Semantic Variations Among UML StateMachines

Ali Taleghani and Joanne M. Atlee

David R. Cheriton School of Computer Science
University of Waterloo, Canada

Abstract. In this paper, we use *template-semantics* to express the execution semantics of UML 2.0 StateMachines, resulting in a precise description that not only highlights the semantics decisions that have been documented but also explicates the semantics choices that have been left unspecified. We provide also the template semantics for StateMachines as implemented in three UML CASE tools: Rational Rose RT, Rhapsody, and Bridgepoint. The result succinctly explicates (1) how each of the tools refines the standard's semantics and (2) which tools' semantics deviate from the standard.

1 Introduction

Unified Modeling Language (UML) Behavioral State Machines (hereafter called StateMachines) are an object-based variant of Harel statecharts [6] that are used primarily to describe the behaviour of class instances (objects) in a UML model. Their semantics, as defined by the Object Management Group (OMG), is described in a multi-hundred-page natural-language document [19] that is not easy to use as a quick reference for precise queries about semantics. Moreover, the OMG standard leaves unspecifed a number of details about the execution semantics of UML 2.0 StateMachines. This underspecification means that users can create a UML semantic variant that suits their modelling needs and yet still complies with the OMG standard.

Template semantics [17] is a template-based approach for structuring the operational semantics of a family of notations, such that semantic concepts that are common among family members (e.g., enabled transitions) are expressed as parameterized mathematical definitions. As a result, the task of specifying a notation's semantics is reduced to providing a collection of parameter values that instantiate the template. And the task of comparing notations' semantics is reduced to comparing their respective template-parameter values.

In this paper, we extend the template-semantics templates and composition operators to support notations that allow queue-based message passing among concurrent objects. We then use the extended template semantics to document concisely the semantics of UML 2.0 StateMachines, as defined in the OMG standard [19]. Related efforts [7, 13, 12, 24] to provide a precise semantics for UML StateMachines refine the standard's semantics, so as to produce a complete, formal semantics that is suitable for automated analysis. In contrast, our template-semantics representation retains the semantics variation points that are documented in the standard.

O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199, pp. 245–259, 2006.

We also express the template semantics for StateMachines as implemented in three UML CASE tools: Rational Rose RT [8], Rhapsody [5, 11], and Bridgepoint [1]. Related efforts [2] to compare UML StateMachine variants mention some semantic distinctions, but they focus more on the differences in syntax and in the language constructs supported. In contrast, our work formally compares the variants' execution semantics. As a side effect, the template-semantic description of a UML model can be used in configurable analysis tools, where the template parameters provide the configurability.

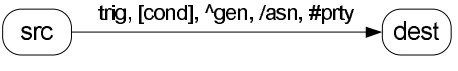
The rest of the paper is organized as follows. Section 2 is a review of template semantics, as used in this paper. Sections 3 and 4 provide the template semantics for the OMG standard for UML 2.0 StateMachines. Section 5 compares this semantics with the template semantics for StateMachines as implemented in three UML CASE tools. The paper concludes with related work and conclusions.

2 Template Semantics

In this section, we review the template semantics and the template parameters that we use to represent UML StateMachine semantics. A more comprehensive description of template semantics can be found in [17, 18].

2.1 Computation Model

Template semantics are defined in terms of a computation model called a *hierarchical transition system (HTS)*. An HTS is an extended StateMachine, adapted from statecharts [6], that includes control states and state hierarchy, state transitions, events, and typed variables, but not concurrency. Concurrency is achieved by composing multiple HTSs. Transitions have the following form:



whose elements are defined in Table 1. Each transition may have one or more source states, may be triggered by zero or more events, and may have a guard condition (a predicate on variable values). If a transition executes, it may lead to one or more destination states, may generate events, and may assign new values to variables. A transition may also have an explicitly defined priority *prty*, which is an integer value. An HTS includes designation of initial states, of default substates for hierarchical states, and of initial variable values.

Table 1. HTS accessor functions from state s or transition τ

Function	Signature	Description
$src(\tau)$	$T \to 2^S$	set of source states of τ
$dest(\tau)$	$T \to 2^S$	set of destination states of τ
$trig(\tau)$	$T \to 2^E$	events that trigger τ
$cond(\tau)$	$T \to exp$	τ's guard condition, where exp is a (predicate) expression over V

Function	Signature	Description
$prty(\tau)$	$T \to \mathbb{N}$	τ's priority value
$ancest(s)$	$S \to 2^S$	ancestor states of s
$gen(\tau)$	$T \to [E]^*$	sequence of events generated by τ
$asn(\tau)$	$T \to [V \times exp]^*$	sequence of variable assignments made by τ

We use helper functions to access static information about an HTS. The functions used in this paper appear in Table 1. In the definitions, S is the HTS's set of control states, T is the set of state transitions, V is the set of variables, and E is the set of events. The notation 2^X refers to the powerset of X; thus, *src* maps a transition τ to its set of source states. The notation $[X]^*$ refers to a sequence of zero or more elements of X.

2.2 Parameterized Execution Semantics

The execution of an HTS is defined in terms of sequences of snapshots. A *snapshot* is data that reflects the current status of the HTS's execution. The basic snapshot elements are

CS - the set of current states
IE - the set of current internally generated events
AV - the current variable-value assignments
O - the set of generated events to be communicated to other HTSs

In addition, the snapshot includes auxiliary elements that store history information about the HTS's execution:

CS_a - data about states, like enabling states or history states
IE_a - data about internal events, like enabling or nonenabling events
I_a - data about inputs I from the StateMachine's environment
AV_a - data about variable values, like old values

The types of information stored in the auxillary elements differ among modelling notations. The expression $ss.X$ (e.g. $ss.CS$) refers to element X in snapshot ss.

An execution of an HTS is a sequence of snapshots, starting from an initial snapshot of initial states and variable values. Template semantics defines a notation's execution semantics in terms of functions and relations on snapshots:

ENABLED_TRANS*(ss,T)* $\subset T$ returns the subset of transitions in T that are enabled in snapshot *ss*.

APPLY*(ss,τ,ss') : bool* holds if applying the effects of transition τ (e.g., variable assignments, generated events) to snapshot *ss* results in next snapshot *ss'*.

N_{MICRO}*(ss,τ,ss') : bool* is a *micro* execution step (a *micro-step*) representing the execution of transition τ, such that τ is enabled in snapshot *ss* and its execution results in next snapshot *ss'*.

RESET*(ss,I) :* ss^r resets snapshot *ss* with inputs I, producing snapshot ss^r.

N_{MACRO}*(ss,I,ss') : bool* is a *macro* execution step (a *macro-step*) comprising a sequence of zero or more micro-steps taken in response to inputs I. The macro-step starts in snapshot RESET*(ss,I)* and ends in snapshot *ss'*, in which the next inputs are sensed.

We provide the definitions of ENABLED_TRANS and APPLY below, as examples of our template definitions. The other definitions can be found in [17].

$$\text{APPLY}(ss, \tau, ss') \equiv$$
$$\text{let } \langle CS', IE', AV', O', CS'_a, IE'_a, AV'_a, I'_a \rangle \equiv ss' \text{ in}$$
$$\mathbf{next\_CS}(ss, \tau, CS') \wedge \mathbf{next\_CS}_a(ss, \tau, CS'_a) \wedge \mathbf{next\_IE}(ss, \tau, IE') \wedge \mathbf{next\_IE}_a(ss, \tau, IE'_a) \wedge$$
$$\mathbf{next\_AV}(ss, \tau, AV') \wedge \mathbf{next\_AV}_a(ss, \tau, AV'_a) \wedge \mathbf{next\_O}(ss, \tau, O') \wedge \mathbf{next\_I}_a(ss, \tau, I'_a)$$
$$\text{ENABLED\_TRANS}(ss, T) \equiv$$
$$\{\tau \in T \mid \mathbf{en\_states}(ss, \tau) \wedge \mathbf{en\_events}(ss, \tau) \wedge \mathbf{en\_cond}(ss, \tau)\}$$

Table 2. Template parameters provided by users

	States	Events	Variables	Outputs
Beginning of Macro-step	**reset_CS**(ss, I): CS^r **reset_CS**$_a(ss, I)$: CS^r_a	**reset_IE**(ss, I):IE^r **reset_IE**$_a(ss, I)$:IE^r_a **reset_I**$_a(ss, I)$:I^r_a	**reset_AV**(ss, I) : AV^r **reset_AV**$_a(ss, I)$: AV^r_a	**reset_O**(ss, I): O^r
Micro-Step	**next_CS**(ss, τ, CS') **next_CS**$_a(ss, \tau, CS'_a)$	**next_IE**(ss, τ, IE') **next_IE**$_a(ss, \tau, IE'_a)$ **next_I**$_a(ss, \tau, I'_a)$	**next_AV**(ss, τ, AV') **next_AV**$_a(ss, \tau, AV'_a)$	**next_O**(ss, τ, O')
Enabledness	**en_states**(ss, τ)	**en_events**(ss, τ)	**en_cond**(ss, τ)	
Others	**macro_semantics** **pri**$(T) : 2^T$			

The SMALL-CAPS FONT denotes a template definition, and **bold font** denotes a template parameter. Thus, definition APPLY uses template parameters **next_X**, each of which specifies how a single snapshot element, X, is updated to reflect the effects of executing transition τ. And definition ENABLE_TRANS uses template parameters to determine whether a transition's source states, triggering events, and guard conditions are enabled in snapshot ss.

The template parameters are listed in Table 2. Functions **reset_X**(ss, I) specify how inputs I are incorporated into each snapshot element $ss.X$ at the start of a macro-step, returning new value X'. Predicates **next_X**(ss, τ, X') specify how the contents of each snapshot element $ss.X$ is updated to new value X', due to the execution of transition τ. Parameters **en_states, en_events,** and **en_cond** specify how the state-, event-, and variable-related snapshot elements are used to determine the set of enabled transitions. Parameter **macro_semantics** specifies the type of HTS-level macro-step semantics (e.g., when new inputs are sensed). Parameter **pri** specifies a priority scheme over a set of transitions. Each of the 21 parameters[1] represents a distinct semantics decision, although the parameters associated with the same construct are often related.

2.3 Composition Operators

So far, we have discussed the execution of a single HTS. Composition operators specify how multiple HTSs execute concurrently, in terms of how the HTSs' snapshots are collectively updated.

A *Composed HTS (CHTS)* is the composition of two or more operands via some composition operator *op*. The operands may be HTSs or may themselves be composed HTSs. The snapshot of a CHTS is the collection of its HTSs' snapshots, and is denoted using vector notation, $\overrightarrow{ss}$. Template definitions and access functions are generalized to apply to collections of snapshots. Thus, ENABLED_TRANS$(\overrightarrow{ss}, T)$ returns all transitions in T that are enabled in any snapshot in $\overrightarrow{ss}$.

A micro-step for a CHTS that composes operands N_1 and N_2 via operation *op* has the general form

$$N^{\text{OP}}_{\text{MICRO}}((\overrightarrow{ss_1}, \overrightarrow{ss_2}), (\overrightarrow{\tau_1}, \overrightarrow{\tau_2}), (\overrightarrow{ss_1}', \overrightarrow{ss_2}'))$$

[1] Template semantics has a 22nd parameter, **resolve**, that specifies how to resolve concurrent assignments to shared variables. This parameter is not used in this paper.

$$
\begin{array}{l}
\mathrm{N}^{\mathrm{INTERR}}_{\mathrm{MICRO}}((\overrightarrow{ss_1}, \overrightarrow{ss_2}),\ (\overrightarrow{r_1}, \overrightarrow{r_2}),\ (\overrightarrow{ss_1}', \overrightarrow{ss_2}'))\ T_{interr} \equiv \\
\quad \left[\begin{array}{l} \overrightarrow{r_1} \subset \overrightarrow{T_1}\ \wedge\ \overrightarrow{r_1} \subset \mathbf{pri}(\mathrm{ENABLED\_TRANS}(\overrightarrow{ss_1}, \overrightarrow{T_1} \cup T_{interr})) \\ \wedge\ \mathrm{N}^1_{\mathrm{MICRO}}(\overrightarrow{ss_1}, \overrightarrow{r_1}, \overrightarrow{ss_1}')\ \wedge\ \overrightarrow{ss_2}' = \overrightarrow{ss_2}|^{AV}_{assign(\overrightarrow{ss_2}.AV,\ \overrightarrow{ss_1}'.AV)} \end{array} \right] \quad (* \text{ component 1 takes a step } *) \\
\\
\vee \\
\quad \exists \overrightarrow{iss}. \left[\begin{array}{l} \overrightarrow{r_1} \in T_{interr}\ \wedge\ \overrightarrow{r_1} \in \mathbf{pri}(\mathrm{ENABLED\_TRANS}(\overrightarrow{ss_1},\ \overrightarrow{T_1} \cup T_{interr})) \\ \wedge\ \mathrm{APPLY}(\overrightarrow{ss_2}, \overrightarrow{r_1}, \overrightarrow{iss})\ \wedge\ \overrightarrow{ss_2}' = \overrightarrow{iss}|^{CS}_{ent\_comp(\overrightarrow{ss_2}, \overrightarrow{r_1})} \\ \wedge\ \overrightarrow{ss_1}' = \overrightarrow{ss_1}|^{CS}_{\emptyset}|^{AV}_{assign(\overrightarrow{ss_1}.AV,\ \overrightarrow{ss_2}'.AV)} \end{array} \right] \quad (* \text{ transition to component 2 } *) \\
\\
\vee \\
\quad (* \text{ symmetric cases of the above two cases, replacing 1 with 2 and 2 with 1 } *)
\end{array}
$$

Fig. 1. Micro-step for CHTS with interrupt operator

where operand N_1 starts the micro-step in snapshots $\overrightarrow{ss_1}$, executes transitions $\overrightarrow{r_1}$ (at most one transition per HTS), and ends the micro-step in snapshots $\overrightarrow{ss_1}'$. Operand N_2 executes in a similar manner, in the same micro-step.

What differentiates one composition operator from another are the conditions under which it allows, or forces, its two operands to take a step. For example, a composition operator may force its two operands to execute concurrently in lock step, may allow its operands to execute nondeterministically, or may coordinate the transfer of a single thread of control from one operand to the other. Operators also differ in the assignments they make to their components' snapshots. For example, a composition operator may affect message passing by inserting each operand's set of generated events into the other operand's event pool.

We use substitution notation to specify an operator-imposed override on snapshot contents. Expression $ss|^x_v$ is equal to snapshot ss, except for element x, which has value v. Substitution over a collection of snapshots denotes substitutions to all of the snapshots. For example, substitution $\overrightarrow{ss}\,|^{CS}_{\emptyset}$ is equal to snapshots $\overrightarrow{ss}$, except that all of the snapshots' CS elements are empty.

Interleaving. Composition operator *interleaving*, defined by template $\mathrm{N}^{\mathrm{INTL}}_{\mathrm{MICRO}}$, specifies that one but not both of its operands executes in a micro-step:

$$
\begin{array}{l}
\mathrm{N}^{\mathrm{INTL}}_{\mathrm{MICRO}}((\overrightarrow{ss_1}, \overrightarrow{ss_2}),\ (\overrightarrow{r_1}, \overrightarrow{r_2}),\ (\overrightarrow{ss_1}', \overrightarrow{ss_2}')) \equiv \\
\vee\ N^1_{micro}(\overrightarrow{ss_1}, \overrightarrow{r_1}, \overrightarrow{ss_1}')\ \wedge\ \overrightarrow{ss_2}' = \overrightarrow{ss_2}|^{AV}_{assign(\overrightarrow{ss_2}.AV,\ \overrightarrow{ss_1}'.AV)} \\
\quad\ N^2_{micro}(\overrightarrow{ss_2}, \overrightarrow{r_2}, \overrightarrow{ss_2}')\ \wedge\ \overrightarrow{ss_1}' = \overrightarrow{ss_1}|^{AV}_{assign(\overrightarrow{ss_1}.AV,\ \overrightarrow{ss_2}'.AV)}
\end{array}
$$

In each micro-step, exactly one of the CHTS's operands takes a micro-step. The snapshot of the non-executing operand is overridden, to update its variable values to reflect the executing transitions' assignments to shared variables. (The macro $assign(X, Y)$ updates variable-value mappings in X with variable-value mappings in Y, ignoring mappings for variables in Y that are not in X.)

Interrupt. *Interrupt* composition, shown in Figure 1, specifies how control is passed between an CHTS's two operands, via a provided set of *interrupt transitions*, T_{interr}. In each micro-step, the operand that has control either takes a micro-step or transfers control to the other operand. The first bracketed clause in Figure 1 shows operand N_1 taking a micro-step:

Table 3. Mapping UML syntax to HTS syntax

UML	Template Semantics
simple state	$s \in S$
event	$e \in E$
simple attribute	$v \in V$
state variable	$v \in V$
pseudostate (except fork and join)	$s \in S$
simple transition	$\tau \in T$

UML	Template Semantics
transition segment (except fork and join)	$\tau \in T$
maximal composite state with no orthogonal substate	HTS
orthogonal composite state	CHTS (interleaving)
nonorthogonal composite state with orthogonal substates	CHTS (interrupt)
fork, join transitions	interrupt transitions

- Transitions $\overrightarrow{\tau_1}$ in operand N_1 have the highest priority (according to template parameter **pri**) among enabled transitions, including interrupt transitions.
- Operand N_1 takes a micro-step.
- The snapshot of N_2 is updated to reflect assignments to shared variables.

The second bracketed clause shows a transition from operand N_1 to operand N_2:

- Interrupt transition $\overrightarrow{\tau_1}$ has the highest priority among all enabled transitions.
- N_2's snapshots $\overrightarrow{ss_2}$ are updated by (1) applying the effects of the interrupt transition $\overrightarrow{\tau_1}$ and (2) overriding their CS elements with the sets of states entered by the interrupt transition (as determined by macro *ent_comp*).
- N_1's snapshots $\overrightarrow{ss_1}$ are updated by (1) emptying their CS elements (since this operand no longer has control) and (2) updating their variable-value elements AV to reflect $\overrightarrow{\tau_1}$'s assignments to shared variables.

The cases in which operand N_2 has control are symmetric.

3 Syntactic Mapping from UML to HTS

The first step in defining a template semantics for UML is to map UML modelling constructs to our computational model, the HTS. This is essentially a mapping from UML syntax to HTS syntax, and is summarized in Table 3. Most of the mappings are straightforward: Simple states, events, variables, and simple transitions in UML have corresponding constructs in HTS syntax. Pseudostates in UML (except for fork and join) are mapped to simple states in HTS syntax, and the pseudostates' transition segments are mapped to transitions in HTS syntax. As will be seen in the next section, a UML compound transition maps to a sequence of HTS transitions that executes over several micro-steps.

Recall that an HTS is a state machine with no internal concurrency, and that concurrency is introduced by composition operators. Thus, each highest-level (maximal) nonorthogonal composite state that contains no orthogonal descendant states is mapped to an HTS. A UML orthogonal state is mapped to a CHTS whose operands are the orthogonal regions and whose operator is *interleaving* composition. And a nonorthogonal composite state that has one or more orthogonal descendant states is mapped to a CHTS, whose operands are the state's child substates, whose composition operator is *interrupt*, and whose

interrupt transitions transfer control between the operands. Fork and join transitions in UML, which enter or exit multiple regions of an orthogonal state, map to interrupt transitions that enter or exit interleaved CHTSs (see Section 4.2).

We treat state entry/exit actions and submachines as *syntactic macros* that are expanded by a preprocessor into transition actions and complete StateMachines, respectively. Entry and exit pseudostates can be treated similarly if the action language evaluates actions sequentially. To simplify our presentation, we do not consider history states in this paper, but they can be handled [17]. State activities and operations can be supported if their effects can be represented as generated events and variable assignments. We have not attempted a template semantics for object creation and termination and do not describe TimeEvents in the current work.

4 Semantics of OMG UML

In this section, we describe the execution semantics of a UML StateMachine, in terms of its corresponding HTS's template parameters and composition operators. In what follows, we use OMG-UML to refer to the semantics of UML as defined by the OMG [19]. In addition, we assume that a StateMachine describes the behaviour of a UML object, and we use these two terms interchangeably.

4.1 Template Parameters

The template-parameter values for OMG-UML are listed in the second column of Table 4. To the right of each entry (i.e., the corresponding entry in the third column) are the page numbers in the OML-UML documentation [19] that contain the textual description of semantics that we used in formulating that entry's value. Unused parameters, IE and IE_a are omitted from the table.

State-Related Parameters. Rows 1-5 in Table 4 pertain to the semantics of states. We use snapshot element CS to record the set of current states. This set does not change at the start of a macro-step (i.e., **reset_CS** does not modify CS). When a transition τ executes, element CS is updated by template parameter **next_CS** to hold the states that are current, or that become current, whenever τ's destination state is entered, including the destination state's ancestors and all relevant descendants' default states.

We use snapshot element CS_a to record the states that can enable transitions ($en\_states = (src(\tau) \subseteq CS_a)$). In OMG-UML, only one compound transition can execute per macro-step. To model this semantics, CS_a is set to $dest(\tau)$ if the destination state is a pseudostate, so that only the rest of the compound transition may continue executing; otherwise, CS_a is set to $\emptyset$, thereby ending the macro-step.

Event-Related Parameters. Rows 6-10 in Table 4 pertain to event semantics. We use snapshot element I_a to hold the event that an HTS is currently processing. At the start of a macro-step, an event I from the event pool is input to the

Table 4. Template Parameter Values for Multiple UML Notations

	Parameter	OMG-UML	[19] Page#	RRT-UML	[9] Page#	RH-UML	[5] Page#	BP-UML	[23] Page#	
States	$reset\_CS(ss, I) =$	$ss.CS$	-	$ss.CS$	-	$ss.CS$	-	$ss.CS$	-	1
	$next\_CS(ss, \tau, CS')$	$CS' = active(dest(\tau))$	531	$CS' = active(dest(\tau))$	52	$CS' = active(dest(\tau))$	25	$CS' = dest(\tau)$	50, 101	2
	$reset\_CS_a(ss, I) =$	$ss.CS$	-	$ss.CS$	-	$ss.CS$	-	$ss.CS$	-	3
	$next\_CS_a(ss, \tau, CS'_a)$	**if** $pseudo(dest(\tau))$ **then** $CS'_a = dest(\tau)$ **else** $CS'_a = \emptyset$	523, 535, 547	**if** $pseudo(dest(\tau))$ **then** $CS'_a = dest(\tau)$ **else** $CS'_a = \emptyset$	52, 60	$CS'_a = active(dest(\tau))$	3, 5, 26	$\emptyset$	50	4
	$en\_states(ss, \tau)$	$src(\tau) \subseteq ss.CS_a$	556	$src(\tau) \subseteq ss.CS_a$	52	$src(\tau) \subseteq ss.CS_a$	25	$src(\tau) \subseteq ss.CS_a$	50	5
Events	$reset\_I_a(ss, I) =$	I	546	I	54	I	25	I	103	6
	$next\_I_a(ss, \tau, I'_a)$	$I'_a = \emptyset$	546	$I'_a = \emptyset$	54, 49	$I'_a = \emptyset$	26	$I'_a = \emptyset$	47, 107	7
	$en\_events(ss, \tau)$	$ss.I_a \subseteq trig(\tau)$	556	$ss.I_a \subseteq trig(\tau)$	52	$trig(\tau) = ss.I_a$	3, 25	$trig(\tau) = ss.I_a$	50	8
	$reset\_O(ss, I) =$	$\emptyset$	-	$\emptyset$	-	$\emptyset$	-	$\emptyset$	-	9
	$next\_O(ss, \tau, O')$	$O' = gen(\tau)$	557	$O' = gen(\tau)$	48, 49	$O' = gen(\tau)$	6	$O' = gen(\tau)$	47, 107	10
Variables	$reset\_AV(ss, I) =$	$ss.AV$	-	$ss.AV$	-	$ss.AV$	-	$ss.AV$	-	11
	$next\_AV(ss, \tau, AV')$	$AV' = ss.AV \oplus_? asn(\tau)$	557	$AV' = assign(ss.AV, seq\_eval(ss.AV, asn(\tau)))$	47	$AV' = assign(ss.AV, seq\_eval(ss.AV, asn(\tau)))$	7, 25	$AV' = assign(ss.AV, seq\_eval(ss.AV, asn(\tau)))$	45, 111	12
	$reset\_AV_a =$	$ss.AV$	-	$ss.AV$	-	$ss.AV$	-	N/A	-	13
	$next\_AV_a(ss, \tau, AV'_a)$	**if** $choice(dest(\tau))$ **then** $AV'_a = (ss.AV \oplus_? asn(\tau))$ **else** $AV'_a = ss.AV_a$	523	**if** $choice(dest(\tau))$ **then** $AV'_a = assign(ss.AV, seq\_eval(ss.AV, asn(\tau)))$ **else** $AV'_a = ss.AV_a$	47, 60	$AV'_a = ss.AV_a$	25	N/A	-	14
	$en\_cond(ss, \tau)$	$ss.AV_a \models cond(\tau)$	556	$ss.AV_a \models cond(\tau)$	52	$ss.AV_a \models cond(\tau)$	25	$TRUE$	-	15
	$macro\_semantics$	stable	546	stable	54, 57	stable	24	simple	50, 103	16
	pri	$pri(\Gamma) \equiv \{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(src(\tau)) \geq rank(src(t))\}$	547	$pri(\Gamma) \equiv \{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(src(\tau)) \geq rank(src(t))\}$	62	$pri(\Gamma) \equiv \{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(src(\tau)) \geq rank(src(t))\}$	22	N/A	-	17
	Composition	INTERR, INTL, OBJECT, MULTI-OBJECT	523, 535, 547, 555	OBJECT, MULTI-OBJECT	50, 82 83	INTERR, INTL, OBJECT, MULTI-OBJECT	14, 24 26	OBJECT, MULTI-OBJECT	104, 107	18

Key

	Semantics that refine a semantic variation point in the OMG standard
	Semantics that deviate from the OMG standard
$active(s)$	States that are active when state s becomes active, including s's ancestors and relevant descendants' default states
$pseudo(s)$	Returns true if state s is a choice, junction or initial pseudostate
$choice(s)$	Returns true if state s is a choice point.
$assign(X, Y)$	Updates assignments X with the assignments Y, and ignores assignments in Y to variables that are not in X
$seq\_eval(X, A)$	*Sequentially* evaluates assignment expressions in A, starting with variable values in X and updating these as assignments are processed; returns updated variable-value assignments.
$pri(\Gamma)$	Returns the subset of transitions Γ that have highest priority.
$rank(s)$	Distance of state s from the root state. $rank(root) = 0$. $rank(S)$ returns the rank of the state with the highest rank within set S.

HTS and saved in I_a. A transition is enabled only if one of its triggers matches this event ($ss.I_a \subseteq trig(\tau)$). I_a is set to $\emptyset$ after the first transition executes; but subsequent segments of a compound transitions may still be enabled, since they have no triggers.

We use snapshot element O to hold an HTS's outputs, which are the events generated by the HTS's executing transition. These events are output in the same micro-step in which they are generated. Thus, element O need only record the events generated by the most recent transition.

Variable-Related Parameters. Rows 11-15 in Table 4 pertain to the semantics of variables. We use snapshot element AV to record the current values of variables. A transition may perform multiple variable assignments, and may even perform multiple assignments to the same variable. OMG-UML [19] does not pin down the action language, so the semantics of variable assignments, especially with respect to evaluation order or execution subset, is a semantic variation point. We use the symbol $\oplus_?$ to indicate that some of the assignments in τ's actions have an overriding effect on the variable values in AV, but that the exact semantics of this effect is left open.

We use auxiliary snapshot element AV_a to record the variable values that are used when evaluating transition guards ($ss.AV_a \models cond(\tau)$) and assignment expressions. In OMG-UML semantics, transition guards are evaluted with respect to variables' values at the start of a macro-step – unless the transition is exiting a choice pseudostate. Thus, AV_a records the variables' current values at the start of a macro-step, and is not updated during the macro-step unless an executing transition enters a choice pseudostate.

Macro-Semantics and Priority Parameters. OMG-UML has *stable* macro-step semantics, meaning that an HTS processes an event to completion before inputing the next event. With respect to priority among transitions, transitions whose source states have the highest rank (i.e., are deepest in the state hierarchy) have highest priority. Thus, substate behaviour overrides super-state behaviour. The priority of a join transition is the priority of its highest-ranked segment.

4.2 Composition Operators

We use composition operators to compose HTSs into CHTSs that represent UML StateMachines and collections of communicating StateMachines. We use *interleaving* and *interrupt* operators for intra-object composition, to create orthogonal and composite states, respectively. We also introduce two inter-object composition operators that define the behaviour of object-level composition: *object composition* defines how a single object takes a micro-step with respect to UML's run-to-completion step semantics, and *multi-object composition* defines how multiple objects execute concurrently and communicate via directed events.

Interleaving Composition. We use *interleaving composition*, defined in Section 2.3, to model orthogonal composite states. According to OMG-UML semantics [19], each orthogonal region executes at most one compound transition

per run-to-completion step, and the order in which the regions' transitions, or transition segments, execute is not defined. This behaviour is captured by the micro-step interleaving operator, which allows fine-grained interleaving of HTSs and their transitions. The order in which the interleaved transitions' generated events or variable assignments occur is nondeterministic.

Interrupt Composition. We use *interrupt composition*, defined in Figure 1, to model nonorthogonal composite states that contain orthogonal substates. The semantics of execution is as described in Section 2.3: Only one of the composite state's direct substates is ever active; and in each micro-step, either the active substate executes internal transitions, or one of the interrupt transitions executes and transfers control from the active substate to another substate.

In a typical case, interrupt composition models fork and join transitions that enter or exit, respectively, an orthogonal composite state. We model forks and joins as single HTS transitions that have multiple destination states (forks) or multiple source states (joins). If a fork does not specify a destination state in one of the orthogonal regions, the macro *ent_comp* in the interrupt operator determines the region's implicit (default) destination states.

Object Composition. In OMG-UML, each active object is modelled as a StateMachine with its own event pool and thread of control[2]. An object executes by performing *run-to-completion steps*, defined as follows:

1. An event is removed from the object's event pool for the object to process.
2. A maximal set of non-conflicting enabled transitions are executed. Conflicts are resolved using priorities (reflected in template parameter **pri**)
3. The events generated by these transitions are sent to the targeted objects.
4. Steps 2 and 3 are repeated, until no more transitions are enabled.

The *object composition* operator, shown in Figure 2, defines an allowable micro-step taken by an object. Macro $stable(\overrightarrow{ss})$ determines whether a run-to-completion step has ended, meaning that no transitions are enabled. If so, then a new event e is selected from the object's event pool and is incorporated into the object's snapshots ($\textsc{reset}(\overrightarrow{ss}, e)$). To effect a micro-step, the operator invokes the micro-step operator for the object's top-most hierarchical state: $\mathrm{N}^{\mathrm{HTS}}_{\mathrm{MICRO}}$, if the state represents an HTS; $\mathrm{N}^{\mathrm{INTL}}_{\mathrm{MICRO}}$, if the state is an interleaved CHTS; or $\mathrm{N}^{\mathrm{INTERR}}_{\mathrm{MICRO}}$, if the state is an interrupt CHTS.

In OMG-UML, the order in which events are removed or added to an event pool is purposely left undefined. To model this semantics variation, we introduce new template parameters **reset_Q** and **next_Q** to specify how an event pool Q is updated with inputs from the environment or with events sent by other objects, respectively; parameter **pick** specifies how an event is selected from an event pool. The second column of Table 5 presents the parameter values for OMG-UML. We use symbols $+_?$ and $-_?$, to represent OMG-UML's undefined semantics for adding and removing events from an event pool. In addition, we

[2] UML also has the notion of a *passive object*, which contains data only and which executes only when an active object invokes one of its methods.

$$
\begin{array}{l}
\mathrm{N}^{\text{OBJECT}}_{\text{MICRO}}(\overline{ss}\,, \overrightarrow{T}\,, \overline{ss}\,')(Q, Q') \equiv \\
\quad \text{if } \mathit{stable}(\overline{ss}\,) \text{ then} \\
\qquad \exists\, \overline{ss}\,^{r}, e\,. \left[\begin{array}{l} \mathbf{pick}(\overline{ss}\,, Q, e, Q') \ \wedge\ \overline{ss}\,^{r} = \text{RESET}(\overline{ss}\,, e) \ \wedge \\ (\mathrm{N}_{\text{MICRO}}(\overline{ss}\,^{r}, \overrightarrow{T}\,, \overline{ss}\,') \ \vee\ (\mathit{stable}(\overline{ss}\,^{r}) \wedge \overline{ss}\,^{r} = \overline{ss}\,')) \end{array} \right] \\
\quad \text{else} \\
\qquad \mathrm{N}_{\text{MICRO}}(\overline{ss}\,, \overrightarrow{T}\,, \overline{ss}\,') \wedge Q = Q' \\
\\
\mathrm{N}^{\text{MULTI-OBJECT}}_{\text{MACRO}}((\overline{ss}_1, ..., \overline{ss}_n), I, (\overline{ss}'_1, ..., \overline{ss}'_n))(Q_1 ... Q_n, Q'_1 ... Q'_n) \equiv \\
\quad \exists\, k,\ \overrightarrow{T},\ Q''_k,\ Q^r_1, ..., Q^r_n\,. \\
\qquad \left[\begin{array}{l} \forall i\,.\, 1 \le i \le n\,.\, Q^r_i = \mathbf{reset\_Q}(Q_i, \mathit{directed\_events}(I, i)) \ \wedge \\ 1 \le k \le n \ \wedge\ \mathrm{N}^{\text{OBJECT}}_{\text{MICRO}}(\overline{ss}\,_k, \overrightarrow{T}\,, \overline{ss}\,'_k)(Q^r_k, Q''_k) \ \wedge \\ \forall i\,.\, 1 \le i \le n\,.\, ((i = k \rightarrow \mathbf{next\_Q}(Q''_k, \mathit{directed\_events}(\overline{ss}\,'_k.O, k), Q'_k) \wedge \\ \qquad\qquad (i \ne k \rightarrow \mathbf{next\_Q}(Q^r_i, \mathit{directed\_events}(\overline{ss}\,'_k.O, i), Q'_i)) \end{array} \right]
\end{array}
$$

Fig. 2. Multi-object and object composition

Table 5. Event-Pool Related Template-Parameter Values

	OMG-UML [19] pg. 546	RRT-UML [10] pg. 79	RH-UML [5] pg. 25	BP-UML [23] pg. 107
$pick(\overline{ss}, Q, e, Q')$	$ready\_ev(\overline{ss}, Q, e) \wedge$ $Q' = Q -_? e$	$e = top(Q) \wedge$ $Q' = pop(Q)$	$e = top(Q) \wedge$ $Q' = pop(Q)$	$e = top(Q) \wedge$ $Q' = pop(Q)$
$reset\_Q(Q, I) =$	$Q +_? I$	$append(Q, I)$	$append(Q, I)$	$append(Q, I)$
$next\_Q(Q, I, Q')$	$Q' = Q +_? I$	$Q' = append(Q, I)$	$Q' = append(Q, I)$	$Q' = append(Q, I)$

Key

	Semantics that refine a semantic variation point in the OMG standard
	Semantics that deviate from the OMG standard
$ready\_ev(\overline{ss}, Q, e)$	Select e such that $deferred(e, \overline{ss}.CS) = \emptyset \vee$ $(rank(deferred(e, \overline{ss}.CS)) \le rank(src(\text{EN\_TRANS}(\text{RESET}(\overline{ss}, e), \overrightarrow{T}))))$
$deferred(e, S)$	Returns the subset of the states S in which event e is deferred
$X +_? Y$	Undefined operator for adding element Y to container X
$X -_? Y$	Undefined operator for removing element Y from container X
$append(Q, I)$	Appends the event sequence I to the end of Q, and returns the resulting queue
$top(Q)$	Returns the front element of queue Q
$pop(Q)$	Removes the front element from Q, and returns the resulting queue

use macro $ready\_ev(\overline{ss}\,, Q, e)$ to help represent deferred events: it returns an event e that either (1) is not deferred in any current state or (2) triggers a transition whose source state has higher priority than the state(s) that defer e.

Multi-object Composition. *Multi-object composition*, shown in Figure 2, models the concurrent execution of n objects. It is a UML model's top-most composition operator, and thus defines how input events I (e.g., user inputs) and inter-object messages are handled. In each macro-step, (1) the inputs I are added to the appropriate objects' event pools, (2) some object is nondeterministically chosen to execute a micro-step, and (3) the events generated in that micro-step are added to the target objects' event pools. Macro *directed_events* filters events by their target object, returning only the events destined for that object.

5 Semantics of UML Tools

In this section, we present template-semantics descriptions for StateMachines as implemented in three UML CASE tools: Rational Rose RealTime [8]

(RRT-UML), Rhapsody [5, 11](RH-UML) and BridgePoint [1, 23](BP-UML). We then evaluate how well each tool complies with UML 2.0 semantics by comparing how well its template-parameter values match those for OMG-UML 2.0, which were presented in the last section.

The template-parameter values for the three UML CASE tools are given in Table 4, in columns 4, 6, and 8. The reference that we used in determining each parameter value is given in the table entry to the right of the parameter value.

State-Related Parameters. RRT-UML's state semantics match exactly those of OMG-UML. In RH-UML, the set of enabling states, CS_a, is always equal to the current set of states, CS. Thus, an HTS may execute multiple transitions in a macro-step, but only the first transition can have a trigger. An HTS can even get into an infinite loop if the states and variable values always enable a next transition. In BP-UML, an HTS never executes more than one transition in a macro-step, so CS_a is always empty after the first transition executes.

Event-Related Parameters. All three UML variants have similar event semantics: an input event can trigger only the first transition of a macro-step, and generated events are output (to the target objects' event pools). The only difference is that, in OMG-UML and RRT-UML, a transition may have multiple triggers ($ss.I_a \subseteq trig(\tau)$), whereas in RH-UML and BP-UML, a transition may have only one trigger ($trig(\tau) = ss.I_a$).

Variable-Related Parameters. OMG-UML does not specify how variable values are updated due to transitions' assignments. RRT-UML, RH-UML, and BP-UML all refine OMG-UML's semantics in the same way: variable values are updated in the order, left to right, in which they appear in the transition label.

In RRT-UML and RH-UML, (non-choice-point) transition guards and assignment expressions are always evaluated with respect to variable values from the start of the macro-step ($ss.AV_a \models cond$). In contrast, RH-UML does not support dynamic choice points, so its $\mathbf{next\_AV_a}$ variable values are never updated in the middle of a macro-step. BP-UML does not support guard conditions, so its predicate *en_cond* is always *true*.

Macro-Semantics and Priority Parameters. RRT-UML and RH-UML have *stable* macro-semantics, to support compound transitions (in RRT-UML) or to allow an HTS to execute multiple transitions in a macro-step (in RH-UML). In contrast, BP-UML does not support compound transitions, and its semantics allow an HTS to execute at most one transition per macro-step, so BP-UML has *simple* macro-semantics. RRT-UML and RH-UML use the same transition priority scheme as OMG-UML uses. BP-UML has no priority scheme.

Composition Operators. Neither RRT-UML nor BP-UML support orthogonal composite states. Thus, a StateMachine in these notations maps to an HTS and no intra-object composition is needed. RH-UML supports orthogonal composite states, as well as join and fork pseudostates. Moreover, the order in which orthogonal regions execute, and thus the order in which their transitions' actions

take effect, is nondeterministic [5]. As a result, the *interleaving* and *interrupt* composition operators defined in Sections 2.3 apply also to RH-UML.

All three UML variants use the *object* and *multi-object* composition operators; their template-parameter values for these operators appear in columns 3-5 of Table 5. All three variants implement event pools as FIFO queues to ensure that the order of events, as generated by a transition or as sensed by the environment, is preserved during message passing. And they all deviate from OMG-UML semantics by not supporting deferred events. In RRT-UML and RH-UML, several objects may share a thread of control and an event pool for efficiency reasons [4, 15, 21], but this has no effect on the semantics of execution.

6 Evaluation

Our template-semantics description of UML 2.0 is based on OMG documents [19], supplemented by questions sent to the "Ask an Expert" facility on the OMG Website. For RRT-UML, we used the Modeling Language Guide [9], our experiences with Rational Rose RT [8], and e-mail correspondence with Bran Selic [21]. For RH-UML, we used conference papers [5], our experiences with Rhapsody [11], and e-mail correspondence with David Harel [4]. For BP-UML, we used Shlaer and Mellor's text [23], our experiences with Nucleus Bridgepoint [1], and e-mail correspondence with Campbell McCausland [15].

Because these sources are written in a combination of natural language, pseudocode, and examples, it is impossible for us to formally prove that our template-semantics descriptions accurately represent the documented semantics. Instead, we trace each of our template-semantics' parameter values to statements in these sources. We include this traceability information in Tables 4 and 5.

7 Related Work

There has been extensive work to formalize the semantics of statecharts [6, 20, 16] and to compare different semantics [22, 14]. Shankar et al. [22] describe a two-dimensional temporal logic that could be used to describe semantic variations of statecharts. Maggiolo-Schettini et al. [14] use structural operational semantics and labeled transition systems to describe the semantics of two statechart variants. In both cases, it could be argued that it would be somewhat harder to use their logics to compare statecharts variants, because it would mean comparing collections of free-form axioms rather than collections of specific template parameters.

There have been several attempts at making the semantics of UML StateMachines more precise [7, 12, 13, 24], usually to enable automated analysis. Fecher et al. [3] outline 29 new unclarities in the semantics of UML 2.0 and provide informal pointers as how to eliminate those ambiguities. To our knowledge, there has not been any other attempt to formally define and compare the semantics of different UML StateMachine variants.

Crane and Dingel [2] informally compare Rhapsody StateMachines against the UML standard. Most of their results relate to syntax, language constructs, and well-formedness constraints rather than the semantics of execution. In particular, little discussion is devoted to crucial aspects of the semantics, such as orthogonal composite states, composition operators, and event pools. Also, the differences are described using natural language, which makes an exact definition and comparison very difficult. In contrast, our work focuses on execution semantics; we use a formalism that highlights semantics variation points; and our work takes into consideration composition, concurrency, and event pools.

8 Conclusions

The contributions of this work are threefold. First, we add event-pool-related template parameters to template semantics, to model message passing between components. Second, we provide a template-semantics representation of the execution semantics of UML StateMachines, as defined by the OMG. Unlike similar work, our approach does not result in a more precise semantics of UML, but rather it results in a formal and concise description of UML semantics that highlights the semantics variation points in the standard. Third, we provide template-semantics representations for StateMachines as implemented in three UML CASE tools, showing precisely how these tools address unspecified semantics in the standard and how they deviate from specified semantics in the standard.

One of our future goals is a more comprehensive comparison of UML StateMachine variants and traditional statecharts variants, in the form of a formal version of von der Beeck's informal comparison of statechart variants [25]. In addition, we are investigating the potential of automatically analyzing UML models using tools that are semantically configured by template-parameter values.

Acknowledgments

We thank Bran Selic from IBM, David Harel from the Weizman Institute, and Campbell McCausland and Stephen Mellor from Accelerated Technology for helping us to understand the semantics details of UML StateMachines and of their respective tools.

References

1. Accelerated Technology. Bridgepoint. www.acceleratedtechnology.com/, 2005.
2. M. Crane and J. Dingel. UML vs. Classical vs. Rhapsody State machines: Not All Models are Created Equal. In *Proc. 8th Int. Conf. on Model Driven Eng. Lang. and Sys. (MoDELS/UML 2005)*, Montego Bay, Jamaica, Oct. 2005.
3. H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 New Unclarities in the Semantics of UML 2.0 State Machines. In *ICFEM 2005*, volume 3785, pages 52–65. Springer-Verlag, 2005.
4. D. Harel. Email disucssion. Email, July 2005.

5. D. Harel and H. Kugler. The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML). In *Integration of Software Specification Techniques for Appl. in Eng.*, volume 3147 of *LNCS*, pages 325–354. Springer-Verlag, 2004.
6. D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the Formal Semantics of State machines. In *Logic in Comp. Sci.*, pages 54–64. IEEE Press, 1987.
7. Z. Hu and S. M. Shatz. Explicit Modeling of Semantics Associated with Composite States in UML State machines. *Intl. Jour. of Auto. Soft. Eng.*, 2005.
8. IBM Rational. Rational Rose RealTime. http://www.ibm.com/rational, 2002.
9. IBM Rational. Rational Rose RealTime - Modeling Language Guide, Version 2003.06.00. http://www.ibm.com/rational, 2002.
10. IBM Rational. Rational Rose RealTime - UML Services Library, Version 2003.06.00. http://www.ibm.com/rational, 2002.
11. ilogix, Inc. Rhapsody. http://www.ilogix.com, 2005.
12. Y. Jin, R. Esser, and J. W. Janneck. Describing the Syntax and Semantics of UML State machines in a Heterogeneous Modelling Environment. In *Proc. 2nd Int. Conf. on Diag. Repr. and Infer. (DIAGRAMS '02)*, pages 320–334, London, UK, 2002. Springer-Verlag.
13. J. Jürjens. A UML State Machines Semantics with Message-passing. In *Proc. ACM Symp. on App. Comp.(SAC '02)*, pages 1009–1013, 2002.
14. A. Maggiolo-Schettini, A. Peron, and S. Tini. A comparison of statecharts step semantics. *Theor. Comput. Sci.*, 290:465–498, 2003.
15. C. McCausland. Email disucssion. Email, July 2005.
16. E. Mikk, Y. Lakhnech, C. Petersohn, and M. Siegel. On Formal Semantics of Statecharts as Supported by STATEMATE. In *2nd BCS-FACS Northern Formal Methods Workshop*. Springer-Verlag, 1997.
17. J. Niu, J. M. Atlee, and N. Day. Template Semantics for Model-Based Notations. *IEEE Trans. on Soft. Eng.*, 29(10):866–882, October 2003.
18. J. Niu, J. M. Atlee, and N. A. Day. Understanding and Comparing Model-Based Specification Notations. In *Proc. IEEE Intl. Req. Eng. Conf.*, pages 188–199, 2003.
19. OMG. Unified Modelling Language Specification: Version 2.0, Formal/05-07-04. http://www.omg.org, 2003.
20. A. Pnueli and M. Shalev. What is a Step: On the Semantics of Statecharts. In *Proc. TACS*, volume 526, pages 244–264. Springer-Verlag, 1991.
21. B. Selic. Email disucssion. Email, July 2005.
22. S. Shankar, S. Asa, V. Sipos, and X. Xu. Reasoning about Real-Time State machines in the Presence of Semantic Variations. In *ASE*, pages 243–252, 2005.
23. S. Shlaer and S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Yourdon Press, Upper Saddle River, NJ, USA, 1992.
24. A. Simons. On the Compositional Properties of UML State machine Diagrams. In *Proc. of Rigorous Object-Oriented Methods (ROOM2000)*, York, UK, 2000.
25. M. von der Beeck. A Comparison of State machines Variants. In *Formal Techniques in Real Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer-Verlag, 1994.