., Janneck, J.: A method for describing the syntax and semantics of UML statecharts. Software and Systems Modeling 3(2), 150–163 (2004)
12. Jürjens, J.: A UML statecharts semantics with message-passing. In: Proceedings of the 2002 ACM Symposium on Applied Computing, pp. 1009–1013. ACM (2002)
13. Knapp, A., Merz, S.: Model checking and code generation for UML state machines and collaborations. In: Proc. 5th W. Tools System Design & Verif, vol. 11, pp. 59–64 (2002)
14. Knapp, A., Merz, S., Rauh, C.: Model checking - timed UML state machines and collaborations. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–416. Springer, Heidelberg (2002)
15. Kong, J., Zhang, K., Dong, J., Xu, D.: Specifying behavioral semantics of UML diagrams through graph transformations. Journal of Systems and Software 82(2), 292–306 (2009)
16. Lilius, J., Paltor, I.P.: vUML: A tool for verifying UML models, pp. 255–258 (1999)
17. Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., Dong, J.S.: A formal semantics for complete UML state machines with communications (report). Technical report, National University of Singapore (2013), http://www.comp.nus.edu.sg/ lius87/uml/techreport/uml_sm_semantics.pdf

18. Ng, M., Butler, M.: Towards formalizing UML state diagrams in CSP. In: SEFM 2003, p. 138 (2003)
19. Schönborn, J.: Formal semantics of UML 2.0 behavioral state machines. Technical report, Inst. Computer Science and Applied Mathematics, Christian-Albrechts-Univ. of Kiel (2005)
20. Shen, W., Compton, K., Huggins, J.: A toolset for supporting UML static and dynamic model checking. In: COMPSAC 2002, pp. 147–152 (2002)
21. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15(1), 92–122 (2006)
22. Von Der Beeck, M.: A structured operational semantics for UML-statecharts. Software and Systems Modeling 1(2), 130–141 (2002)
23. Zhang, S., Liu, Y.: An automatic approach to model checking UML state machines. In: 4th Int. Conf. Secure Software Integration & Reliability etc (SSIRI-C), pp. 1–6. IEEE (2010)