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

Shuang Liu¹, Yang Liu², Étienne André³, Christine Choppy³, Jun Sun⁴, Bimlesh Wadhwa¹, and Jin Song Dong¹

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

January 27, 2013
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 USM²C, a model checker supporting editing, simulation and automatic verification of UML state machines. Experiments show the effectiveness of our approach.
## Contents

1  Introduction ......................................................... 2
2  UML Features, Unclarities and Our Assumptions .......................... 4
   2.1  Introduction of Basic Features of UML State Machines ............. 4
   2.2  Unclarities in UML 2.4.1 State Machine Specification ............... 8
   2.3  Basic Assumptions on UML State Machine Semantics ................. 11
3  Syntax of UML State Machines ...................................... 13
   3.1  Syntax Formalization ......................................... 13
   3.2  Abstraction of the Event Pool .................................. 17
4  A Formal Semantics for UML State Machines .......................... 18
   4.1  Active State Configuration Changes ................................ 19
   4.2  Behavior Execution ............................................ 20
   4.3  The Run to Completion Semantics ................................ 22
   4.4  System Semantics .............................................. 25
5  Implementation and Evaluation ...................................... 26
6  Related Work ....................................................... 29
7  Conclusion ........................................................... 30
8  Auxiliary Functions ............................................... 38
1 Introduction

UML diagrams have become the de facto modeling language in object oriented system design. Particularly, UML state machine diagrams are widely used to model the dynamic behavior of an object, which serve as the basis for code development. However, UML specification is documented in natural language, which introduces inconsistencies and ambiguities. The importance of providing a formal semantics for UML has attracted researchers’ attention [?]. The benefit of a formal UML semantics is threefold. Firstly, it allows more precise and efficient communication between engineers. Secondly, it yields more consistent and rigorous models. Lastly and most importantly, it 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.

There were some works [17, 22, 6, 10, 1] in the literature which provide formal semantics for a subset of UML state machine features. However, some important issues are not addressed. Firstly, none of the existing formalization approaches achieved a full coverage of UML state machine features. Among all the related works, only a few [10, 1] have considered UML 2.0 specifications, which has major changes to UML 1.x specifications as discussed by [10]. All the related work which considered UML 2.0 specifications cover only a subset of UML state machine features. Fecher et al. [10] provided formal semantics for a subset of UML state machine features. The remaining features are informally transformed to the defined subset of features. The semantics defined in [10] blurred the RTC (Run To Completion) step, which is the basic semantic step of UML state machines. Moreover, the informal transformation procedure as well as the extra costs it introduces may make tool implementation based on this approach not feasible. We believe that all the features provided by UML state machine specification should be considered, since each of them has particular usage, especially choice, fork, join pseudostates, completion transitions and event deferral, which are commonly used but are often left out in existing formalizations.

Secondly, in the existing approaches, communications between state machines are not considered. 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, the verification of the entire system is more meaningful than its subparts, which are in turn modeled
by respective state machines. For example, Figure 1\(^1\) illustrates the RailCar system which is composed of Car state machine and Rail state machine, which communicate with each other through synchronized events. We need to guarantee that the synchronized events passed between the two state machines lead to correctly coupling, i.e., properties such as deadlock-free must hold throughout the RailCar system. The importance of the communications between state machines has been realized by researchers [10, 14]. But none of existing approaches have provided formalization for the communication aspects of UML 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. [?] 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 [?] 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. We also develop a self-contained tool which can bring model checking of UML state machine diagrams into practice. 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.

The rest of this paper is organized as follows. Section 2 provides the preliminaries of UML state

\(^1\) A modification of example used in [?].
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 7 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 [?]) and the BankATMDB system in Fig. 2 as running examples. 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. The BankATMDB system is composed of the Bank state machine and the ATM state machine. They communicate through synchronized call events and asynchronous messages.

Vertex. UML state machine uses the concept vertex to represent all nodes in the graphical notation. Therefore a vertex is the general designation of state, pseudostate, final state and connection point reference which are introduced below.

Transitions. A Transition is a relation between a source vertex and a target vertex. In Fig. 1, the arrow labeled $t_0$ is a transition. Guards, triggers and effects are associations of a transition. A guard is a boolean constraint which must be evaluated to true in order for the transition it guards to be enabled to execute. A trigger relates an event to a behavior and will cause execution of the behavior when the event specified by the trigger occurs. An effect is a behavior, which is a sequence of actions $^3$. There are three kinds of transitions, i.e., internal transition, local transition and external

---

$^2$ A modified version of the example used in [?]

$^3$ Action is a basic unit of behavior specification. It takes a set of inputs and turns them into a set of outputs. Actions
transition. Internal transitions have identical source and target states and they do not leave or enter any states. Local transitions emanate from a composite state or the entry point of a composite state and target one of its substates, as a consequence, local transitions do not leave any states, but will enter some states. All the other transitions which both leave and enter some states are external transitions. The container of a transition is the region which owns the transition. The container of a transition is not precisely defined in UML state machine specification. We will discuss this in detail in Section 2.2. 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. States are nodes which represent the scenarios that some invariant holds. There are three kinds of states, i.e., simple state (Idle in Fig. 1), composite state (Departure in Fig. 1) and submachine state. Composite state is further divided into orthogonal composite state (WaitArrivalOK in Fig. 1) and composite state (Departure in Fig. 1) depending on whether it has exactly one or more include send/receive messages, update values and so on.

Figure 1: The RailCar State Machine
Figure 2: The BankATM State Machine

than one region. States can have optional entry/exit/do behaviors. An entry (resp. exit) behavior is executed on entering (resp. exiting) the state. A do behavior is executed right after an enter behavior (if any) finishes and keeps executing as long as the state is active. The do behavior execution can be interrupted by dispatched event. On completion of a do behavior, a completion event will be generated, which will trigger a completion transition immediately.

A final state (Final1 state in Fig. 1) is a special kind of state which indicates finishing of its enclosing region. It does not have regions, entry/exit/do behaviors.
Pseudostates. Pseudostates are introduced to connect multiple transitions to form complex transition paths. There are 10 kinds of pseudostates. An initial pseudostate (Initial1 in Fig. 1) is used to indicate the default state for each region of a composite state, it cannot act as the target of a transition. A Join pseudostate is used to merge transitions from states (as opposed to pseudostates) in orthogonal regions (join1 in Fig. 1). The transitions entering a join vertex cannot have guards or triggers. A Fork pseudostate (fork1 in Fig. 2) is used to split transitions targeting states in orthogonal regions. The transitions outgoing from a fork vertex must not have guards or triggers. A Junction pseudostate is a semantic-free vertex and is introduced as syntactic sugar to merge/split incoming transitions into outgoing transitions. It represents a guarded branching point. Choice pseudostates (Choice1 in Fig. 1), though also represent a guarded branching, are different from junction pseudostates. When a choice pseudostate is encountered, the transition path emanating from it should be evaluated under the environment when the choice pseudostate is reached instead of the environment at the beginning of the compound transition. In this aspect, choice pseudostates resemble states instead of pseudostate. Entry point (resp. exit point) pseudostate is a way to explicitly indicate the execution of an entry (resp. exit) behavior. On entering (resp. exiting) a composite state, it can also play the role of a junction or fork (resp. join) pseudostate. Entry/exit point pseudostate explicitly indicates the entrance and exit to a submachine or composite state. For example in Fig. 2, EntryPoint1 and ExitPoint1 pseudostates on the border of OperationSM state machine are examples of entry point and exit point pseudostates. Entry point (resp. exit point) pseudostates also allow the fine-grained behavior granularity. Entering a composite state via an entry point pseudostate implies that the entry behavior is executed before the behavior associated with the transition emanating from the entry point pseudostate. Similarly, the behavior associated with the transition targeting an exit point pseudostate should be executed before the exit behavior of the composite state. History pseudostate is a mechanism for the state machine to remember the previous active “snapshot” of its container composite state. History pseudostate is further divided into shallow history pseudostate and deep history pseudostate. Shallow history pseudostates indicate the last active substate of its direct container state. Deep history pseudostates indicate all the last transitively active substates of its direct container state. History pseudostates can act as both source and target of a transition. Transition emanating from a history pseudostate indicate the default history state\[5\]. Entering a terminate pseudostate represents the termination of object which was active on the current state machine.

\[5\] Default history state is entered when its container composite state has never been activated or the last active substate was a final state.
Without exiting any states nor executing any exit actions, the state machine terminates immediately.

**Connection Point Reference.** It is an entry/exit point of a submachine state. It refers to entry/exit pseudostate of the state machine that the submachine state refers to. In Fig. 2, the two small circles *EntryPoint* and *ExitPoint* on the border of *Operation* submachine state are connection point references to the *EntryPoint* and *ExitPoint* pseudostates of *OperationSM* state machine.

**Active State Configuration.** An active State configuration is the set of active states of a state machine when it is in a stable status, i.e., waiting for some event to trigger the next RTC step (introduced below). For example in Fig. 2, \{*Idle*\} and \{*CardValid, PinCorrect, Verifying*\} are an active state configurations of *Bank* State Machine.

**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 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

Since UML state machine specification is documented in natural language, ambiguities may arise in the descriptions. Fecher et.al. [?] listed 29 unclarities of UML 2.0 state machine specification. We find six new ambiguities in UML 2.4.1 state machine specification, mainly in the aspect of non-determinism in UML state machine. We provide discussions on the newly found ambiguities in this section. **Transition Execution Sequence.** When defining transition execution sequence, transitions and compound transitions are used interleavingly. This makes it unclear whether the rule is applied to a transition or to a compound transition.

““Once a transition is enabled and is selected to fire, the following steps are carried out in order: The main source state is properly exited. Behaviors are executed in sequence following their liner order along the segments of the transition: The closer the behavior to the source state, the earlier it is executed.””

[?, Chapter 15, Transition, Semantics, Transition execution sequence p585]
**Transition Execution Sequence.**

Since each transition can have only one behavior and the concept of segment is only used on compound transitions. The above statement raises ambiguity. For example in Fig. 3, the compound transition $t1.t3$ will always be executed (guard of $t3$ is always true). If we define the transition execution sequence based on the compound transition, then when enter state $S1$, 0 should be printed (the behavior execution sequence is $i = 0; i + +; i − −; print(i)$). But if we define the transition execution sequence based on a single transition, the behavior execution sequence should be $i = 0; i + +; i = i * 2; i − −; print(i)$ and value 1 will be printed.

![Diagram](image.png)

**Figure 3: Illustration of transition execution sequence**

**Conflict Resolution in Presence of Choice Pseudostate.** An RTC step in UML state machine represents the effects of dispatching one event, which may include multiple compound transitions in different orthogonal regions. All the firable compound transitions should be decided before the RTC execution, except when a choice vertex is encountered. The dynamic evaluation on a choice vertex may introduce new conflicts with existing firable compound transitions. For example in the Bank State Machine of Fig. 2, suppose transition $t4$ and $t7$ are enabled by the same event. Since both of their target vertices are choice pseudostates, a dynamic evaluation would be taken on reaching the choice pseudostate. Suppose the guard condition of transition $t5$ is evaluated to true, then the orthogonal composite state Verifying will be exited, which implies the source state of transition $t7$ will also be exited. So the compound transition $t4.t5$ conflicts (see Definition 1) with transition $t7$. But this cannot be decided until the choice pseudostate is reached. More clarification should be provided for this situation.

**Basic Interleave Execution Step.** If compound transitions in orthogonal regions are fired by
the same event, it is not clearly defined in which interleaving granularity should the compound transitions execute, i.e., either to interleave on transition level or on compound transition level. For example in *Bank* State Machine of Fig. 2, suppose we compound transition \( t4.t6 \) and compound transition \( t7.t8 \) are fired. Do we allow the execution trace of \( t4\rightarrow t7\rightarrow t6\rightarrow t8 \)? We have shown that different execution orders may cause different effects in global variable updates in the previous description. This issue should also be clearly stated in UML state machine specifications.

**LCA of a compound transition.** The LCA operation is defined on the state machine but is not precisely stated. LCA of a compound transition is defined as a region or orthogonal composite state of the LCA of its source and target states. [?], p.584]. But for a transition whose source and target vertices are identical, either it is an internal transition or an external transition, the LCA is not clear, since LCA for a single state is an ambiguous statement.

**Container of a transition.** The suggested container of a transition is the Least Common Ancestor (LCA)\(^6\) of source and target vertices of the transition. But the LCA of two vertices can be a region or an orthogonal composite state and the container of a transition is defined as “the region that owns this transition”. Moreover, due to the ambiguities for LCA, for an external transition which has identical source and target vertex, LCA may also not be a good definition for containers.

**Order issue of entering orthogonal composite states.** When entering orthogonal composite states, no interleaving order is specified. The UML2.4.1 specification [?] just specifies that “each one of its orthogonal regions is entered.” This statement is not clear enough. For example in Figure ??, when transition \( t2 \) is fired, in which order should state \( STop \) and all its substates be entered? Is it a total concurrency on all levels of involved orthogonal region, meaning that the interleaving order \( STop\rightarrow SubS2\rightarrow ss1\rightarrow SubS1\rightarrow ss2 \) allowed? Is it the case that \( SubS2 \) can only interleave with the states that is in the same hierarchy \( (SubS2) \) with it?

\(^6\)Refer to Definition 1 for formal definition.
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 decide to adopt a single transition execution sequence in our work, i.e., the behavior execution sequence should 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 instead of a transition as the interleaving execution step. As discussed in Section 2.2, this is not clearly stated in UML state machine specification. We decide to adopt a compound transition as basic interleaving step since a compound transition is a semantic complete path.

Order issues of entering orthogonal composite states

All possible interleaving orders between its substates that are entered due to entering the orthogonal composite state are allowed, as long as the hierarchical order is preserved, i.e., composite states are
entered before their substates. So in the example of Section 2.2, the order \( \text{Stop} \rightarrow \text{SubS2} \rightarrow \text{ss1} \rightarrow \text{SubS1} \rightarrow \text{ss2} \) is allowed.

**Compositional Operators**

In this work, we use ; , ||, \( ||^C \), \( \nabla \) to represent sequential, interleave, interleave with synchronous communication (on events in set \( C \)) and interrupt composition respectively.

\[ || \] represents a non-determinism in the execution orders of all the involved objects. So if \( n \) objects are involved, there would be \( n! \) (number of permutations on \( n \) objects) possible sequential composition execution orders on the involved objects.

\[
\square_{p \in [1,n]} O_{p1}; \ldots; O_{pn} \quad [ \text{interleave} ]
\]

The operator \( \square \) represents a non-deterministic choice among all the possible permutations on all the involved objects. The indexes \( p_1, \ldots, p_n \) represents a permutation on all the involved objects.

Interleaving composition with synchronous communications is a special case of interleaving since it requires the state machine to synchronize on the specified event (In UML state machine, the synchronized message passing is accomplished particularly via call events.) This composition requires both the caller and the callee to finish execution, the callee does not start until receiving the call event from the caller and the caller does not finish execution until callee returns. We use the symbol \( ||^C \) to represent interleave with synchronous communications. \( C \) is the call event alphabet, i.e., the set of events which the involved state machines should synchronize on. We formally define interleaving with synchronous communications as follows.

\[
O_1 ||^C O_2 \triangleq \begin{cases} 
O_1; O_2, & O_1!c \in C \land O_1!c \simeq O_2.?c \\
O_2; O_1, & O_2!c \in C \land O_2!c \simeq O_1.?c \\
O_1 || O_2, & O_1!c \notin C \land O_2!c \notin C
\end{cases}
\]

The symbol \( O!c \) represents the call event generated by object \( O \) and \( O?c \) represents the event specified by Object \( O \). The operator \( \simeq \) represents the match operation. The detailed matching rules are not specified.

A Do activity is executed concurrently with other behaviors of its belonging state. It may be interrupted by some triggered transitions which causes its belonging state to be inactivated. In order to express such an interruption semantics, we introduce the interruption operator (\( \nabla \)). \( \alpha \nabla e \) means that the execution of behavior \( \alpha \) will be terminated and the effect triggered by \( e \) will follow to execute.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Symbol</th>
<th>Type</th>
<th>Symbol</th>
<th>Pseudostate type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K_S$</td>
<td>active state config</td>
<td>$B$</td>
<td>boolean</td>
<td>$SH_{ps}$</td>
<td>deep history</td>
</tr>
<tr>
<td>$T$</td>
<td>compound transition</td>
<td>$C$</td>
<td>constraints</td>
<td>$I_{ps}$</td>
<td>initial</td>
</tr>
<tr>
<td>$K$</td>
<td>configurations</td>
<td>$S_f$</td>
<td>final state</td>
<td>$C_{ps}$</td>
<td>choice</td>
</tr>
<tr>
<td>$\langle T \rangle$</td>
<td>compound transition list</td>
<td>$S$</td>
<td>state</td>
<td>$J_{op_{ps}}$</td>
<td>join</td>
</tr>
<tr>
<td>$V$</td>
<td>vertex</td>
<td>$T_{rig}$</td>
<td>triggers</td>
<td>$J_{up_{ps}}$</td>
<td>junction</td>
</tr>
<tr>
<td>$K_V$</td>
<td>active vertex config</td>
<td>$T$</td>
<td>transition</td>
<td>$T_{ps}$</td>
<td>terminate</td>
</tr>
<tr>
<td>CPR</td>
<td>connection point reference</td>
<td>$E$</td>
<td>event</td>
<td>$E_{n_{ps}}$</td>
<td>entry point</td>
</tr>
<tr>
<td>SM</td>
<td>state machine</td>
<td>$R$</td>
<td>region</td>
<td>$F_{ps}$</td>
<td>fork</td>
</tr>
<tr>
<td>$B$</td>
<td>behaviors</td>
<td>$PS$</td>
<td>pseudostate</td>
<td>$SH_{ps}$</td>
<td>shallow history</td>
</tr>
<tr>
<td>$\langle B \rangle$</td>
<td>behavior list</td>
<td>$\mathbb{N}$</td>
<td>natural number</td>
<td>$E_{x_{ps}}$</td>
<td>exit point</td>
</tr>
</tbody>
</table>

### 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.

#### 3.1 Syntax Formalization

We use tuples as syntax domain and refer to the OMG specification [?, Chapter15] as the basis for syntax definition.

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

- $\hat{r} \subset R$ is the set of regions directly contained in this state.
- $\hat{t}_{def} \subset Trig$ 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.
- $\hat{en} \in PS$ and $\hat{ex} \in PS$ are the entry point reference and exit point reference associated with the state.
- $\hat{cpr} \subset CPR$ 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.

• $\hat{t} \subset T$ 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 $ps = (\iota, \hat{h})$, where $\iota \in R$ is the region in which the pseudostate is defined, and $\hat{h} \in S$ 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.

Reaching a final state means enclosing of the region directly contains it. If a region is enclosed by a final state, the composite state which is the direct ancestor of the region will generate a completion event and trigger the completion transition emanates from it. For orthogonal composite states, only when all the orthogonal regions are enclosed by final states can it generate a completion event. The $\iota$ field of a final state is the composite or orthogonal composite state that contains the direct ancestor region of the final state.

**Definition 4 (Connection Point Reference)** A Connection Point Reference is defined as a tuple $(\hat{e}n, \hat{e}x, s)$ where

- $\hat{e}n \subset En_{ps}$ is the entry point kind pseudostates corresponding to this connection point.
- $\hat{e}x \subset Ex_{ps}$ is the exit point kind pseudostates corresponding to this connection point.
- $s$ is the state in which the connection point reference is defined.
For example in Fig. 2, the connection point reference *EntryPoint* (resp. *Exitpoint*) of *Operation* sub-machine state refers to the *EntryPoint* (resp. *Exitpoint*) pseudostate in *OperationSM* state machine and is represented as (*EntryPoint*, ε, *Operation*) (resp. (ε, *ExitPoint*, *Operation*)), respectively.

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 \( t = (sv, tv, \hat{t}g, g, \alpha, \iota, \hat{t}c) \) where:

- \( sv \in V, tv \in V \) are the source and target vertex of the transition respectively.
- \( \hat{t}g \subset Trig, g \in C \) 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.
- \( \hat{t}c \) is a set of tuples of the form \( \text{segt} = (ss, \alpha_{st}, \iota_{st}) \). 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 \( \alpha_{st} \in B \) is the behavior associated with the segment transition. \( \iota_{st} \in R \) is the container of the segment transition.

We group all the transitions that emanate from a fork pseudostate (resp. end in a join pseudostate)\(^8\) together with the transition ends in the same fork pseudostate (resp. emanates from the same join pseudostate). A transition with a non-empty \( \hat{t}c \) field is called a *fork* (resp. *join*) transition depending on the type of the pseudostate. For notation convenience, we will use the single transition ending in the fork (resp. emanate from the join) pseudostate to represent the fork (resp. join) transition and it is referred to as the main transition of the fork (resp. join) transition. For example, in Fig. 1, transition \( t_{10} \) is the main transition of the join transition it represents.

Entry/exit point vertices have the semantics with fork/join vertices when orthogonal regions are involved. But they can be involved in more general scenarios where only one region is involved. We treat exit point pseudostate the same way with join pseudostate and entry point pseudostate the same way with fork pseudostate.

---

\(^7\)The other end is the fork (resp. join) pseudostate.

\(^8\)The transitions which compose a compound transition are called segment transitions.
The UML specification specifically defines two concepts on transitions, viz., the main source and main target of a transition. They are the highest level state left/entered when firing the transition. Formal definition of the two operations are provided in Definition 1 in Appendix We also define two functions \( \text{isFork}(t) \) and \( \text{isJoin}(t) \) to decide whether the transition \( t \) is a fork transition/join transition. For convenience of the following definition, we use notation \( t.\hat{s}v \triangleq \{ t.sv \} \cup \{ tp.ss \mid tp \in t.\hat{tc} \} \) to represent the set of source vertices when a join pseudostate is involved and notation \( t.\hat{tv} \triangleq \{ t.tv \} \cup \{ tp.ss \mid tp \in t.\hat{tc} \} \) to represent the set of target vertices when a fork pseudostate is involved.

The sequences of actions along a fork (resp. join) transition is defined as follows:

\[
t.\tilde{\alpha} \triangleq \begin{cases} t.\alpha; \parallel \{ \text{seg}t.\alpha \mid \text{seg}t \in t.\hat{tc} \} & \text{isFork}(t) \\ \parallel \{ \text{seg}t.\alpha \mid \text{seg}t \in t.\hat{tc} \}; t.\alpha & \text{isJoin}(t) \\ t.\alpha & \text{otherwise} \end{cases}
\]

The non-determinism raised by multiple segment transitions of a fork (resp. join) transition is captured by the interleave (\( \parallel | \)) operator.

**Definition 6 (Region)** A region is defined as a tuple \( r \triangleq (\hat{v}, \hat{t}) \) where:

- \( \hat{v} \subset (S \cup PS) \) is the set of vertices directly contained in this region.
- \( \hat{t} \subset T \) is the set of transitions which are owned by the region.

**Definition 7 (State Machine)** A UML state machine is defined by a tuple \( sm \triangleq (\hat{r}, \hat{cp}) \), where

- \( \hat{r} \) is top most region which is directly contained by the state machine \( sm \).
- \( \hat{cp} \) 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 \( \tilde{T} = \{ \tilde{t} \mid \tilde{t} \in ST \land \tilde{t}.\hat{sv} \in S \land \tilde{t}.\hat{tv} \in S \} \). where \( st \in ST \equiv st \in T \lor \exists st_i, st_j \in ST : \text{last}(st_i) = \text{first}(st_j) \wedge st = st_i \prec st_j \).

The operator \( \prec \) represents the operation of connecting transitions in order. The following functions are defined on compound transitions:

- \( \text{first} : \tilde{T} \rightarrow T \) is an operation which returns the first segment of a given compound transition.
last : $\tilde{T} \to T$ is an operation which returns the last segment of a given compound transition.

seg : $\tilde{T} \times \mathbb{N} \to T$ returns the $i$-th segment specified by the natural number index ($i$) of a given compound transition.

len : $\tilde{T} \to \mathbb{N}$ This function returns the total number of segment transitions the compound transition is composed of.

prefix : $\tilde{T} \times \tilde{T} \to \mathbb{B}$ specifies the prefix relationship between two compound transitions. Formally,

$$\text{prefix}(\tilde{t}, \tilde{t}' \}) \triangleq \begin{cases} T, & \text{len}(\tilde{t}) < \text{len}(\tilde{t}') \land \forall i \in [1, \text{len}(\tilde{t})] \text{seg}(\tilde{t}, i) = \text{seg}(\tilde{t}', i) \\ F, & \text{otherwise} \end{cases}$$

Recall that in our syntax definition, we define a join/fork transition within a single transition representation. So we use $\tilde{sv}$ and $\tilde{tv}$ instead of $sv$ and $tv$ to represent the general situation, where a join/fork transition may involved. Since a compound transition is a semantic step of interleaving executions, we will use $\tilde{i}$ in some functions related to a transition execution. Therefore we define $\tilde{l}.\tilde{sv} = \text{first}(\tilde{l}, \tilde{sv})$, $\tilde{l}.\tilde{tv} = \text{last}(\tilde{l}, \tilde{tv})$ for the convenience of semantic definitions. The compound transition involving choice pseudostate is a special case since its dynamic evaluation feature requires a temporary stop within a RTC step. We treat a choice vertex as a temporary stop when executing a compound transition, i.e., a dynamic evaluation of enabled transitions based on the current environment will be conducted.

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

3.2 Abstraction of the Event Pool

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.

So when we need to decide which event to dispatch for the next RTC step, we need to check all completion events first before considering deferral events and normal triggering events. Then
we need to inspect all the deferred events if any. Note that a synchronous call event is consumed immediately on its generation. So we do not put synchronous call events into event pools. We provide three pools, i.e., completion event pool (CEP), deferred event pool (DEP) and normal event pool (NEP), to resolve the ordering issues discussed above. When dispatching events, the three event pools are inspected in order, i.e., \( CEP \prec DEP \prec NEP \), where the symbol \( \prec \) represents the preceding partial order. We use \( EP \triangleq CEP \cup DEP \cup NEP \) to represent all kinds of event pools and define the two basic operations, viz \( \text{EventDispatch()} \) and \( \text{Merge}(e, P) \) on \( EP \) to represent event dispatch and event merge operations.

**Definition 10 (Event dispatch)** The following function formally defines an event dispatch mechanism.

\[
!EP \triangleq \begin{cases} 
!CEP & \text{if } CEP \neq \emptyset \\
!DEP & \text{if } CEP == \emptyset \land DEP \neq \emptyset \land \text{uncyclic}(DEP) \\
!NEP & \text{if } CEP == \emptyset \land DEP == \emptyset \land NEP \neq \emptyset \\
e & \text{otherwise}
\end{cases}
\]

The function definition guarantees that the order \( CEP \prec DEP \prec NEP \) is preserved. The operator \( ! \) represents output from the event pool. The deferral pool is different from the other event pools. Indeed, if a dispatched event from the deferral pool is deferred in the current state, it will be added back to the deferral event pool. We need to avoid inspecting the same event in deferral event pool multiple times while staying in the same active state configuration. So we use \( \text{uncyclic}(DEP) \) to represent that no events in \( DEP \) will be dispatched more than once within the same active state configuration. The OMG specification does not explicitly define the storage structure of events and the event dispatch orders in order to support different priority-based schemes. So we will use a general function to model the event dispatch mechanism and leave the details open for all possibilities.

**Definition 11 (Event Merge)** The following function formally defines merge an event into the event pool referred to by parameter \( P \).

\[
\text{Merge}(e, P) \triangleq \begin{cases} 
CEP \dashv e & \text{if } P == CEP \\
DEP \dashv e & \text{if } P == DEP \\
NEP \dashv e & \text{if } P == NEP
\end{cases}
\]

Here we choose to use the symbol \( \dashv \) instead of \( \cup \) to represent the operation of join event \( e \) into the event pool \( EP \) and avoid the confusion of treating the event pool as a set.
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 effects 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

One effect of an RTC step is to cause active state changes in a UML state machine, i.e., to move from one active state configuration to the next active state configuration. An active state configuration is a set of states which are in active status at the same time. Active state configuration is defined hierarchically in UML state machines. If a simple state is active, then all its ancestor composite states are active too. If a composite state is active, then there can be at most one active substate in each of its regions. We formalize the constrains as follows:

**Definition 12 (Active State Configuration)** An active state configuration

\[ ks \triangleq \{ s \mid (\forall s \in ks \land (s \in S_s \lor s \in S_f) \Rightarrow \forall s' \in S \land isAncestor(s', s), s' \in ks) \land (\forall s \in ks \land (s \in S_c \lor s \in S_o) \Rightarrow (\forall r \in s.\tilde{r}, \exists s' \in r.\tilde{v} : s' \in ks \Rightarrow (\forall s'' \in r.\tilde{v}\{s'\}, s'' \notin ks)))\} \]

An active state configuration describes a stable state status of a UML state machine execution, i.e., the status when the previous RTC step finishes and the state machine waits for the event dispatcher to dispatch the next event. So an active state configuration only contains states and final states. But an RTC step may contain multiple compound transitions which are composed of multiple segment transitions connected by pseudostates.

Remind that we define the transition execution sequence based on transitions, which may emanate from or target pseudostates. So we use Active Vertex Configuration \( \tilde{K}_V \) 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.

**Next Active State Configuration.** \( NextK : \tilde{K}_S \times \langle \tilde{T} \rangle \rightarrow \tilde{K}_S \) is a function that computes the next active state configuration after executing the list of compound transitions. Formally:

\[ NextK(ks, (\tilde{t}_1; \ldots; \tilde{t}_n)) \triangleq NzK(ks_{n_i}, \tilde{t}_n), \text{where } \forall i \in [2, n], ks_i = NzK(ks_{i-1}, \tilde{t}_{i-1})\wedge ks_1 = ks. \]
Function $N_x K : K_S \times \tilde{T} \rightarrow K_S$ computes the next active state configuration after executing a compound transition. Formally, we have: $N_x K (k_s, \tilde{t}) \triangleq N_x P K (k_v, seg(\tilde{t}, n))$, where $n = \text{len}(\tilde{t})$, $k_v = k_s$, and $\forall i \in [2, n], k_v = N_x P K (k_v, seg(\tilde{t}, i - 1))$.

Function $N_x P K : K_V \times T \rightarrow K_V$ computes the next active vertex configuration after executing a transition. Formally: $N_x P K (kv, t) \triangleq kv \setminus \text{Leave}(kv, t) \cup \text{Enter}(t)$. Functions \text{Leave} and \text{Enter} represent the set of states left and entered after executing a transition and are formally defined in Appendix 1.

### 4.2 Behavior Execution

Another effect of executing an RTC step is to cause behaviors to be executed. The behaviors may include the exit behaviors of states exited by the fired transition, behaviors along the fired transitions and the entry behaviors entered by the fired transition. All the behaviors should be collected in the correct order. We define the following functions to collect the behavior execution sequence.

**Exit Behavior.** $\text{ExitBehavior} : K_V \times T \rightarrow \langle B \rangle$ collects the ordered exit behaviors of states that a given transition leaves in the current vertex configuration. Formally:

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

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

$$\text{ExitR}(kv, r, t) \triangleq \begin{cases} 
\text{SetSH}(h, v); \text{ExitV}(kv, v, t) & \text{if } r \in R \land \exists v \in r.\tilde{v} : v \in kv \land v \in S \\
& \land \exists h \in SH_{ps} : h \in r.\tilde{v} \\
\text{SetDH}(h, v); \text{ExitV}(kv, v, t) & \text{if } r \in R \land \exists v \in r.\tilde{v} : v \in kv \land v \in S \\
& \land \exists h \in DH_{ps} : \text{isAncestor}(h.i, r) \\
& \land \text{isAncestor}(t.i, h.i) \\
\text{ExitV}(kv, v, t) & \text{if } r \in R \land \exists v \in r.\tilde{v} : v \in kv \\
& \land \forall s' \in r.\tilde{v}, s' \notin SH_{ps} \\
& \land \exists h \in DH_{ps} : \text{isAncestor}(h.i, r) \\
& \land \text{isAncestor}(t.i, h.i)
\end{cases}$$
The exit behaviors of executing a transition are collected recursively starting from its main source state (computed by function \( \text{MainSource}(t) \)). Exit behaviors should be collected in an innermost-out order. No composite state should be exited before their substates exit. We define function \( \text{ExitV} \) and \( \text{ExitR} \) 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 \( \text{SetSH} \) and \( \text{SetDH} \) respectively) before exiting the region. Functions \( \text{SetSH} \) and \( \text{SetDH} \) set the contents of the corresponding history pseudostate properly on exiting the region. Exiting simple states means terminate the do behavior (if any) and executes the exit behavior, as defined by function \( \text{exit} \). Otherwise, a pseudostate must be encountered and no behavior is collected (denoted by \( \epsilon \)).

**Entry Behavior.** \( \text{EntryBehavior} : T \rightarrow \langle B \rangle \) collects the ordered entry behaviors of states a given transition enters. Formally:

\[
\text{EntryBehavior}(t) = \text{EntryV}(\text{MainTarget}(t), \text{Enter}(t))
\]

\[
\text{EntryV}(v, \hat{V}) \triangleq \begin{cases} 
\text{v.} \alpha_{en}; (||_{r \in v.\hat{r}} \text{EntryR}(r, \hat{V}) || v.\alpha_{do}) & \text{if } v \in S_o \\
\text{v.} \alpha_{en}; (\text{EntryR}(r, \hat{V}) || v.\alpha_{do}) & \text{if } v \in S_c \\
\text{v.} \alpha_{en}; \text{v.} \alpha_{do} & \text{if } v \in S_s \\
\text{GenEvent(v.ι)} & \text{if } v \in S_f \land \forall r \in v.\hat{r}, \\
& \exists s' \in r.\hat{v} : s' \in kw \Rightarrow s' \in S_f \\
\epsilon & \text{otherwise}
\end{cases}
\]

\[
\text{EntryR}(r, \hat{V}) \triangleq \text{EntryV}(s', \hat{V}) \text{ where } r \in R \land s' \in r.\hat{v} \land s' \in \hat{V}
\]

Entry behaviors are collected in a similar manner to exit behaviors, except that the order should be outermost-in. We define the function \( \text{EntryV} \) and \( \text{EntryR} \) to recursively collect the entry behaviors of all the vertices in \( \hat{V} \) in order. All the states entered by firing the transition \( t \) are computed by function \( \text{Enter}(t) \). 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 \( \text{GenEvent}(s) \) generates a completion event for state \( s.ι \) (the container state of final state \( s \)) and merges the generated event in the completion event pool (CEP). For orthogonal composite
states, we can only generate a completion event when active states in all its regions are final states as constrained by $v \in S_f \land \forall r \in v.t.r, \exists s' \in r.\widehat{v}: s' \in kv \Rightarrow s' \in S_f$. The generated event must be indicated to belong to state $s.t$ since it is meant to trigger a completion transition emanating from $s.t$. For other vertices which do not have particular entry behaviors indicated, their entry behaviors are represented by $\epsilon$.

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

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

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

where $n = \text{len}(\tilde{t})$, $kv_1 = ks$ 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 13 (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 describes the currently active states as well as the current event pool and the global variables. It can be considered as a 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. The rules are in the form of inference rules, with the preconditions above the line and the conclusions below the line.

**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.

$$e =!EP, EP' = EP \setminus \{e\},$$

$$\forall s \in ks, e \not\in s.\widehat{t}_{\text{def}}, \text{Enable}((ks, EP', GV), e) = \emptyset$$

$$(ks, EP, GV) \xrightarrow{} (ks, EP', GV)$$

[Wandering]
Event $e$ is dispatched from event pool ($\!EP$), but no transition is triggered by $e$ (i.e., $Enable((ks, EP', GV), e) = \emptyset$), and no deferred event in the current configuration matches the event $e$ (i.e., $\forall s \in ks, e \notin s.t_{\text{def}}$).

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.

RTC 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.

$$ e = \!EP, EP' = EP \setminus \{e\}, \exists s \in ks : e \in s.t_{\text{def}}, Enable((ks, EP', GV, e) = \emptyset, EP'' = \text{Merge}(e, EP'.DEP), 
(ks, EP, GV) \xrightarrow{e} (ks, EP'', GV) \quad [\text{Deferral1}] $$

$Enable((ks, EP, GV, e) = \emptyset$ represents the fact that no transitions are triggered by event $e$. “$\exists s \in ks : e \in s.t_{\text{def}}$” represents that $e$ is deferred by some states in the current active state configuration.

As a consequence, the state machine remains in the current configuration and the dispatched event is merged to the deferred event pool ($DEP$). Since event $e$ is deferred, it should not be discarded but merged back to the deferred event pool (depicted by $\text{Merge}(e, EP'.DEP)$). So after the RTC execution, only the event pool is changed to $EP''$.

RTC 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.

$$ e = \!EP, EP' = EP \setminus \{e\}, \exists s \in ks : e \in s.t_{\text{def}}, \hat{T} = Enable((ks, EP', GV, e), \hat{T} \neq \emptyset, \forall \tilde{t} \in \hat{T} \Rightarrow \text{deferralConflict}(\tilde{t}, (ks, EP', GV, e)), EP'' = \text{Merge}(e, EP'.DEP), 
(ks, EP, GV) \xrightarrow{e} (ks, EP'', GV) \quad [\text{Deferral2}] $$

$\hat{T}$ 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 ($\exists s \in ks : e \in s.t_{\text{def}}$) and the event deferral has higher priority over transition firing ($\forall \tilde{t} \in \hat{T} \Rightarrow \text{deferralConflict}(\tilde{t}, (ks, EP', GV), e)$). Function $\text{deferralConflict}$ is used to solve deferral conflicts and is formally defined in Definition 1 in Appendix.. 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.

$$ A(\tilde{t}_1, \ldots, \tilde{t}_n) = \text{CollectAct}(\tilde{t}_1); \ldots; ; \text{CollectAct}(\tilde{t}_n) \quad \text{represents the execution of the behaviors along } \tilde{t}_1, \ldots, \tilde{t}_n \text{(i.e., a list of compound transitions).} $$
$\text{Merge}(A((\bar{t})), EP)$ represents merging the event generated by actions in $A((\bar{t}))$ if any into event pool $EP$. $\text{UpdateV}(A((\bar{t})), GV)$ represents updating of global shared variables $GV$ by actions in $A((\bar{t}))$.

**RTC 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.

$$
e = \text{!}\text{EP}, EP' = EP\setminus\{e\}, \tilde{T} = \text{Firable}((ks, EP', GV), e), |\tilde{T}| = n, \langle \bar{t} \rangle \in \text{Permutation}(\tilde{T}), EP'' = \text{MergeA}(A((\bar{t})), EP'), V' = \text{UpdateV}(A((\bar{t})), GV) \tag{Progress}$$

Function $\text{Firable}((ks, EP', GV), e)$ (defined in Definition 1 in Appendix) returns a set of compound transitions which is the maximal non-conflicting subset of enabled transitions, and they have higher priority over event deferral in the current configuration. As a result, the firable set of transitions will be executed in an order specified by $\langle \bar{t} \rangle$, which is an ordered list of compound transitions. Function $\text{Permutation}$ (defined in Definition 1 in Appendix) computes all possible total orders on the set of compound transitions $\tilde{T}$. This function captures the orthogonal composite state level nondeterminism, i.e., when multiple compound transitions are fired. Behaviors are collected along the transition execution sequence following the permutation order (indicated by $A((\bar{t}))$). Active state configuration is changed as computed by function $NextK(ks, (\bar{t}))$. There may be events generated by behaviors during the execution of an RTC step, they will be merged into the event pool. So when an RTC step is finished, the event pool is represented by $EP''$, which includes all the merged events. Another side effect of executing those behaviors is to update the set of global variables. The updated global variables are indicated by $GV'$

**RTC 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.

$$
e = \text{!}\text{EP}, EP' = EP\setminus\{e\},\tilde{T} = \text{Firable}((ks, EP', GV), e), |\tilde{T}| = n,\bar{t}_1 \in \tilde{T}, (\bar{t}) = (\bar{t}_1, \ldots, \bar{t}_1, \ldots, \bar{t}_n) \in \text{Permutation}(\tilde{T})$$

$$GV' = \text{UpdateV}(A(\bar{t}_1, \ldots, \bar{t}_1), GV), EP'' = \text{MergeA}(A(\bar{t}_1, \ldots, \bar{t}_1), EP')$$

$$\bar{t}_2 \in \text{Firable}((\text{last}(\bar{t}_1).tv, EP'', GV'), e), EP''' = \text{MergeA}(A(\bar{t}_2, \ldots, \bar{t}_n), EP'')$$

$$GV''' = \text{UpdateV}(A(\bar{t}_2, \ldots, \bar{t}_n), GV') \tag{ProgressC}$$

$(ks, EP, GV) \xrightarrow{\text{ProgressC}} (NextK(ks, (\bar{t})), EP'''', GV''')$
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 split by a choice pseudostate into $t^1_i$ and $t^2_i$. The second half of $t_i$ is evaluated based on the current environment $GV'$ as indicated by $(\tilde{t}^2_i \in Firable((\{last(\tilde{t}^1_i).tv\}, EP'', GV'), e))$. The firable set of transitions will be executed in an order specified by the Permutation $\langle \tilde{i} \rangle$. Note that although we need to do dynamic evaluation at the point where a choice pseudostate is encountered, the permutation $\langle \tilde{i} \rangle$ can be decided before the execution of the RTC step (meaning just with the knowledge of the first half of transition path, for example $\tilde{t}^1_i$, for a compound transition). Behaviors are collected along the transition execution sequence following the permutation order and generated events are merged into the event pool.

### 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. Verifying the dynamic behavior of a single state machine is far from enough from whole system perspective since interactions between state machines may also cause potential problems as we had discussed in Section 1. 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 14 (Semantics of a system)** The semantics of a system is defined as a Labeled Transition System (LTS) $L \triangleq (S, S_{init}, \leadsto)$, with:

- $S$ is the set of states of $L$. 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;

- $S_{init}$ is the initial state of $L$;

- $\leadsto \subseteq S \times S$ is the transition relation of $L$;

The LTS transition relations are defined as follows.

$$\frac{\parallel_{i \in [1, n]} Sm_i, k_j \rightarrow k'_j}{(k_1, \ldots, k_j, \ldots, k_n) \leadsto (k_1, \ldots, k'_j, \ldots, k_n)} \quad [LTS1]$$
\[ I_i \in [1, n] Sm_i, k_j \rightarrow k'_j, e = SendSignal(j, k), Merge(e, EP_k) \ \ [LTS2] \]

\[ (k_1, \ldots, k_k, \ldots, k_j, \ldots, k_n) \leadsto (k_1, \ldots, k'_k, \ldots, k'_j, \ldots, k_n) \]

\[ I_i \in [1, n] Sm_i, k_j \rightarrow k'_j, e = Call(j, k), e \in C, k_k \xrightarrow{e} k'_k \ \ [LTS3] \]

\[ (k_1, \ldots, k_k, \ldots, k_j, \ldots, k_n) \leadsto (k_1, \ldots, k'_k, \ldots, k'_j, \ldots, k_n) \]

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 \( SendSignal(j, k) \) and \( Call(j, k) \) represent the \( j \)th state machine sends an asynchronous and a synchronous message to the \( k \)th state machine, respectively.

5 Implementation and Evaluation

We have implemented the formal semantics defined in Section 4 in a self-contained tool USM²C. 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, which are intuitive to follow.

We show the correctness of our formal semantics by generating the LTS state space with the simulation functionality of our tool. The result of simulation on \( BankATM \)\(^9\) system with initial value \{numIncorrect = 0, cardValid = true, pinValid = false\} is shown in Fig. 5. We use the state name and the transition name to represent the completion event which triggers the corresponding

\(^9\)A modified version of example used in [14] and is shown in Fig. 6. It obeys the UML2.4.1 state machine specification.
completion transition. For example in Fig 5, the event $\text{CardEntry}_\text{cet}20$ represent that the event is the completion event generated by state $\text{CardEntry}$ and it triggers transition labeled $t20$. The symbol $\text{cet}$ indicates that it is a completion event.

Fig. 5 shows the whole state space for the $\text{BankATM}$ system. It shows that our method correctly generated the state space and we do not have redundancies (as will be discussed later in comparison with HUGO).

The first experiment is a comparison on the $\text{BankATM}$ state machine provided in [14] with the off-the-shelf tool HUGO [14]$^{10}$. The $\text{BankATM}$ 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 compatibility problem with Spin, we manually inspect the Promela code generated by HUGO, write LTL properties accordingly and invoke Spin. The property we checked is Property $1 = \Box (\text{retain} \rightarrow ((\neg \text{cardValid} \land \text{numIncorrect} \geq \text{maxNumIncorrect}))$. It guarantees that when a card is retained, it must be the case that at least $\text{maxNumIncorrect}$ times of wrong pin are entered. This property should hold for the $\text{BankATM}$ system. Both 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. Compared to HUGO, our tool generated half less states and consumes only one-third of the memory used. Since we are unaware of any formal semantics defined for HUGO, we manually inspected the generated Promela code and find the reason that may cause the verification process to visit more states. 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 i.e., set state status, set transition status and execute transitions. Another reason for more memory cost is because of duplicate states. For example in Fig. 5, HUGO will generate two states from the node labeled 2, representing whether to enter state $\text{VerifyingCard}$ first or to enter the state $\text{VerifyingPIN}$ first in the $\text{BankATM}$ state machine (Fig. 6). But either way will lead to the same result in this case since no entry behaviors occur and no behaviors are associated with the fork transitions either. This is another source of redundancy in HUGO. As a consequence, a number of variables are introduced in

---

$^{10}$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: Evaluation Result

<table>
<thead>
<tr>
<th>Model</th>
<th>Communication</th>
<th>Result</th>
<th>USM²C Time (s)</th>
<th>States</th>
<th>Transitions</th>
<th>Memory (KiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RailCar</td>
<td>Property1</td>
<td>not valid</td>
<td>0.017</td>
<td>57</td>
<td>71</td>
<td>44562.6</td>
</tr>
<tr>
<td>RailCar</td>
<td>deadlockfree</td>
<td>not valid</td>
<td>0.011</td>
<td>42</td>
<td>41</td>
<td>9690.9</td>
</tr>
</tbody>
</table>

order to represent those status and more states are generated. Recall that an RTC step is the basic step of UML execution behavior implemented in our tool. As a result, we can avoid having multiple redundant copies of variables which might be introduced in the multiple-step Promela translation. This in turn results in a reasonable amount of savings in the memory requirement of verification process.

The second experiment is on the example in Fig. 1, which modifies the example provided in [?][11] 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 □(alert100 → ♦arriveAck) as well as deadlockfreeness property on RailCar state machine. Our tool reported invalid on both properties. The LTL property, 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 deadlockfree property is violated because, when state Final2 is reached, no outgoing transition can be invoked. The results of the two properties are shown in Table. 2

The third experiment is to evaluate the scalability of our tool. We modeled the dining philosopher problem with UML state machines and conducted model checking with our tool. Fig. 7 shows our state machine for 2 dining philosophers. Each philosopher and each fork are modeled with a separate state machine. We use state machine level concurrency to represent the non-determinism. To increase the extend of non-determinism, we use a hand-shake message passing when requiring forks. Table 3 shows the result of this experiment.

Table 3: Scalability Evaluation Result

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

The data listed in Table 3 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 Related Work

Approaches formalizing UML state machine semantics There are some existing works which provide formal semantics for UML state machines. The most related work are proposed by Fecher [10] and Schönborn [20]. Fecher et.al. [10] 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 defined semantics in [10] blurred the 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 may make it not feasible for automatic tool developing. Schönborn’s work [20] does not achieve full coverage of UML state machine features either. For example junction and choice pseudostates, submachine state are not considered. But this work considered non-determinisms in action executions, which leverage over previous approaches. Latella et al.[17] and Michael Von de Beeck[22] adopt Labelled Transition System (LTS) as a semantic model to formalize UML state machine semantics. Their approaches cover only a very small subset of UML1.x state machine features. Most commonly used constructs such as junction, choice, fork and join
pseudostates, submachine state etc. are not supported. Besides, the work proposed in [4, 5, 6, 8, 12] formalizes UML state machine semantics with Abstract State Machine (ASM) as semantic model. This kind of approaches cover more features than approaches using LTS as semantic models, but choice pseudostate is still not considered. Moreover, some of their formalizations [12], 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 [7, 2, 1]. These approaches do not support most of the pseudostates such as junction, choice, history [7] and fork, join [1]. There are also approaches [18, 13, 15, 14, 16, 9, 3, 23] which translate UML state machine into the input language of various model checking tools such as Spin, SMV, PAT etc. 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 counter example is detected.

Tools for model checking UML state machines

There were some tools developed for the formal verification of UML diagrams. vUML [19] is one of the early tools which translates a UML diagram into PROMELA models. It supports checking of deadlock, livelock, reachability properties. But these properties need the aid of explicitly annotating states with stereotypes and constraints. LTL properties are not supported by vUML. The tool is currently unavailable. HUGO [14] is a tool which aims at verifying the consistencies of UML state machine diagrams against properties specified by collaboration diagram or sequence diagrams. It translates UML state machine diagrams into PROMELA, the input language of Spin Model Checker. It supports deadlock properties, LTL properties. But due to compatibility problems with new versions of Spin(Spin5.x, Spin6.x), a lot of manual efforts need to be involved during the verification procedure and knowledge of Promela and Spin is required. TABU [3] is another tool which supports formal verification of UML state machines. It translates a UML state machine model into the input language of SMV model checker. Unlike vUML and HUGO, it can support verification of LTL properties by providing property patterns which guides the writing of properties. [21] introduced a tool based on SMV model checker. It is capable of checking both the static, i.e. well-formed rules, and dynamic properties of a UML state machine. JACK [11] is an integrated environment based on the use of process algebras, automata and temporal logic, and supports many phases of system development process by integrating different editing tools and verification tools. The AMC component inside JACK is able to conduct model checking against ACTL properties. But the components of JACK communicate with FC2 format, which is not widely supported by the state-of-practice tools. Among all the tools discussed
here, only HUGO is currently available. All of them except JACK conduct a translation approach, which suffers from efficiency [19] and tractability problems. JACK, though directly implement the semantics, is not fully automatic and is unavailable now.

Among all the proposed formal semantics for UML state machines, only some of them consider UML2.x state machine specifications and all of them cover a subset of UML state machine features. None of the existing approaches proposed a formalization for the communication, especially synchronous communication of UML machines. Our work leverages over previous approaches to cover all features of UML2.0 state machines as well as communications between state machines. We also implement our semantics into a self-contained tool, which is capable of model checking safety and LTL properties of UML state machines.

7 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 composite state level nondeterminisms 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 [?]. 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.
Figure 5: Simulation result of BankATM system
Figure 6: The modified version of BankATM state machine
Figure 7: Dining Philosopher state machine
Bibliography


1 Auxiliary Functions

We have defined auxiliary functions which are needed to formalize the dynamic steps of UML state machine. We formalize their defines in this subsection.

Since UML state machine has a hierarchical structure, we define functions which decide the ancestor/decedent relations between states and regions $is\text{Ancestor} : S \cup R \times S \cup R \cup PS \cup S_f \rightarrow \mathbb{B}$ as follows:

$$
is\text{Ancestor}(s, s') \triangleq \begin{cases} 
T, & \text{if } (s \in S \land \exists r \in s.\hat{r} : is\text{Ancestor}(r, s')) \lor 
(s \in R \land \exists s_0 \in s.\hat{v} : is\text{Ancestor}(s_0, s')) \lor s == s' \\
F, & \text{otherwise}
\end{cases}
$$

where $is\text{Ancestor}(s, s') == T$ represents that $s$ is the ancestor of $s'$. The isAncestor relation is reflective and transitive.

**Sequence Transition** A sequence transition$(st)$ is is an ordered list of transitions which satisfy the following conditions: (i)$\forall t \in T, t \in ST$ (ii)$\forall st_i, st_j \in ST : last(st_i) == first(st_j) \Rightarrow st_i \prec st_j \in ST$. So a sequence transition is an ordered list of transitions such that the sibling transitions are linked head-to-tail. The compound transition is a special case of sequence transition where the sources of the first transition and the targets of the last transition are constrained to be states.

**Least Common Ancestor** Least Common Ancestor (LCA) is an operation defined for a state machine. It returns the smallest common ancestor of the given set of vertices. $LCA : \mathbb{P} V \rightarrow R \cup S_o \cup PS$ Formally,

$$
LCA(\hat{V}) \triangleq \begin{cases} 
\ s, & \text{if } \exists s \in \hat{V} : (s \in S_o \land (\forall s' \in \hat{V}, is\text{Ancestor}(s, s'))) \\
\ r, & \text{if } \nexists s \in \hat{V} : s \in S_o \land (\forall s' \in \hat{V}, is\text{Ancestor}(s, s')) \land \\
& (\forall r' \in R : \forall s \in \hat{V}, is\text{Ancestor}(r', s)) \Rightarrow is\text{Ancestor}(r', r)
\end{cases}
$$

LCA of a set of vertices can be a region or an orthogonal composite state. It can either be an orthogonal composite state or a region. We need to guarantee “least” in both situation. If a region acts as the LCA of a set of states, we guarantee “least” by constraint $\forall r' \in R : \forall s \in \hat{V}, is\text{Ancestor}(r', s) \Rightarrow is\text{Ancestor}(r', r)$. This constraint indicates that if both region $r$ and $r'$ are ancestors of the set of vertices $\hat{V}$, then $r'$ must also be the ancestor of $r$, which implies that $r$ is the “least” among all the ancestors of $\hat{V}$.
**Main Source/Target**  The Main source of a transition is the outermost vertex left by the transition and is defined by function $MainSource : T \rightarrow V$ Let $S(t) = t.sv \cup t.tv$

$$MainSource(t) \triangleq \begin{cases} LCA(S(t)), & \text{if } LCA(S(t)) \notin R \\ s, & \text{if } LCA(S(t)) \in R \wedge ((\text{isJoin}(t) \land \exists s \in LCA(S(t)).sv : isAncestor(s, t.sv)) \lor \\
(isJoin(t) \land \exists s \in LCA(S(t)).sv : (\forall s' \in t.sv \setminus t.tv \Rightarrow isAncestor(s, s'))) \end{cases}$$

The Main target of a transition is the outermost vertex entered by a transition and is defined by function $MainTarget : T \rightarrow V$

$$MainTarget(t) \triangleq \begin{cases} LCA(S(t)), & \text{if } LCA(S(t)) \notin R \\ s, & \text{if } LCA(S(t)) \in R \wedge ((\text{isFork}(t) \land s \in LCA(S(t)).tv \land isAncestor(s, t.tv)) \lor \\
(isFork(t) \land s \in LCA(S(t)).tv (\forall s' \in t.tv \setminus t.tv \Rightarrow isAncestor(s, s'))) \end{cases}$$

The Main source/target are defined to capture the hierarchical state structure during execution, i.e. exiting and entering the set of states caused by execution of a transition in a particular order. If the LCA of the source and target states of a transition is an orthogonal composite state, then the main source/target is the orthogonal state. Otherwise, it is the direct subvertex of the LCA that contains the all the source states of $t$.

**Enable**  $K \times E \rightarrow \mathbb{P}ST$ is a function which evaluates the enabled compound transitions under the current configuration and triggering event. Formally,

$$Enable(k, e) \triangleq \{ st | st \in T \lor (st \in ST \land st.tv \in C_{ps} \land st.sv \setminus sv \subset S \cup C_{ps}) \land \\
st.sv \subset k \land (\forall i \in [1, \text{len}(st)] : enable(\text{seg}(st, i), e, k.V)) \}$$

$$enable(t, e, GV) \triangleq \begin{cases} T, & \text{if } \text{Evaluate}(t.g, GV) = \text{T} \land (e \in t.\tilde{g} \lor t.\tilde{g} = \varnothing) \\ F, & \text{otherwise} \end{cases}$$

$Enable(k, e)$ will return the set of sequence transitions of which all the guards of all its segment transitions are evaluated to true under the current configuration $k$ and the dispatched event matches its triggering event $e$. Choice pseudostate which requires explicit information to continue the current RTC step and is considered as a temporal stop. When a compound transition with $n$ choice vertices is encountered, $n + 1$ times of $Enable$ function will be called. The function $enable(t, e, V)$ will determine whether the transition $t$ will be triggered by event $e$ under the current shared variable values $GV$. 

39
**Leave** \( K_V \times T \rightarrow \mathbb{P} S \cup S_f \cup PS \) maps a transition to the set of vertices it leaves on firing. Formally:

\[
\text{Leave}(kv, t) \triangleq \begin{cases} 
\emptyset, & \text{if isAncestor}(t.v, t.i) \\
L_v(\text{MainSource}(t), t.sv, kv), & \text{otherwise}
\end{cases}
\]

\[
L_v(s, \hat{S}, kv) \triangleq \begin{cases} 
\{s''\} \cup L_v(s'', \hat{S}, kv), & \text{if } s \in R \land \{s''\} = s.\hat{v} \cap kv \\
\{s\} \cup \bigcup_{r \in s.\text{regions}} L_v(r, \hat{S}, kv), & \text{if } s \in S_c \lor s \in S_o \\
s', & \text{if } s \in CPR \land \exists ex \in s.\hat{e}x \land \exists s' \in S : ex.i \in s'.\hat{r} \\
\{s\}, & \text{otherwise}
\end{cases}
\]

For internal (transitions which do not leave or enter any states) and local transitions (both satisfies the constrain \( \text{isAncestor}(s, t.i) \)), the set of vertices they left is empty set (indicated by \( \emptyset \)). For external transitions, starting from the main source state, function \( L_v \) recursively compute the set of vertices left on firing the transition. If a region is left by a fired compound transition, then all the current active vertices within the region are left in an innermost-out order. If a composite state is left, all its (orthogonal) regions are left in an innermost-out order. If a connection point reference is left, the submachine state it is defined is left, which means the state machine or composite state represented by the submachine state is left. For simple states, final states and all the other kinds of pseudostates, only themselves are left.

**Enter** \( T \rightarrow \mathbb{P}(S \cup S_f \cup PS) \) maps a transition to the set of vertices it enters on firing. Formally,

\[
Enter(t) \triangleq \begin{cases} 
\emptyset, & \text{if isAncestor}(t.tv, t.i) \\
\text{enter}(\text{MainTarget}(t), t.\hat{tv}), & \text{otherwise}
\end{cases}
\]
where \( \text{enter} : (S \cup S_f \cup PS \cup RCPR) \times \mathbb{P}(S \cup S_f \cup PS) \rightarrow \mathbb{P}(S \cup S_f \cup PS) \) will return all the transitivity activated states due to execution of a transition. Formally,

\[
\text{enter}(ms, \hat{S}) \triangleq \begin{cases}
\{ms\} \cup \bigcup_{r \in s.T} \text{enter}(r, \hat{S}), & \text{if } ms \in S_c \lor ms \in S_o \\
\text{enter}(s', \hat{S}), & \text{if } ms \in R \land (\exists s \in \hat{S} : \text{isAncestor}(ms, s)) \land \\
& \exists s' \in ms.\hat{h} \land \text{isAncestor}(s', s) \\
\text{enter}(\text{initial}(ms), \hat{S}), & \text{if } ms \in R \land (\forall s \in \hat{S}, \neg\text{isAncestor}(ms, s)) \\
\text{defHistory}(ms), & \text{if } (ms \in SH_{ps} \lor ms \in DH_{ps}) \land \\
& (ms.\hat{h} = \emptyset \lor \exists s \in ms.\hat{h} : s \in S_f) \\
ms.\hat{h}! = \emptyset \land \exists s \in ms.\hat{h} : s \in S_f \\
ms.s & \text{if } ms \in CPR \land \exists en \in ms.\hat{en} \land \exists s' \in S : en.\iota \in s'.\hat{r} \\
\{ms\}, & \text{otherwise}
\end{cases}
\]

- For orthogonal (composite) states, the state is entered followed by all its containing regions.
- If the region contains one of the target states\(^{12}\) of the transition, then the substate of the region which is the containing of the target state of the transition is entered.
- If the region does not contain any of the target states of the transition, then the state indicated by its initial pseudostate is entered.
- If a history state is encountered, and it is the first time for its containing state to be activated, or the last accessed state is a final state, the default history state will be entered.
- If a history state is encountered and it is not the first time for its containing state to be activated, and the deepest contents\(^{13}\) recorded by the history pseudostate are not all final states, then the recorded states will be entered.
- If a connection point reference is encountered, then the submachine state it is defined is entered, which means that the composite state/state machine referred to by the submachine state is entered.
- For simple states, final states and all the other kinds of pseudostates, only themselves are entered.

\(^{12}\)For a fork transition, there are multiple target states.

\(^{13}\)The innermost states that is in the state hierarchy of the recorded states.
Conflict \( \mathcal{K}_S \times \mathcal{T} \times \mathcal{T} \rightarrow \mathbb{B} \) is a function which decides whether two compound transitions conflict with each other.

\[
Conflict(ks, \bar{t}, \bar{t'}) \triangleq \\
\begin{cases}
T, & (\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, \text{seg}(\bar{t}, i))) = \emptyset \lor \\
(\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, (\text{seg}(\bar{t'}, i))) = \emptyset) \land \bar{t} \hat{\text{sv}} \cap \bar{t'} \hat{\text{sv}} \neq \emptyset \lor \\
(\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, \text{seg}(\bar{t}, i))) \cap (\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, (\text{seg}(\bar{t'}, i))) \neq \emptyset)
\end{cases} \\
F, \quad \text{otherwise}
\]

There are two situations for two compound transition to be defined as conflicting:

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

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

This is captured by

\[
(\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, \text{seg}(\bar{t}, i))) \cap (\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, (\text{seg}(\bar{t'}, i))) \neq \emptyset.
\]

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

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

This is captured by

\[
(\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, \text{seg}(\bar{t}, i)) = \emptyset \lor (\bigcup_{i \in [1, \text{len}(\bar{t})]} \text{Leave}(ks, (\text{seg}(\bar{t'}, i)) = \emptyset) \land \bar{t} \hat{\text{sv}} \cap \bar{t'} \hat{\text{sv}} \neq \emptyset).
\]

Priority \( \mathcal{T} \times \mathcal{T} \times \mathcal{K}_S \) is a partial relation between two compound transitions. A pair of transitions \((\bar{t}, \bar{t'}) \in \text{Priority}\) in the current active state configuration \(ks\) means that the first segment of transition \(\bar{t}\) has a smaller distance to the innermost simple state in the current active state configuration than the first segment of transition \(\bar{t'}\). Formally,

\[
\text{Priority} \triangleq \{(\bar{t}, \bar{t'}, ks) \mid \exists s \in \hat{\bar{t}}.\hat{\text{sv}} : \forall s' \in \hat{\bar{t'}}.\hat{\text{sv}}, distance(s, \text{Innermost}(ks)) < distance(s', \text{Innermost}(ks))\}
\]

where \(\text{Innermost}(ks) \triangleq \{s \mid s \in S_s \land s \in ks\}\), \(distance(s, \hat{\bar{s}}) \triangleq n\), if \(\forall s' \in \hat{\bar{s}}, \max |s, s'| = n\). Operation \(|s, s'|\) represents the levels of regions between the two states \(s, s'\) in the state hierarchy. If both \(s\) and \(s'\) are simple states, the value of \(|s, s'|\) is 0 regardless of whether \(s\) and \(s'\) are the same states or not. In this case, the priority cannot be decided. This is a semantic variation point in UML state machine v2.4.1 specification. In our approach, we consider such situation as a non-determinism and enumerate all possibilities. The distance operator will
return the maximum distance between a composite state \( s \) and all the simple states which present in the set \( \hat{s} \) and are transitively contained in the composite state \( s \). Since states from orthogonal regions are not comparable, this function guarantees that the distance computation is consistent with the algorithm described in [?], Chapter 15.3.12, Semantics, Firing Priorities, p.576. The Priority of two compound transitions are decided by the priority of their first segments.

**Deferral Conflict** Deferral conflict is the conflict about whether an event should be consumed or not. It is between a state which has deferral event defined and a source state of a transition which consumes the event. We solve such conflicts and return true if the event is deferred by the state and false if the event is consumed by a transition. Formally, function deferralConflict: \( T \times K \times E \) is defined as

\[
\text{deferralConflict}(t, k, e) \triangleq \begin{cases} 
T, & \exists s, s' \in k.ks : (e \in s.\widehat{t}_{def} \land s' \in t.\hat{s}v \setminus t.sv \land \text{isAncestor}(s', s)) \\
F, & \text{otherwise}
\end{cases}
\]

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

“In case of a composite orthogonal state, substates of orthogonal regions may also introduce deferral conflicts. The conflict resolution follows the triggering priorities, where nested states override enclosing states. In case of a conflict between states in different orthogonal regions, a consumer state overrides a deferring state”

[?, Chapter 15.3.11, Semantics, Deferred events]

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

- If the confliction does not involve orthogonal composite state, which means that the involved states must be in the same branch of state hierarchy, then we give higher priority to substates. This is captured by the first condition.

- If the confliction involves orthogonal composite state, we do not compare the state hierarchy as has been done in solving conflicts between transitions in Definition 1, but directly give higher priority to transitions which consumes the current event.

**Firable Transition** \( K \times Trig \rightarrow \tilde{T} \) is the set of enabled transitions of which conflicts are solved by priority rules. Formally, we define Firable Transitions \( \text{FirTrans}(k, e) \triangleq \{ \tilde{t} \mid \exists \tilde{t} \in \text{Enable}(k, e) \land \neg \text{deferralConflict}(\text{first}(\tilde{t}), k, e) \land \)

43
The purpose of the function Firable Transition is to select the largest non-conflicting subset from enabled transitions such that transitions in the selected subset are non-conflicting and have higher priorities over the conflicting ones in the left part. The first step of deciding firable transitions is to check the deferral conflict. deferralConflict(first(\( \tilde{t} \)), k, e) means there is no such conflict or the source state of the (compound) transition is assigned higher priority over the states which have the events deferred, which is the basic condition before we proceed to check conflicts between enabled transitions. The function is defined in a incremental way, i.e., selecting a subset from the set of enabled transitions gradually, from the highest priority to the lowest priority. \( \exists \text{\( \tilde{t}' \)} \in \text{FirTrans}(k, e) : \text{Conflict}(k, ks, \tilde{t}, \tilde{t}') \) guarantees that the newly selected compound transition does not conflict with existing compound transitions in the firable transition set. \( \exists \text{\( \tilde{t}'' \)} \in \text{Enable}(k, e) \setminus \text{FirTrans}(k, e) : \text{Conflict}(k, ks, \tilde{t}, \tilde{t}'') \land \text{Priority}(\tilde{t}'', \tilde{t}, k, k.s) \) guarantees that each time a transition with the highest priority is selected.

**Permutation**  
Function Permutation: \( \mathcal{P} \tilde{T} \rightarrow \langle \tilde{T} \rangle \) takes a set of compound transitions as input and returns all the possible (total order) permutations on the set of compound transitions. This can be implemented by a recursive (on the number of sorted elements) algorithm. The algorithm is listed in 1. The algorithm is is recursively defined, Permutate(list, start, end, &result) compute the permutation from the start position to the end position in the given list. The results are stored in the &result. When function Permutation(\( \tilde{T} \)) is called, the set \( \tilde{T} \) is first given an arbitrary order and is noted as \( \langle \tilde{T} \rangle \). Then, Permutate(\( \langle \tilde{T} \rangle \), 0, Count(\( \langle \tilde{T} \rangle \)), &result) is called and result will be returned.
**Algorithm 1 Permute**

**Input:**
- list: /* the list of items to be permutated */
- start: /* the start position of permutation */
- end: /* the end position of the permutation */
- & result: /* the result, all the total orders computed */

**if** start > end **then**

result.Add(list);

**else**

**for** i = start; i < end; i ++ **do**

swap(list[start], list[i]);

`Permute(list, start + 1, end, &result);`

swap(list[start], list[i]);

**end for**

**end if**