Template Semantics for Model-Based Notations

Jianwei Niu, *Student Member*, *IEEE*,
Joanne M. Atlee, *Member*, *IEEE Computer Society*, and Nancy A. Day

Abstract—We propose a template-based approach to structuring the semantics of model-based specification notations. The basic computation model is a nonconcurrent, hierarchical state-transition machine (HTS), whose execution semantics are parameterized. Semantics that are common among notations (e.g., the concept of an enabled transition) are captured in the template, and a notation's distinct semantics (e.g., which states can enable transitions) are specified as parameters. The template semantics of composition operators define how multiple HTSs execute concurrently and how they communicate and synchronize with each other by exchanging events and data. The definitions of these operators use the template parameters to preserve notation-specific behavior in composition. Our template is sufficient to capture the semantics of basic transition systems, CSP, CCS, basic LOTOS, a subset of SDL88, and a variety of statecharts notations. We believe that a description of a notation's semantics using our template can be used as input to a tool that automatically generates formal analysis tools.

Index Terms—Model-based specification notations, semantics, composition, concurrency, automated generation of analysis tools.

1 Introduction

Model-based notations are notations that allow the user to specify a system's dynamic behavior in terms of an abstract model that describes the possible execution steps that the system can take. Model-based notations include process algebras and statecharts variants. Software practitioners like model-based notations because the notations' execution semantics are relatively intuitive and because their composition operators provide facilities for decomposing large problems into modules.

One of the key benefits of modeling software is the ability to detect in the model subtle errors that would be difficult and time-consuming to find in an implementation. There are many well-established analysis tools for automatically verifying model-based specifications. Such analysis tools are either customized for particular notations (e.g., STATEMATE [17], Concurrency Workbench [11]), or users write translators that map their notations to the input language of one or more analyzers [2], [3], [9]. Users can reduce the number of translators they write by translating specifications into an intermediate language [4], [7], [8], from which they translate to the input languages for different tools. In all of these cases, however, an analyzer or translator needs to be written for each notation, and rewritten whenever the notation changes.

To help ease this effort, we and others [13], [15], [33] are working towards generating analyzers and translators automatically from the definition of a notation's semantics– in the manner that we currently generate parsers from grammar definitions. Part of this work involves structuring a notation's semantics to facilitate automated identification of key analysis concepts, such as transitions' enabling conditions and postconditions. Traditional means for defining a notation's semantics, such as operational semantics and logic, can be structured and constrained to support easy extraction of this information. In doing so, we see that many fundamental semantic concepts are common among notations and vary only in their details.

We propose a template-based approach to structure the operational semantics of model-based specification notations. In this approach, which we call **template semantics**, a parameterized template predefines behavior that is common among notations. Template parameters instantiate the parameterized definitions with notation-specific semantics. For example, parameters that specify *enabling states*, *enabling events*, and *enabling variable values* instantiate the template definition of *enabled transitions*, to create a notation-specific function for determining which transitions are enabled in a given execution state. We define composition operators separately as relations that constrain how collections of components execute together, transfer control to one another, and exchange events and data; the operators' definitions use the same template parameters to ensure that the semantics of composition is consistent with the notation's execution semantics.

We developed the template by attempting to capture the common semantics of eight popular specification notations: CSP [19], CCS [27], LOTOS [20], basic transition systems (BTS) [24], SDL88 processes [21], and three variants on statecharts [16], [17], [22]. In doing so, we considered not only the effects of these eight notations on the template definitions, but we also tried to hypothesize how new variants of these notations would be expressed using the template. In another paper [29], we show how to use the template to express the semantics of such disparate notations as SCR [18], SDL88 systems and blocks [21], and basic Petri Nets [32], whose semantics are quite different from the notations in our original survey and from each other.

The main contribution of this work is a new way to structure the operational semantics of a model-based

• The authors are with the School of Computer Science, University of Waterloo, 200 University Avenue W., Waterloo, Ontario, Canada N2L 3G1. E-mail: {jniu, jmatlee, nday}@uwaterloo.ca.

Manuscript received 3 Mar. 2003; revised 5 June 2003; accepted 20 June 2003.
Recommended for acceptance by W. Griswold.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118724.

notation. By separating a notation's execution semantics from its composition and concurrency operators and by parameterizing notations' common execution semantics, the template partitions the definition of a notation's semantics into a set of smaller, simpler definitions. As a result, the template semantics are easier to read and to write, make it easier to compare variants of the same notation (e.g., to compare statecharts variants [29]), and are easier to parse (as the first step in building a semantics-based compiler). By parameterizing the notations' distinct behaviors, template definitions make it easier for students and practitioners to see precisely the semantic differences between languages. The template also makes it possible to experiment with new parameter values, to see if they lead to interesting and useful languages.

The overall goal of our work is to provide a means to describe a notation's semantics so that it can be input to a tool that generates notation-specific analysis tools. We believe that our template semantics may be such a description. By capturing the commonality among the semantics of notations, we can also begin to consider optimizations such as symmetry reductions on the common underlying model rather than on individual notations.

This paper is organized as follows: Section 2 presents our template for defining notations in terms of hierarchical transition systems (HTS), which is our computation model for basic components. In Section 3, we use template semantics to define several composition operators. Section 4 shows how to use our template to express the semantics of existing model-based notations. In Section 5, we discuss the flexibility and limitations of our template in handling sophisticated notational features, such as statecharts' history feature or timing conditions. Related work is discussed in Section 6 and we conclude in Section 7.

2 Hierarchical Transition Systems

We introduce hierarchical transition systems (HTSs) as our computation model for model-based notations. An HTS is a hierarchical, nonconcurrent, extended state machine. In statecharts terminology, an HTS supports OR-state hierarchy but not AND-state hierarchy. Its syntax is adapted from basic transition systems [24] and statecharts [16], and its semantics are parameterized. Concurrency is introduced by the composition operators, which are defined in the next section. Our presentation in this section is more general than our original work [28], in that it supports nondeterministic specifications.

2.1 Syntax of HTS

A hierarchical transition system (HTS) is an 8-tuple, $\langle S, S^I, S^F, S^H, E, V, V^I, T\rangle$. We use these identifiers implicitly in definitions throughout the paper to refer to these HTS elements. S is a finite set of states, and S^I and S^F are predicates describing the sets of initial and final states, respectively. There may be multiple possible initial sets of states. No transition can exit a final state. Each state $s \in S$ is either a **superstate**, which contains other states, or a **basic state**, which contains no other states. Each superstate has a **default** child state, which is entered if the superstate is the destination state of a transition. The state hierarchy S^H defines a partial ordering on states, with the **root state** of an HTS as the minimal element and basic states as maximal elements. E is a finite set of events, including both internal and external events. V is a finite set of typed data variables. V^I is a predicate describing the possible initial value assignments to the variables in V. We assume that the names of unshared events and variables are distinct among HTSs. T is a finite set of transitions, each of which has the form,

$$\langle\ src, trig, cond, act, dest, prty\ \rangle,$$

where $src, dest \subseteq S$ are the transition's sets of source and destination states, respectively; $trig \subseteq E$ is zero or more triggering events; $cond$ is a predicate over V; act is zero or more actions that generate events and assign values to some data variables in V; and $prty$ is the transition's explicitly defined priority. We use sets of sources and destinations to cover notations that allow transitions to have zero or multiple source or destination states. Depending on the notation, some transition elements may be optional. We assume that a specification conforms to the notation's well-formedness conditions. Typical well-formedness conditions prohibit a transition from having multiple destination states or from making multiple assignments to the same variable.

Throughout the paper, we use the helper functions described in Table 1 to access information about an HTS. The first several functions are accessor functions on states and transitions, whereas the functions below the double line are calculated from the HTS's structure. The helper functions are defined for a single state and a single transition, but we will also apply them to sets of states and transitions. A function's meaning with respect to sets is the same, except that the function will return a set of results, one for each combination of argument values.

TABLE 1
HTS Accessor Functions

Function	Signature	Description
$src(\tau)$	$T \rightarrow 2^S$	set of source states of τ
$dest(\tau)$	$T \rightarrow 2^S$	set of destination states of τ
$trig(\tau)$	$T \rightarrow 2^E$	events that trigger τ
$cond(\tau)$	$T \rightarrow exp$	τ's predicate guard condition
$gen(\tau)$	$T \rightarrow 2^E$	events generated by τ's actions
$asn(\tau)$	$T \rightarrow 2^{V \times exp}$	variable assignments in τ's actions, where exp is an expression over V
$prty(\tau)$	$T \rightarrow \mathbb{N}$	τ's priority value
$parent(s)$	$S \rightarrow S$	parent state of s
$children(s)$	$S \rightarrow 2^S$	child states of s
$type(s)$	$S \rightarrow \{super, basic\}$	state type of s
$default(s)$	$S \rightarrow S$	default state of superstate s
$ancest(s)$	$S \rightarrow 2^S$	ancestor states of s
$entered(s)$	$S \rightarrow 2^S$	states entered when state s is entered, including s's ancestors and relevant descendants' default states
$scope(\tau)$	$T \rightarrow S$	lowest common ancestor state of τ's source and destination states (other definitions are possible, *e.g.*, [34])
$rank$(s)	$S \rightarrow \mathbb{N}$	the distance between state s and the root state, where $rank(root) = 0$

τ is a transition and s is a state.

2.2 Semantics of HTSs

We define the semantics of an HTS as a *snapshot relation*. A **snapshot** is an observable point in an HTS's execution, and a **snapshot relation** relates two snapshots ss and ss' if the system can move from ss to ss' in a step. We define two types of steps between snapshots: A **microstep** is the execution of a single transition and a **macrostep** is a sequence of zero or more microsteps. In this section, we describe the semantics of HTSs as parameterized definitions that are common among model-based notations; in Section 2.3, we describe the parameters used in these definitions.

2.2.1 Snapshots

A **snapshot** is an 8-tuple $\langle CS, IE, AV, O, CS_a, IE_a, AV_a, I_a \rangle$. CS is the set of current states ($CS \subseteq S$), such that if any state $s \in CS$, then so are all of s's ancestors. IE is the set of current internal events ($IE \subseteq E$). The set AV is a function that maps each data variable in V to its current value. O is the current outputs to be communicated to concurrent components and to the environment. CS_a, AV_a, IE_a, and I_a are auxiliary elements that accumulate data about the states, the variable values, and the internal and external events, respectively, that were used or generated in past transitions. The template parameters use these auxiliary snapshot elements to derive the sets of enabling states, of enabling variable values, and of enabling events, which, in turn, determine the transitions that are enabled in the current snapshot. Most model-based notations use only a subset of the snapshot elements. An HTS's external inputs, I, are not part of the snapshot because they lie outside of the system. Instead, the template parameters must record input events and data in the snapshot elements. Throughout the paper, we use notation $ss.XX$ to refer to the value of snapshot element XX in snapshot ss (e.g., $ss.CS$).

2.2.2 Microstep Semantics

The **microstep relation** $N_{micro}(ss, \tau, ss')$ means that the HTS can move from snapshot ss to a next snapshot ss' by executing transition τ. Because an HTS is nonconcurrent, only one transition can execute in a microstep. N_{micro} is defined as:

$$\begin{aligned} &N_{micro}(ss, \tau, ss') \equiv \\ &\qquad (\tau \in pri(enabled\_trans(ss, T))) \wedge apply(ss, \tau, ss'), \end{aligned}$$

where

- Function $enabled\_trans$ takes an HTS's transition set T and returns the subset that is enabled in a snapshot ss:

  $$\begin{aligned} &enabled\_trans(ss, T) \equiv \\ &\quad \{\tau \in T \mid en\_states(ss, \tau) \wedge en\_events(ss, \tau) \\ &\qquad \wedge\, en\_cond(ss, \tau)\}, \end{aligned}$$

  where predicates en_states, en_events, and en_cond specify whether a transition τ is enabled with respect to its source states, its triggering events, and its enabling conditions, respectively. These predicates are user-provided template parameters, described in Section 2.3.
- Function pri finds the maximal subset of transitions with the highest relative priority. It is a user-provided template parameter and is also described in Section 2.3. If pri returns more than one transition, then the specification is nondeterministic, and any of the transitions in the returned set is an admissible microstep.
- Predicate *apply* is defined in terms of user-provided template parameters, $next\_XX$, which specify allowable updates to snapshot elements due to the execution of transition τ. The template parameters are described in Section 2.3.

$$\begin{aligned} &apply(ss, \tau, ss') \equiv \\ &\ \textbf{let}\ \langle\, CS', IE', AV', O', CS'_a, IE'_a, AV'_a, I'_a \,\rangle \equiv ss' \\ &\ \textbf{in}\ \ next\_CS(ss, \tau, CS') \ \wedge next\_CS_a(ss, \tau, CS'_a)\ \wedge \\ &\qquad next\_IE(ss, \tau, IE') \ \wedge next\_IE_a(ss, \tau, IE'_a)\ \wedge \\ &\qquad next\_AV(ss, \tau, AV') \wedge next\_AV_a(ss, \tau, AV'_a)\ \wedge \\ &\qquad next\_O(ss, \tau, O') \qquad \wedge next\_I_a(ss, \tau, I'_a) \end{aligned}$$

2.2.3 Macrostep Semantics

A notation's step-semantics are its macrostep semantics, which define how many microsteps an HTS executes in response to a set of external inputs, before sensing the next set of external inputs. External inputs I may be external events, variable-value assignments, or both. We have identified two macrostep semantics, which we call *simple* and *stable*. A **simple** macrostep is equal to at most one microstep. A **stable** macrostep is a maximal sequence of microsteps, such that the sequence ends only when there are no more enabled transitions. Stable macrostep semantics capture the *synchrony hypothesis* [5], which assumes that the system can always finish reacting to external input before the environment changes the inputs' values.

A macrostep starts with the snapshot that ended the last macrostep. Function $reset$ removes from this snapshot information about transitions that executed in the last macrostep (e.g., events generated). It is defined in terms of user-provided template functions, $reset\_XX$, which clean up the snapshot elements XX as per the notation's semantics. Functions $reset\_XX$ are described in Section 2.3.

$$\begin{aligned} &reset(ss, I) \equiv \\ &\quad \langle\, reset\_CS(ss, I), reset\_CS_a(ss, I), \\ &\quad\ \ reset\_IE(ss, I), reset\_IE_a(ss, I), \\ &\quad\ \ reset\_AV(ss, I), reset\_AV_a(ss, I), \\ &\quad\ \ reset\_O(ss, I), reset\_I_a(ss, I) \,\rangle \end{aligned}$$

In **simple macrostep semantics**, an HTS takes at most one microstep per macrostep in reaction to a set of external inputs I. Notations differ as to whether an enabled transition has priority over **idle step** (i.e., taking no transition or stuttering). In **diligent** [24] simple macrostep semantics, an HTS takes a microstep if a transition is enabled, and otherwise makes no change to the reset snapshot.

```
$N^{dil}_{macro}(ss, I, ss') \equiv$
  let $ss^i = reset(ss, I)$ in
    if $(\exists \tau \,.\, \tau \in enabled\_trans(ss^i, T))$
    then $(\exists \tau \,.\, N_{micro}(ss^i, \tau, ss'))$
    else $(ss^i = ss')$
```

In **nondiligent** simple macrostep semantics, idle steps have the same priority as diligent steps.

```
$N^{nondil}_{macro}(ss, I, ss') \equiv$
  let $ss^i = reset(ss, I)$ in
    $((\exists \tau \,.\, N_{micro}(ss^i, \tau, ss')) \vee (ss^i = ss'))$
```

In **stable macrostep semantics**, a macrostep is a maximal sequence of microsteps that execute in response to a single set of external inputs. A snapshot with no enabled transitions is called a **stable snapshot**.

$$stable(ss) \equiv \neg(\exists \tau \,.\, \tau \in enabled\_trans(ss, T))$$

When a stable snapshot is reached, a new macrostep starts with a new set of external inputs I. We define a relation (N^k) that is true for a pair of snapshots if there is a sequence of k microsteps from the first snapshot to the second.

$$\begin{aligned} N^0(ss, ss') &\equiv (ss = ss') \\ N^{k+1}(ss, ss') &\equiv (\exists ss'', \tau \,.\, N_{micro}(ss, \tau, ss'') \wedge N^k(ss'', ss')) \end{aligned}$$

We define a stable macrostep $N_{macro}(ss, ss')$ as a finite sequence of microsteps, terminating in a stable snapshot.

```
$N^{stable}_{macro}(ss, I, ss') \equiv$
  let $ss^i = reset(ss, I)$ in
    if $\neg stable(ss^i)$
    then $((\exists k > 0 \,.\, N^k(ss^i, ss')) \wedge stable(ss'))$
    else $(ss^i = ss')$
```

Some notations, such as RSML [22], do not guarantee that a macrostep reaches a stable state; it is not possible to write a well-founded recursive definition that matches these macrostep semantics. In practice, an analyst would not implement the above definition of a stable macrostep. Rather, the analyst would use the microstep relation as the next-state relation for the system, and would check properties at the macrostep level by prepending the $stable$ predicate as an antecedent to their properties; this is how macrolevel properties were model checked against the RSML specification for TCAS II [9].

2.2.4 *Initial Snapshot*

An HTS starts executing from a set of possible initial snapshots ss^I, whose definition is the same for all macrostep semantics:

$$ss^I \equiv \{ \langle CS, \emptyset, AV, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle \mid AV \models V^I \wedge CS \models S^I \}.$$

V^I and S^I are HTS elements constraining the initial sets of states and of variable assignments, respectively. The sets of internal events and generated events are initially empty. The auxiliary elements are initialized by function $reset$ at the start of the first macrostep. This definition allows there to be multiple possible initial states or initial variable assignments.

TABLE 2
Parameters to Be Provided by Template User

Construct	Start of Macro-step	Micro-step
states	$reset\_CS(ss, I) : CS$	$next\_CS(ss, \tau, CS')$
	$reset\_CS_a(ss, I) : CS_a$	$next\_CS_a(ss, \tau, CS'_a)$
	$en\_states(ss, \tau)$	
events	$reset\_IE(ss, I) : IE$	$next\_IE(ss, \tau, IE')$
	$reset\_IE_a(ss, I) : IE_a$	$next\_IE_a(ss, \tau, IE'_a)$
	$reset\_I_a(ss, I) : I_a$	$next\_I_a(ss, \tau, I'_a)$
	$en\_events(ss, \tau)$	
variables	$reset\_AV(ss, I) : AV$	$next\_AV(ss, \tau, AV')$
	$reset\_AV_a(ss, I) : AV_a$	$next\_AV_a(ss, \tau, AV'_a)$
	$en\_cond(ss, \tau)$	
outputs	$reset\_O(ss, I) : O$	$next\_O(ss, \tau, O')$
Additional Parameters	$macro\_semantics$	
	$pri(\Gamma) : 2^T$	

2.3 Template Parameters

The six definitions $enabled\_trans$, $apply$, $reset$, $stable$, N_{micro}, and N_{macro} described above constitute the common semantics in our template. These definitions are parameterized by functions and predicates that, for a notation, specialize how to determine which transitions are enabled in a snapshot, how to select an enabled transition to execute, and how to calculate the effects of a transition's actions. As such, these parameters capture the semantic differences among notations. These template parameters are also used in Section 3 to help define the semantics of composition operators.

The list of template parameters is provided in Table 2. For the $reset\_XX$ template parameters, we use the return type to show which snapshot element they return. Parameter $macro\_semantics$ specifies the type of macrostep semantics (simple diligent, simple nondiligent, or stable). Parameter pri specifies a priority scheme over a set of transitions. The rest of the template parameters are predicates. These predicates are passed the entire snapshot because they may refer to snapshot elements other than the one they are affecting. Primed arguments (e.g., CS') refer to values of snapshot elements in next snapshot ss'.

Table 2 is organized by language construct. For example, the seven event-related parameters work together to determine which events can enable transitions:

- $reset\_IE$, $reset\_IE_a$, and $reset\_I_a$ update elements IE, IE_a, and I_a at the start of each macrostep. Their main purpose is to record the environment's input I and to clean out event-related data accumulated in the previous macrostep (e.g., reset the set of current events to be empty).
- en_events determines how event information in IE, IE_a, and I_a is used to enable transitions.

TABLE 3
Sample Definitions for State-Related Template Parameters

Parameter		statecharts [16]	RSML, STATEMATE
$reset\_CS(ss, I)$	$\equiv$	$ss.CS$	
$next\_CS(ss, \tau, CS')$	$\equiv$	$CS' = entered\,(dest(\tau))$	
$reset\_CS_a(ss, I)$	$\equiv$	$ss.CS$	n/a
$next\_CS_a(ss, \tau, CS'_a)$	$\equiv$	$CS'_a = \emptyset$	n/a
$en\_states(ss, \tau)$	$\equiv$	$src(\tau) \subseteq ss.CS_a$	$src(\tau) \subseteq ss.CS$

- $next\_IE$, $next\_IE_a$, and $next\_I_a$ determine how transition τ's actions affect the event-related information stored in elements IE, IE_a, and I_a in the next snapshot ss'.

The following five sections describe how a specifier can use these parameter functions to define the step-semantics for some popular notations. Tables 3, 4, 5, and 6 provide example definitions. In the tables, abbreviation "n/a" means "not applicable," in which case the predicate always returns "true."

2.3.1 States

Table 3 shows example definitions for the state-related template parameters. In RSML and STATEMATE, a transition is enabled if its source states are a subset of the current states; this semantics allows infinite loops in a macrostep. Harel et al.'s [16] original formulation of statecharts avoids infinite loops by allowing each nonconcurrent component (i.e., each HTS) to execute at most one transition per macrostep; a sequence of microsteps comprises transitions from multiple concurrent components. We express this semantics by using snapshot element CS to maintain the set of current states, by using element CS_a to maintain the set of enabling states, and by setting CS_a to the empty set after the HTS takes a step, to disallow future transitions. We can envision an alternate state semantics that allows an HTS to take multiple microsteps in a macrostep, and prevents infinite loops by prohibiting states from being exited more than once in a macrostep.

2.3.2 Events

Table 4 shows example definitions for the event-related template parameters. Events can be internal and/or external events. Process algebras such as CCS use only external events. Statecharts-based notations have both internal and external events. For notations that differentiate syntactically between internal events and external events, we use $intern\_ev(E)$ to mean the set of internal events and $extern\_ev(E)$ for the set of external events.

In RSML and STATEMATE, only internal events generated in the previous microstep can trigger a transition, whereas, in the original statecharts semantics, any internal event generated since the start of the macrostep is an enabling event.

In all statecharts variants, external inputs I are enabling events at the start of the macrostep. In RSML and STATEMATE, external events can trigger transitions only in the first microstep of a macrostep. In statecharts, external events remain enabling events throughout the macrostep. We can also imagine a new notation in which an external event remains an enabling event until it is used to trigger a transition.

Table 4 also shows how the outputs for a microstep are determined. For all the notations, a macrostep starts with an empty set of outputs. Statecharts accumulate as outputs all the events generated during the microstep. RSML accumulates only the external events. STATEMATE considers as outputs only the events generated in the last microstep of the macrostep.

2.3.3 Variable Values

Table 5 shows example definitions for variable-related template parameters. In most notations, transitions' enabling conditions and assignment expressions are evaluated with respect to the current variable values. In contrast, in the original statecharts semantics, conditions and expressions are evaluated with respect to variable values that hold at the start of the macrostep, except for expressions within a cr operator, which are evaluated with respect to current variable assignments. Thus, for statecharts, we use snapshot element AV to maintain the set of current variable values and use element AV_a to maintain the variable values from the start of the macrostep; and we evaluate enabling conditions and assignment expressions with respect to both

TABLE 4
Sample Definitions for Event-Related Template Parameters

Parameter		CCS	statecharts [16]	RSML	STATEMATE
$reset\_IE(ss, I)$	$\equiv$	n/a	$\emptyset$		
$next\_IE(ss, \tau, IE')$	$\equiv$	n/a	$IE' = gen(\tau)$	$IE' = gen(\tau) \cap intern\_ev(E)$	$IE' = gen(\tau)$
$reset\_IE_a(ss, I)$	$\equiv$	n/a	$\emptyset$	n/a	n/a
$next\_IE_a(ss, \tau, IE'_a)$	$\equiv$	n/a	$IE'_a = ss.IE_a \cup gen(\tau)$	n/a	n/a
$reset\_I_a(ss, I)$	$\equiv$	I			
$next\_I_a(ss, \tau, I'_a)$	$\equiv$	$I'_a = ss.I_a \cup gen(\tau)$	$I'_a = ss.I_a$	$I'_a = \emptyset$	
$en\_events(ss, \tau)$	$\equiv$	$trig(\tau) \subseteq ss.I_a$	$trig(\tau) \subseteq ss.IE_a \cup ss.I_a$	$trig(\tau) \subseteq ss.IE \cup ss.I_a$	
$reset\_O(ss, I)$	$\equiv$	$\emptyset$			
$next\_O(ss, \tau, O')$	$\equiv$	$O' = gen(\tau)$	$O' = ss.O \cup gen(\tau)$	$O' = ss.O \cup (gen(\tau) \cap extern\_ev(E))$	$O' = gen(\tau)$

TABLE 5
Sample Definitions for Variable-Related Template Parameters

Parameter		statecharts [16]	RSML	STATEMATE
$reset\_AV(ss,I)$	$\equiv$	$ss.AV$		
$next\_AV(ss,\tau,AV')$	$\equiv$	$AV' = assign(ss.AV, eval((ss.AV, ss.AV_a), asn(\tau)))$	$AV' = assign(ss.AV, eval(ss.AV, asn(\tau)))$	$AV' = assign(ss.AV, eval(ss.AV, last(asn(\tau))))$
$reset\_AV_a(ss,I)$	$\equiv$	$ss.AV$	n/a	
$next\_AV_a(ss,\tau,AV'_a)$	$\equiv$	$AV'_a = ss.AV_a$	n/a	
$en\_cond(ss,\tau)$	$\equiv$	$ss.AV, ss.AV_a \models cond(\tau)$	$ss.AV \models cond(\tau)$	

$assign(X,Y)$:	updates assignments X with assignments Y; ignores assignments in Y to variables not in X
$eval(X,A)$:	evaluates expressions in A with respect to assignments X, and returns variable-value assignments for A
$last(A)$:	returns a subsequence of A, comprising only the last assignment to each variable in the sequence of assignments A

snapshot elements. Variable values are updated by overriding current value assignments with the transition's variable assignments, using the function $assign$ described in Table 5. In STATEMATE, if a transition makes multiple assignments to the same variable, only the last assignment to the variable has an effect. RSML and statecharts do not allow transitions to make multiple assignments to the same variable.

Some notations, such as the SMV input language [25], allow variable assignments to refer to values that hold at the start of the next microstep. By making the $next\_AV$ parameter a predicate that takes the next snapshot's variable values as an argument, we can accommodate such forward-referencing semantics for notations.

2.3.4 Priority

Table 6 shows example definitions for template function pri, which returns the subset of transitions of highest priority. STATEMATE prioritizes transitions by the *ranks* of their *scope*, where $rank$ and $scope$ are described in Table 1. Lower-ranked scopes have priority over higher-ranked scopes, which means that superstate behavior is favored over substate behavior. UML [30] prioritizes transitions by the *ranks* of their *source* states. Transitions with higher-ranked source states have priority over transitions with lower-ranked source states, which means that substate behavior overrides superstate behavior. Some notations allow the specifier to use explicit priorities on transitions (i.e., $prty$) to override a notation's default priority scheme.

3 Composition Operators

In this section, we describe the semantics of a number of well-used composition operators found in process algebras, statecharts variants, and SDL. A composition operator specifies how multiple HTSs execute concurrently. The operands of a composition operator are components, where a component is either a basic component (i.e., an HTS) or a collection of smaller, composed components. We define our composition operators by specifying how the components' snapshots change when the components take a step.

For most composition operators, we define the operator's behavior at the microstep level, and we infer its semantics at the macrostep level as a sequence of zero or more composed microsteps. This allows components to communicate events and variable values with each microstep. However, for operators in which components share information only at the end of their macrosteps, we express composition at the macrostep level. At the start of such a macrostep, each component's output from the previous macrostep is added to the inputs sensed by the other component. If a notation's macrostep semantics is not well-defined (i.e., allows an infinite sequence of microsteps), then macrostep composition for that notation is also not well-defined.

We define composition operators as parameterized, composite, microstep or macrostep relations that relate pairs of consecutive snapshot collections. For example, the composite microstep relation for an operator op is

$$N^{op}_{micro}((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)),$$

which describes the relations between snapshot collections $\vec{ss}_1$ and $\vec{ss}'_1$, and between $\vec{ss}_2$ and $\vec{ss}'_2$, when components one and two execute their respective transition collections $\vec{\tau}_1$ and $\vec{\tau}_2$ in the same microstep. The snapshot collections and transition collections are tree structures that match the specification's composition hierarchy. We overload the notation $\emptyset$ to also mean an empty tree.

The definition of N^{op}_{micro} is based on the microstep semantics, N_{micro}, of the two components. The composition operator also defines how the changes in each component's snapshots are shared, and how control transfers from one component to another. We represent these effects by using a **substitution notation**:

TABLE 6
Sample Definitions for Template Parameter pri

Parameter		STATEMATE	UML [30]
$pri(\Gamma)$	$\equiv$	$\{\tau \in \Gamma \mid \forall t \in \Gamma .\, rank(scope(\tau)) \leq rank(scope(t))\}$	$\{\tau \in \Gamma \mid \forall t \in \Gamma .\, rank(src(\tau)) \geq rank(src(t))\}$

TABLE 7
Abbreviations Used in the Semantics of Composition

Snapshot Element	$update(\vec{ss}, \vec{\tau}, \vec{ss}') \equiv$	$communicate(i\vec{ss}, \vec{ss}, \vec{\tau}, \vec{ss}') \equiv$
CS	$\vec{ss}'.CS = \vec{ss}.CS$	$\vec{ss}'.CS = i\vec{ss}.CS$
CS_a	$\vec{ss}'.CS_a = \vec{ss}.CS_a$	$\vec{ss}'.CS_a = i\vec{ss}.CS_a$
IE	$next\_IE(\vec{ss}, \vec{\tau}, \vec{ss}'.IE)$	$next\_IE(\vec{ss}, \vec{\tau}, \vec{ss}'.IE)$
IE_a	$next\_IE_a(\vec{ss}, \vec{\tau}, \vec{ss}'.IE_a)$	$next\_IE_a(\vec{ss}, \vec{\tau}, \vec{ss}'.IE_a)$
I_a	$next\_I_a(\vec{ss}, \vec{\tau}, \vec{ss}'.I_a)$	$next\_I_a(\vec{ss}, \vec{\tau}, \vec{ss}'.I_a)$
O	$\vec{ss}'.O = \vec{ss}.O$	$\vec{ss}'.O = i\vec{ss}.O$

$$ss' = ss|_{v}^{x},$$

which means that snapshot ss' is equal to ss, except for element x, which has value v. Substitution over a collection of snapshots ($\vec{ss}' = \vec{ss}|_{v}^{x}$) defines substitutions for corresponding pairs of snapshots at the leaves of the trees $\vec{ss}$ and $\vec{ss}'$. For example, substitution $\vec{ss}' = \vec{ss}|_{\emptyset}^{cs}$ is equal to snapshot collection $\vec{ss}$, except that the sets of current states CS in all of $\vec{ss}'$'s leaf snapshots are empty. Two snapshot collections are equal if their corresponding leaf snapshots are equal (we only compare snapshot collections that have corresponding composition hierarchies).

When applied to collections of snapshots and to collections of transitions, the template parameters, $next\_XX(\vec{ss}, \vec{\tau}, \vec{ss}')$, define how snapshot elements XX in corresponding leaf snapshots in collections $\vec{ss}$ and $\vec{ss}'$ change value due to the execution of all of the transitions in $\vec{\tau}$. The helper functions from Table 1 that appear in the template parameters are also generalized to apply to sets of transitions.

In Table 7, we introduce predicates *update* and *communicate*, which are used in the definitions of the composition operators to describe how components communicate events. The predicate *update* is used to specify how the snapshots of a nonexecuting component are affected by the shared events generated by transitions $\vec{\tau}$ in the executing component. The predicate *communicate* is used when both components execute, to specify how the snapshots of one executing component may be affected by the shared events generated by the other executing component. The predicate *communicate* starts from an intermediate snapshot ($i\vec{ss}$) that reflects the effects of the component's own transitions. However, *communicate* changes the event-related snapshot elements with respect to the beginning snapshot collection $\vec{ss}$. The predicate *communicate* is usually provided with the executing transitions from both components, so that a rational ordering among all the transitions' generated events, if desired, can be collected into the event-related snapshot elements. Both predicates use the template parameters, so that they adhere to their components' semantics for updating snapshot elements.

Variable values are not modified by *communicate* and *update* because conflicts among the assignments to shared variables must be resolved so that all components have the same value associated with a variable. At the composition level, there is one template parameter,

$$resolve(\vec{AV_1}, \vec{AV_2}, asnAV),$$

to capture different notations' policies for resolving conflicting variable-value assignments. This predicate specifies how sets of assignments $\vec{AV_1}$ and $\vec{AV_2}$ can be resolved to a single set of variable-value assignments $asnAV$. To handle the communication of shared variable-values, we introduce the predicate *communicate_vars*, which uses template parameter *resolve*:

$$\begin{aligned}
&communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \equiv \\
&\quad \exists \vec{AV_1},\ \vec{AV_2},\ asnAV,\ \tau\ . \\
&\qquad next\_AV(\vec{ss}_1, \vec{\tau}_1, \vec{AV_1}) \wedge next\_AV(\vec{ss}_2, \vec{\tau}_2, \vec{AV_2}) \wedge \\
&\qquad resolve(\vec{AV_1}, \vec{AV_2}, asnAV) \wedge asn(\tau) = asnAV \wedge \\
&\qquad \vec{ss}'_1.AV = assign(\vec{ss}_1.AV, asnAV) \wedge \\
&\qquad \vec{ss}'_2.AV = assign(\vec{ss}_2.AV, asnAV) \wedge \\
&\qquad next\_AV_a(\vec{ss}_1, \tau, \vec{ss}_1.AV'_a) \wedge \\
&\qquad next\_AV_a(\vec{ss}_2, \tau, \vec{ss}_2.AV'_a).
\end{aligned}$$

Predicate *communicate_vars* creates a dummy transition τ whose actions consist of the resolved assignments; it uses the template parameters and τ to ensure that each component receives the same variable assignments and yet is able to treat the assignments as per the notation's semantics. Predicates *communicate_vars* and *update* (or *communicate*) can be applied to the same snapshot collection $\vec{ss}$ without conflict, because the predicates constrain complementary elements.

We also introduce predicates *bothstep*, *comp1steps*, and *comp2steps* (Fig. 1), which combine predicates that commonly occur together in the definitions of composition operators. Predicate *bothstep* captures the case in which

$$\begin{aligned}
&bothstep((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \equiv \\
&\quad \exists\ i\vec{ss}_1, i\vec{ss}_2\ .\ \left[\begin{array}{ll} & N^1_{micro}(\vec{ss}_1, \vec{\tau}_1, i\vec{ss}_1) \ \wedge\ N^2_{micro}(\vec{ss}_2, \vec{\tau}_2, i\vec{ss}_2) \\ \wedge & communicate(i\vec{ss}_1, \vec{ss}_1, \vec{\tau}_1 \cup \vec{\tau}_2, \vec{ss}'_1) \ \wedge\ communicate(i\vec{ss}_2, \vec{ss}_2, \vec{\tau}_1 \cup \vec{\tau}_2, \vec{ss}'_2) \\ \wedge & communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \end{array}\right]
\end{aligned}$$

$$comp1steps((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \equiv N^1_{micro}(\vec{ss}_1, \vec{\tau}_1, \vec{ss}'_1) \ \wedge\ \vec{\tau}_2 = \emptyset \ \wedge\ update(\vec{ss}_2, \vec{\tau}_1, \vec{ss}'_2) \ \wedge\ communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \emptyset), (\vec{ss}'_1, \vec{ss}'_2))$$

$$comp2steps((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \equiv N^2_{micro}(\vec{ss}_2, \vec{\tau}_2, \vec{ss}'_2) \ \wedge\ \vec{\tau}_1 = \emptyset \ \wedge\ update(\vec{ss}_1, \vec{\tau}_2, \vec{ss}'_1) \ \wedge\ communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\emptyset, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2))$$

Fig. 1. Predicates for both components taking a step, component 1 taking a step, and component 2 taking a step.

$$N_{micro}^{para}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv$$
$$\textbf{if } \neg stable(\vec{ss}_1) \wedge \neg stable(\vec{ss}_2)$$
$$\textbf{then } bothstep((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \quad \text{(* both take a step *)}$$
$$\textbf{else } (comp1steps((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \vee \text{(* symmetric case *)}) \quad \text{(* one executes; other changes shared variables and events *)}$$

$$N_{micro}^{para\text{-}Harel}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv$$
$$\vee \begin{array}{ll} (bothstep((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \wedge dom(asn(\vec{\tau}_1)) \cap dom(asn(\vec{\tau}_2)) = \emptyset) & \text{(* both take a step *)} \\ (comp1steps((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \vee \text{(* symmetric case *)}) & \text{(* one executes; other changes shared variabled and events *)} \end{array}$$

$$N_{macro}^{para\text{-}SDL}((\vec{ss}_1, \vec{ss}_2),\ I,\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv\ N_{macro}^1(\vec{ss}_1, I \cup \vec{ss}_2.O, \vec{ss}'_1) \wedge N_{macro}^2(\vec{ss}_2, I \cup \vec{ss}_1.O, \vec{ss}'_2)$$

Fig. 2. Semantics of parallel composition for micro and macrosteps.

both components take a microstep: The components' next snapshots should satisfy N^1_{micro} and N^2_{micro}, except for the values of shared variables and events. We introduce intermediate snapshots $i\vec{ss}_1$ and $i\vec{ss}_2$ that reflect the effects of the components' N_{micro} relations, and we use the predicates *communicate* and *communicate_vars* to define the next snapshots $\vec{ss}'_1$ and $\vec{ss}'_2$ in terms of the intermediate snapshots and the shared events and variable values. Predicates *comp1steps* and *comp2steps* capture the cases in which one component takes a microstep in isolation, and the other component's snapshots are simply updated to include the shared events and variable assignments generated by the executing component's transitions. Many of the composition operators can be defined using these predicates, combined with additional predicates that reflect the operator's unique pre and postconditions.

In the following, we describe six composition operators: parallel, interleaving, synchronization, sequence, choice, and interrupt. Although many of these operators can be defined at both the macrostep and the microstep levels and for both diligent and nondiligent semantics, we present only the operator variants that correspond to the key composition operators in our original survey of eight notations. In all cases except interrupt composition, the initial snapshot for the composed machine comprises the component machines' initial snapshots ($\vec{ss}^I = (\vec{ss}^I_1, \vec{ss}^I_2)$). We assume that initial values of shared variables are consistent among components.

3.1 Parallel

Fig. 2 shows various definitions of parallel composition. In parallel composition at the microstep level, N^{para}_{micro}, both components execute transitions in the same microstep if both components have enabled transitions; otherwise, only one component executes and the other updates shared variables and events. The case where both components do not change (i.e., no transition is executed) is not an allowable microstep. This composition operator matches the AND-state composition found in most statecharts variants.

Composition operator, $N^{para-Harel}_{micro}$, used in Harel's statechart semantics [16], differs from N^{para}_{micro} in that $N^{para-Harel}_{micro}$ does not force both components to execute if they are both enabled and it prohibits parallel transitions from making assignments to the same variable.

In parallel composition at the macrostep level, both components must take macrosteps (which may be idle steps). The components' N_{macro} relations determine whether this composition is diligent or nondiligent. The outputs from each component's previous step are added as inputs to the other component's next step. The only notation that we have surveyed that uses parallel composition at the macrostep level is SDL processes. In SDL, processes do not share variables. Therefore, our expression of $N^{para-SDL}_{macro}$ does not include any resolution for conflicting variable assignments, which could be needed for another notation.

3.2 Interleaving

In interleaving composition, only one component can execute transitions in a step (Fig. 3). In microstep interleaving, N^{intl}_{micro}, exactly one component takes a step; because idle transitions are not admissible microsteps, microstep semantics are necessarily diligent.

At the macrostep level, interleaving composition could be either *diligent* or *nondiligent*. In nondiligent interleaving, $N^{intl-nondil}_{macro}$, either component, but not both, can take a step, regardless of which components are enabled. This operator is used by BTS.

$$N_{micro}^{intl}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv\ comp1steps((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}'_1, \vec{ss}'_2)) \vee \text{(* symmetric case *)}$$

$$N_{macro}^{intl\text{-}nondil}((\vec{ss}_1, \vec{ss}_2),\ I,\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv$$
$$\vee \begin{array}{l} N^1_{macro}(\vec{ss}_1, I \cup \vec{ss}_2.O, \vec{ss}'_1) \wedge \vec{ss}'_2 = \vec{ss}_2 \,|^{AV}_{assign(\vec{ss}_2.AV,\ \vec{ss}'_1.AV)} \\ N^2_{macro}(\vec{ss}_2, I \cup \vec{ss}_1.O, \vec{ss}'_2) \wedge \vec{ss}'_1 = \vec{ss}_1 \,|^{AV}_{assign(\vec{ss}_1.AV,\ \vec{ss}'_2.AV)} \end{array} \quad \text{(* either component can take a step, but not both *)}$$

Fig. 3. Semantics of interleaving composition for micro and macrosteps.

$$
\begin{array}{l}
N^{\mathit{env\text{-}sync}}_{\mathit{micro}}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}', \vec{ss}_2{}'))\ \mathit{sync\_events} \equiv \\
\quad \exists\, e \left[\begin{array}{ll} & \mathit{trig}(\vec{\tau}_1 \cup \vec{\tau}_2) = \{e\} \ \wedge\ e \in \mathit{sync\_events} \\ \wedge & \forall T \in \vec{T}_1 \cup \vec{T}_2 \,.\, (\exists\, \tau \in T.\ \mathit{trig}(\tau) = \{e\}) \Longrightarrow ((\vec{\tau}_1 \cup \vec{\tau}_2) \cap T \neq \varnothing) \\ \wedge & \mathit{bothstep}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}', \vec{ss}_2{}')) \end{array} \begin{array}{l} \text{(* line 1 *)} \\ \text{(* line 2 *)} \end{array} \right] \text{(* multiple components sync on same event *)} \\
\vee \\
\quad \left[\begin{array}{ll} & \mathit{trig}(\vec{\tau}_1) \cap \mathit{sync\_events} = \varnothing \ \wedge\ \mathit{comp1steps}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}', \vec{ss}_2{}')) \\ \vee & \text{(* symmetric case of above replacing 1 with 2 and 2 with 1 *)} \end{array} \right] \text{(* unsync case: only one component executes. *)}
\end{array}
$$

Fig. 4. Semantics of environmental synchronization for microsteps.

$$
\begin{array}{l}
\mathit{comm\_event}(\vec{ss}, \vec{\tau}, \vec{ss}') \equiv \\
\quad \vec{ss}.CS = \vec{ss}'.CS \ \wedge\ \vec{ss}.CS_a = \vec{ss}'.CS_a \ \wedge\ \vec{ss}.AV = \vec{ss}'.AV \ \wedge\ \vec{ss}.AV_a = \vec{ss}'.AV_a \ \wedge\ \vec{ss}.O = \vec{ss}'.O \ \wedge \\
\quad \mathit{next\_IE}(\vec{ss}, \vec{\tau}, \vec{ss}'.IE) \ \wedge\ \mathit{next\_IE_a}(\vec{ss}, \vec{\tau}, \vec{ss}'.IE_a) \ \wedge\ \mathit{next\_I_a}(\vec{ss}, \vec{\tau}, \vec{ss}'.I_a) \\
\\
N^{\mathit{rend\text{-}sync}}_{\mathit{micro}}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}', \vec{ss}_2{}'))\ \mathit{sync\_events} \equiv \\
\quad \exists\, \vec{iss}_1, \vec{iss}_2, \vec{rss}_2, e. \left[\begin{array}{ll} & e \in \mathit{sync\_events} \wedge \mid \vec{\tau}_1 \mid = 1 = \mid \vec{\tau}_2 \mid \ \wedge\ \mathit{gen}(\vec{\tau}_1) = \mathit{trig}(\vec{\tau}_2) = \{e\} \\ \wedge & \mathit{comm\_event}(\vec{ss}_2, \vec{\tau}_1, \vec{rss}_2) \ \wedge\ \mathit{trig}(\vec{\tau}_1) = \varnothing \ \wedge\ \mathit{gen}(\vec{\tau}_2) = \varnothing \\ \wedge & N^1_{\mathit{micro}}(\vec{ss}_1, \vec{\tau}_1, \vec{iss}_1) \ \wedge\ \mathit{communicate}(\vec{iss}_1, \vec{ss}_1, \varnothing, \vec{ss}_1{}') \\ \wedge & N^2_{\mathit{micro}}(\vec{rss}_2, \vec{\tau}_2, \vec{iss}_2) \ \wedge\ \mathit{communicate}(\vec{iss}_2, \vec{ss}_2, \varnothing, \vec{ss}_2{}') \\ \wedge & \mathit{communicate\_vars}((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}_1{}', \vec{ss}_2{}')) \end{array} \right] \begin{array}{l} \text{(* sync case: sync event generated by} \\ \text{one transition in component 1} \\ \text{triggering only one transition in component 2 *)} \end{array} \\
\vee \\
\quad \text{(* symmetric case of above: transition in component 2 generates a sync event that triggers a transition in component 1 *)} \\
\vee \\
\quad \left[\begin{array}{ll} & \mathit{gen}(\vec{\tau}_1) \cap \mathit{sync\_events} = \varnothing \ \wedge\ \mathit{trig}(\vec{\tau}_1) \cap \mathit{sync\_events} = \varnothing \ \wedge\ \mathit{comp1steps}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}', \vec{ss}_2{}')) \\ \vee & \text{(* symmetric case of above replacing 1 with 2 and 2 with 1 *)} \end{array} \right] \text{(* unsync case *)}
\end{array}
$$

Fig. 5. Semantics of rendezvous synchronization for microsteps.

3.3 Synchronization

We define two synchronization operators at the microstep level: environmental synchronization and rendezvous synchronization (Figs. 4 and 5).

In environmental synchronization, $N^{env-sync}_{micro}$, both components execute in the same microstep if the executing transitions all have the same trigger event, e, which is a designated synchronization event (line 1), and if all components that can react to this event participate in the step (line 2). This clause refers back to the basic components' sets of transitions $\vec{T}_1$ and $\vec{T}_2$, and tests that each HTS that synchronizes on event e contributes a transition to either $\vec{\tau}_1$ or $\vec{\tau}_2$. In the "unsync" case, none of the executing transitions is triggered by a synchronization event, so one or the other component takes a step in isolation (interleaving). Environmental synchronization corresponds to the parallel composition operators of CCS ($P||Q$), CSP ($P||Q$), and LOTOS ($P|[a, b, c]|Q$).

In rendezvous synchronization, $N^{rend-sync}_{micro}$, exactly one transition in the sending component generates a synchronization event that triggers exactly one transition in the receiving component. Rendezvous is the only example of a composition operator in which events generated in one component are transferred to the other component within the same microstep. This requires an extra intermediate snapshot collection ($r\vec{s}s_2$), which is set by $comm\_event$ to incorporate $\vec{\tau}_1$'s generated event into $r\vec{s}s_2$'s event-related snapshot elements; $r\vec{s}s_2$ then becomes the starting snapshot collection for component two. We use $communicate$ here with an empty set of transitions to ensure that neither of the final snapshots $\vec{ss}_1'$ and $\vec{ss}_2'$ store $\vec{\tau}_1$'s generated event since it has been transferred and processed in this microstep. Rendezvous is used in CCS ($a.P \mid \bar{a}.Q$). Although CCS does not have variables, we include the $communicate\_vars$ constraint in the general definition of this operator.

3.4 Sequence

In sequence composition (Fig. 6), the first component executes in isolation until it terminates (i.e., reaches its final basic states) and then the second component executes in isolation. If component one is a composite component, then all of its basic components must reach final basic states before the second component can start. (Recall that no transition can exit a final state in an HTS.) Sequence

$$
\begin{array}{l}
N^{\mathit{seq}}_{\mathit{micro}}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}',\ \vec{ss}_2{}')) \equiv \\
\quad [\ \mathit{basic\_states}(\vec{ss}_1.CS) \not\subseteq S^F_1 \ \wedge\ \mathit{comp1steps}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}_1{}',\ \vec{ss}_2{}'))\ ] \ \text{(* component 1 steps *)} \\
\vee \\
\quad \left[\begin{array}{ll} & \mathit{basic\_states}(\vec{ss}_1.CS) \subseteq S^F_1 \ \wedge\ \vec{ss}_1.CS \neq \varnothing \ \wedge\ \vec{\tau}_1 = \varnothing \ \wedge\ N^2_{\mathit{micro}}(\vec{ss}_2,\ \vec{\tau}_2,\ \vec{ss}_2{}') \\ \wedge & \mathit{update}(\vec{ss}_1 \mid^{CS}_{\varnothing}, \vec{\tau}_2, \vec{ss}_1{}') \ \wedge\ \mathit{communicate\_vars}((\vec{ss}_1, \vec{ss}_2), (\varnothing, \vec{\tau}_2), (\vec{ss}_1{}', \vec{ss}_2{}')) \end{array} \right] \text{(* component 2 starts and steps *)} \\
\vee \\
\quad [\ \vec{ss}_1.CS = \varnothing \ \wedge\ \mathit{comp2steps}((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{ss}_1{}',\ \vec{ss}_2{}'))\ ] \ \text{(* component 2 steps *)}
\end{array}
$$

Fig. 6. Semantics of sequence composition for microsteps.

$$
\begin{array}{l}
N^{choice}_{micro}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2)) \equiv \\
\quad \left[\begin{array}{l}
\quad \vec{ss}_1.CS \subseteq S^I_1 \ \wedge\ \vec{ss}_2.CS \subseteq S^I_2 \ \wedge\ \vec{ss}_1.CS \neq \varnothing \ \wedge\ \vec{ss}_2.CS \neq \varnothing \\
\wedge \left(\left[\begin{array}{ll} \wedge & N^1_{micro}(\vec{ss}_1, \vec{\tau}_1, \vec{ss}'_1) \ \wedge\ \vec{\tau}_2 = \varnothing \\ & update(\vec{ss}_2 \,|^{CS}_{\varnothing}, \vec{\tau}_1, \vec{ss}'_2) \ \wedge\ communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\vec{\tau}_1, \varnothing), (\vec{ss}'_1, \vec{ss}'_2)) \end{array} \right] \vee \begin{array}{l}\text{(* symmetric case} \\ \text{for component 2 *)}\end{array} \right)
\end{array} \right] \text{(* choose a component *)} \\
\vee \\
\quad ((\vec{ss}_2.CS = \varnothing \ \wedge\ comp1steps((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2))) \ \vee\ \text{(* symmetric case for component 2 stepping *))} \quad \text{(* chosen component steps *)}
\end{array}
$$

Fig. 7. Semantics of choice composition for microsteps.

composition is used in process algebras such as CCS and CSP ($P;Q$).

There are three stages to a sequence composition. In the first stage, component one executes and the shared variables of component two are updated. We introduce the function $basic\_states$, which returns the subset of the current states that are basic states, in the test of whether component one has terminated. In the second stage, component one has reached its final states, control transfers to component two, and component two takes a step. The operator also clears the states from component one's snapshots so that component one can no longer execute. In the third stage, component two executes and, for consistency, the snapshots of component one are updated.

3.5 Choice

The choice composition operator (Fig. 7) nondeterministically chooses one component to execute in isolation. Once this choice is made, the composite machine behaves only like the chosen component, and never executes the other component. Choice composition is used in LOTOS ($P[\,]Q$), CCS ($P+Q$), and CSP ($P[\,]Q$). We capture these semantics by clearing the set of current states from the unchosen component's snapshots to keep it from executing. For consistency, we continue to update the unchosen component's snapshots.

3.6 Interrupt

Interrupt composition allows control to pass between two components via a provided set of **interrupt transitions** (T_{interr}). These transitions may have sources and destinations that are substates of the components. We use interrupt composition to describe transitions between statecharts components that have AND-states as subcomponents.

In interrupt composition (Fig. 8), there are four cases. In the first case, component one has enabled transitions $\vec{\tau}_1$, and any enabled interrupt transitions have lower priority than $\vec{\tau}_1$. Therefore, component one executes and component two is updated. We introduce the predicate $higher\_pri(x, y)$ to test if a transition in the set x has equal or higher priority than the transitions in the set y; this predicate is defined in terms of the template parameter pri, which may be based on the state and composition hierarchies. The state hierarchy and the ranks of states grow as components are composed.

In the second case, one of the interrupt transitions is enabled and has priority over all enabled transitions in component one, which means that control passes from component one to component two. We introduce predicate ent_comp to determine the current states of component two; predicate ent_comp uses the state and composition hierarchy of component two and the set of states entered by the executing transition to determine which default states also need to be entered (e.g., default states of concurrent subcomponents). The composition operator also clears the current states in component one, so that the component will not execute, and it applies the actions of the executing transition to component one's snapshots.

The final two cases of interrupt composition semantics are symmetric to the first two cases, in that we now consider transitions whose source states are in component two. Only one component ever has current states, so only one component can have enabled transitions in any snapshot.

The initial composite snapshot for interrupt composition requires the designation of one of the components as the starting component. The current states for this component are set to its default states, and the current states for the other component are set to empty.

4 Semantics of Notations

We can describe the semantics of several specification notations concisely in template semantics. Table 8 provides values for the template parameters for eight notations; it also maps the notations' composition operators to our

$$
\begin{array}{l}
startcomp(\vec{ss}, \vec{\tau}, \vec{ss}') \equiv \\
\quad ent\_comp(\vec{ss}, \vec{\tau}, \vec{ss}'.CS) \ \wedge\ next\_CS_a(\vec{ss}, \vec{\tau}, \vec{ss}'.CS_a) \ \wedge\ next\_IE(\vec{ss}, \vec{\tau}, \vec{ss}'.IE) \ \wedge\ next\_IE_a(\vec{ss}, \vec{\tau}, \vec{ss}'.IE_a) \ \wedge \\
\quad next\_I_a(\vec{ss}, \vec{\tau}, \vec{ss}'.I_a) \ \wedge\ next\_O(\vec{ss}, \vec{\tau}, \vec{ss}'.O) \\
\\
N^{interr}_{micro}((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2))\ T_{interr} \equiv \\
\qquad [\ \vec{ss}_1.CS \neq \varnothing \ \wedge\ higher\_pri(\vec{\tau}_1, pri(enabled\_trans(\vec{ss}_1, T_{interr}))) \ \wedge\ comp1steps((\vec{ss}_1, \vec{ss}_2),\ (\vec{\tau}_1, \vec{\tau}_2),\ (\vec{ss}'_1, \vec{ss}'_2))\ ] \ \text{(* component 1 steps *)} \\
\vee \quad \exists\, \tau. \left[\begin{array}{ll} & \tau \in pri(enabled\_trans(\vec{ss}_1,\ T_{interr})) \ \wedge\ higher\_pri(\{\tau\}, pri(enabled\_trans(\vec{ss}_1,\ \vec{T}_1))) \\ \wedge & \vec{\tau}_1 = \{\tau\} \ \wedge\ \vec{\tau}_2 = \varnothing \ \wedge\ startcomp(\vec{ss}_2, \{\tau\}, \vec{ss}'_2) \\ \wedge & update(\vec{ss}_1 \,|^{CS}_{\varnothing}, \{\tau\}, \vec{ss}'_1) \ \wedge\ communicate\_vars((\vec{ss}_1, \vec{ss}_2), (\{\tau\}, \varnothing), (\vec{ss}'_1, \vec{ss}'_2)) \end{array} \right] \ \text{(* transition to component 2 *)} \\
\vee \\
\qquad \text{(* symmetric cases of the above two cases, replacing 1 with 2 and 2 with 1 *)}
\end{array}
$$

Fig. 8. Semantics of interrupt semantics for microsteps.

TABLE 8
Template Parameters and Compositions Operators for Notations

Parameter	CCS	CSP	Basic LOTOS	BTS	SDL88 (processes)	statecharts [16]	RSML	STATEMATE
reset_CS(ss, I)	$ss.CS$					$ss.CS$		
next_CS(ss, τ, CS')	$CS' = dest(\tau)$					$CS' = entered\ (dest(\tau))$		
reset_CS$_a(ss, I)$	n/a					$ss.CS$	n/a	n/a
next_CS$_a(ss, \tau, CS'_a)$	n/a					$CS'_a = \varnothing$	n/a	n/a
en_states(ss, τ)	$src(\tau) \subseteq ss.CS$					$src(\tau) \subseteq ss.CS_a$	$src(\tau) \subseteq ss.CS$	
reset_IE(ss, I)	n/a		n/a	n/a	$ss.IE \frown I$	$\varnothing$		
next_IE(ss, τ, IE')	n/a		n/a	n/a	$IE' = tail(ss.IE) \frown gen(\tau)$	$IE' = ss.IE \cup gen(\tau)$	$IE' = gen(\tau)$ $\cap\ intern\_ev(E)$	$IE' = gen(\tau)$
reset_IE$_a(ss, I)$	n/a		n/a	n/a	n/a	n/a	n/a	n/a
next_IE$_a(ss, \tau, IE'_a)$	n/a		n/a	n/a	n/a	n/a	n/a	n/a
reset_I$_a(ss, I)$	I		I	n/a	n/a	I		
next_I$_a(ss, \tau, I'_a)$	$I'_a = ss.I_a \cup gen(\tau)$		true	n/a	n/a	$I'_a = ss.I_a$	$I'_a = \varnothing$	
en_events(ss, τ)	$trig(\tau) \subseteq ss.I_a$		$trig(\tau) \subseteq ss.I_a$	n/a	$trig(\tau) = \{head(ss.IE)\}$	$trig(\tau) \subseteq ss.IE \cup ss.I_a$		
reset_O(ss, I)	$\varnothing$		n/a	n/a	$\varnothing$			
next_O(ss, τ, O')	$O' = gen(\tau)$		n/a	n/a	$O' = ss.O \cup\ gen(\tau)$		$O' = ss.O \cup (gen(\tau)$ $\cap\ extern\_ev(E))$	$O' = gen(\tau)$
reset_AV(ss, I)	n/a			$ss.AV$				
next_AV(ss, τ, AV')	n/a			$AV' = assign(ss.AV, eval(ss.AV, asn(\tau)))$		$AV' = assign(ss.AV,$ $eval((ss.AV, ss.AV_a),$ $asn(\tau)))$	$AV' = assign(ss.AV,$ $eval(ss.AV,$ $asn(\tau)))$	$AV' = assign(ss.AV,$ $eval(ss.AV,$ $last(asn(\tau))))$
reset_AV$_a(ss, I)$	n/a			n/a	n/a	$ss.AV$	n/a	n/a
next_AV$_a(ss, \tau, AV'_a)$	n/a			n/a	n/a	$AV'_a = ss.AV_a$	n/a	n/a
en_cond(ss, τ)	n/a			$ss.AV \models cond(\tau)$		$ss.AV, ss.AV_a \models cond(\tau)$	$ss.AV \models cond(\tau)$	
macro_semantics	simple diligent			simple non-diligent	stable	stable	stable	stable
pri(Γ)	no priority			no priority	no priority	no priority	no priority	lowest-ranked scope
resolve (vv$_1$,vv$_2$,vv)	n/a			n/a	n/a	n/a	n/a	resolve(vv$_1$, vv$_2$, vv)
Parallel	n/a			n/a	process composition $(N_{macro}^{para-SDL})$	AND-state composition $(N_{micro}^{para-Harel})$	AND-state composition (N_{micro}^{para})	
Environmental-sync	$a \rightarrow P \mid\mid a \rightarrow Q$		$P \mid [a, b, c] \mid Q$	n/a	n/a	n/a	n/a	n/a
Rendezvous-sync	$a.P \mid \bar{a}.Q$	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Interleaving	n/a	$P \mid\mid\mid Q$	$P \mid\mid\mid Q$	interleaving (macro)	n/a	n/a	n/a	n/a
Sequence	$P; Q$	$P; Q$	$P \gg Q$	concatenation (;)	n/a	n/a	n/a	n/a
Choice	$P + Q$	$P[\,]Q$	$P[\,]Q$	selection (OR)	n/a	n/a	n/a	n/a
Interrupt	n/a	n/a	n/a	n/a	n/a	OR-state composition		

*("*n/a*" means "not applicable")*

template composition operators. This description extends the information previously provided in Tables 3, 4, 5, and 6. In this section, we highlight some of the results in Table 8.

We handle CSP's parallel (||), interleaving (|||), sequential composition (;), and general choice ([ ]) operators. We have not yet formalized CSP's interrupt (^) operator; it would be a modification of our sequence composition, where the second component begins whenever the second component is enabled rather than waiting for the first component to terminate.

We can express LOTOS's parallel ($P \mid [a, b, c] \mid Q$), pure interleaving (|||), sequential composition ($\gg$), and choice ([ ]) operators. We do not yet handle LOTOS's disabling ([>) construct; it would be a variant of our sequence composition.

In SDL template semantics, each process (HTS) has its own *input queue* that stores unprocessed events, both internal and external. At the start of each macrostep, function $reset\_IE$ appends the new external events to the end of the event queue (using concatenation operator $\frown$). Only the event at the $head$ of the input queue is an enabling event. To record the effects of executing a transition, predicate $next\_IE$ removes from the input queue the $head$ event (by accessing the $tail$ of the queue) and appends the events that the transition generates. (There exist implicit transitions whose sole effect is to remove from the input queue the head event, if the event does not enable an explicitly specified transition.) SDL has stable macrostep semantics (because an SDL transition that contains decision points maps to a tree of HTS transitions, each of which executes in a microstep). SDL processes are composed using macrolevel parallel composition. In [29], we used template semantics to describe SDL blocks and channel communication. Template semantics do not support dynamic process creation or ACT ONE data definitions (used in both SDL and LOTOS).

A statechart with only OR-states is an HTS. Statecharts' AND-states are formed using our parallel composition. OR-state composition of components that contain AND-states corresponds to our interrupt composition. The three statecharts variants have stable macrostep semantics, but they have different microstep semantics, as shown in Table 8 and described in Section 2.3. STATEMATE is the only notation in Table 8 that permits conflicting variable assignments, thus needing the $resolve$ template parameter; the conflicts are resolved nondeterministically:

$$\begin{aligned} &resolve_{stm}(vv_1, vv_2, vv) \equiv \\ &\quad vv \subseteq vv_1 \cup vv_2 \ \wedge\ dom(vv) = dom(vv_1 \cup vv_2) \wedge \\ &\quad (\forall (a, b) \in vv \cdot \forall (c, d) \in vv\ .\ a = c \Longrightarrow b = d). \end{aligned}$$

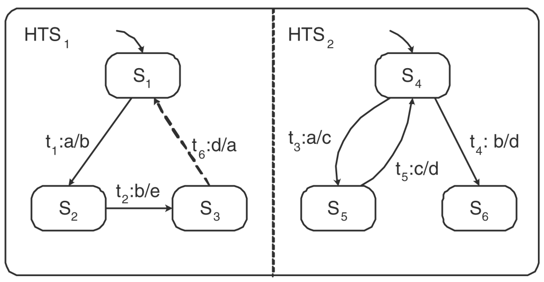

Fig. 9. A statecharts example.

We use a simple example (Fig. 9) to show how the statecharts variants' semantics admit different macrosteps in a specification. Consider the example statechart without the dotted transition t_6 first. The two orthogonal states are mapped to two HTSs (HTS_1 and HTS_2) composed with parallel composition operators: Harel's AND-state matches $N^{para-Harel}_{micro}$ and AND-states in RSML and in STATEMATE match N^{para}_{micro}. Tables 9 and 10 show the admissible macrosteps from the components' default states, given input event a.

In Harel's semantics, there are four admissible macrosteps, because when two AND-components are both enabled, Harel's parallel composition operator nondeterministically chooses whether one or both of the components executes in a microstep. If both components execute in the first microstep, the snapshot reaches a stable state (because the set of enabling states in both components becomes empty). If only t_1 in HTS_1 executes in the first microstep, its generated event enables t_4, and the persistent external event a enables t_3 in the second microstep. After HTS_2 executes one of these two transitions in the second microstep, no more transitions are enabled.

In contrast to Harel's statecharts, both RSML and STATEMATE's parallel composition requires each HTS to execute if it is enabled; thus, both t_1 and t_3 execute in the first micro step. The enabling states are the current states CS. Therefore, both t_2 and t_5 are enabled in and execute in the second microstep, after which both HTSs are in stable snapshots. If we assume that e is the only external event, then RSML and STATEMATE differ only in their outputs (O). If we add transition t_6 to HTS_1, the composed machine in both notations will have an infinite macrostep.

To ensure that our template representations of these notations' semantics correspond to existing representations of their semantics, we would have to prove the correspondence between the two semantics definitions. For many notations (except process algebras), it is difficult to find a complete and consistent definition of semantics. When such a definition is available, the proof would show the correspondence of a step in the two semantics. The proof would proceed by structural induction, showing the correspondence of micro and/or macrosteps at the HTS level as a base case, and then showing that each of the composition operators preserves this correspondence. Potentially, the most challenging part of the proof is determining what the correspondence should be because the two semantics may involve very different notions of snapshots. A further complication is that many semantic representations assume certain well-formedness properties of the notation (e.g., transition triggers are mutually

TABLE 9
The Possible Macrosteps of Harel's Statecharts

Macro Step	HTS_1				HTS_2				HTS_1 para HTS_2	
	enabled trans	*executed trans*	CS'	O'	*enabled trans*	*executed trans*	CS'	O'	*stable*	*executed trans*
microstep-1	$\{t_1\}$	t_1	$\{s_2\}$	$\{b\}$	$\{t_3\}$	t_3	$\{s_5\}$	$\{c\}$	*false*	$\{t_1, t_3\}$
	$\varnothing$				$\varnothing$				*true*	
microstep-1	$\{t_1\}$	t_1	$\{s_2\}$	$\{b\}$	$\{t_3\}$		$\{s_4\}$	$\varnothing$	*false*	$\{t_1\}$
microstep-2	$\varnothing$		$\{s_2\}$	$\{b\}$	$\{t_3, t_4\}$	t_3	$\{s_5\}$	$\{c\}$	*false*	$\{t_3\}$
	$\varnothing$				$\varnothing$				*true*	
microstep-1	$\{t_1\}$	t_1	$\{s_2\}$	$\{b\}$	$\{t_3\}$		$\{s_4\}$	$\varnothing$	*false*	$\{t_1\}$
microstep-2	$\varnothing$		$\{s_2\}$	$\{b\}$	$\{t_3, t_4\}$	t_4	$\{s_6\}$	$\{d\}$	*false*	$\{t_4\}$
	$\varnothing$				$\varnothing$				*true*	
microstep-1	$\{t_1\}$		$\{s_1\}$	$\varnothing$	$\{t_3\}$	t_3	$\{s_5\}$	$\{c\}$	*false*	$\{t_3\}$
microstep-2	$\{t_1\}$	t_1	$\{s_2\}$	$\{b\}$	$\varnothing$		$\{s_5\}$	$\{c\}$	*false*	$\{t_1\}$
	$\varnothing$				$\varnothing$				*true*	

TABLE 10
The Possible Macrosteps of STATEMATE

Macro Step	HTS_1				HTS_2				HTS_1 para HTS_2	
	enabled trans	*executed trans*	CS'	O'	*enabled trans*	*executed trans*	CS'	O'	*stable*	*executed trans*
microstep-1	$\{t_1\}$	t_1	$\{s_2\}$	$\{b\}$	$\{t_3\}$	t_3	$\{s_5\}$	$\{c\}$	*false*	$\{t_1, t_3\}$
microstep-2	$\{t_2\}$	t_2	$\{s_3\}$	$\{e\}$	$\{t_5\}$	t_5	$\{s_4\}$	$\{d\}$	*false*	$\{t_2, t_5\}$
	$\varnothing$				$\varnothing$				*true*	

Outputs of RSML

HTS_1	HTS_2
O'	O'
$\varnothing$	$\varnothing$
$\{e\}$	$\varnothing$

exclusive), which allows them to specialize their description of the notation's semantics; to be useful in the proof, these assumptions have to be identified, formalized, and translated into constraints on the semantics of the template representation of the notation.

5 Generality

Many sophisticated notations include features beyond simple states, events, and variables to help specifiers to structure and to simplify their specifications. This section discusses three ways in which to incorporate a language feature into a notation's template semantics; it concludes by characterizing the limitations on what types of language features can be defined using template semantics.

5.1 Syntactic Transliteration

Some language features are simply notational conveniences or "syntactic sugar" that have no effect on the notation's expressiveness, but enable the specifier to state certain ideas more succinctly or more clearly. Examples of such conveniences include macros, compound transitions, AND/OR tables [22], actions associated with entering and exiting states [16], event hiding and event relabeling [27], SDL's save construct, structured variables, and parameterized machines. We accommodate these features by working with the features' expanded definitions (e.g., expanding macros, separating compound transitions, instantiating a finite number of parameterized machines). In doing so, we accommodate these features entirely during the syntactic transliteration from the specification's syntax into HTS syntax; the features have no representation in the notation's semantics or composition operators.

5.2 Template Parameters

Language features that affect the sets of enabling states, enabling events, or enabling variables, or that affect a transition's effects on states, events, or variables can be accommodated using the auxiliary snapshot elements and the template parameters, without changing the definitions of the template's common semantics. Examples of such features include negated events and other event expressions, event parameters, point-to-point communication, history states, Maggiolo-Schettini et al.'s compatible transitions [23], and event queues [21], [30].

For example, to handle **negated events**, we distinguish between positive trigger events, denoted as $pos(\tau)$, and negated trigger events (i.e., lack of an event), denoted as $neg(\tau)$. Snapshot element I_a records the external events, IE accumulates the events that occur in the macrostep, and predicate en_events ensures that a transition is enabled only if its trigger events have occurred and its negated trigger events have not occurred:

$$\begin{aligned}&en\_events(ss,\tau) \equiv\\ &\quad (pos(\tau) \subseteq ss.IE \cup ss.I_a) \wedge (neg(\tau) \cap (ss.IE \cup ss.I_a) = \emptyset).\end{aligned}$$

Maggiolo-Schettini et. al. [23] have a stronger definition of enabling events that prohibits two transitions from executing in the same macrostep if one is triggered by negated event *not a* and a subsequent transition in the macrostep generates a; they call these *incompatible transitions*. To model these semantics, we use IE_a to accumulate the negated events that trigger transitions in the macrostep [29]. Subsequent transitions are enabled only if their actions are consistent with this set IE_a:

$$\begin{aligned}reset\_IE_a(ss,I) &\equiv \emptyset\\ next\_IE_a(ss,\tau,IE'_a) &\equiv IE'_a = ss.IE_a \cup neg(\tau)\\ en\_events(ss,\tau) &\equiv (pos(\tau) \subseteq ss.IE \cup ss.I_a) \wedge\\ &\qquad (neg(\tau) \cap (ss.IE \cup ss.I_a) = \emptyset) \wedge\\ &\qquad ((neg(\tau) \cup ss.IE_a) \cap gen(\tau) = \emptyset).\end{aligned}$$

As a more complicated example, we show how to accommodate statecharts history. Briefly, **history** is a mechanism by which a reentered superstate can continue executing from the substate that was current when control last transitioned out of the superstate. To accommodate history, we partition the set of states S into *basic states*; *history states*, denoted by Ⓗ; *deep-history states*, denoted by $\textcircled{H}^*$; and *superstates*. If a transition's destination is a history state Ⓗ, then the transition enters the most recently current substate in Ⓗ's parent state. If a transition's destination is a deep-history state $\textcircled{H}^*$, then *enter-by-history* applies not only to $\textcircled{H}^*$'s parent state but also to all of the parent's descendents.

We use auxiliary variable CS_a (or extend CS_a to have multiple data fields, if it is already being used to collect other state-related information) to record for each superstate the most recent substate that the superstate entered:

$$\begin{aligned}&reset\_CS_a(ss,I) \equiv CS'_a = ss.CS_a\\ &next\_CS_a(ss,\tau,CS'_a) \equiv\\ &\qquad CS'_a = ss.CS_a \oplus \{(parent(s),s) \mid s \in entered(\tau)\},\end{aligned}$$

where $\oplus$ is a function-override operator that updates its first operand with new elements and new functional mappings from the second operand. Specifically, each time a state s is entered, history information CS_a is updated to map s's parent state with its newly entered substate s. This history information is unchanged at the start of a macrostep. At the start of the system's execution, CS_a maps superstates to their default initial states.

Accommodating history states also changes how the set of current states is determined after a transition executes. Without history states, the set of current states CS after executing transition τ is the set of states entered by τ:

$$next\_CS(ss,\tau,CS') \equiv CS' = entered(dest(\tau)).$$

In hierarchical systems, the definition of $entered(dest(\tau))$ is nontrivial: The set of entered states includes not only τ's destination state $dest(\tau)$, but also all of $dest(\tau)$'s ancestor states plus the default states of $dest(\tau)$ and of its entered descendents. Adding history states changes this definition to include cases in which τ's destination is a history state. We define $entered(CS_a, s)$ as a fixed-point definition; the definition uses functions $super(S)$, $hist(S)$, and $hist^*(S)$ to identify superstates, history states, and deep-history states, respectively:

$$entered(CS_a, s) \equiv$$

$$\mu Y \,.\, \left[\begin{array}{l} s \cup ancest(s) \cup \\ \bigcup_{u\in Y} \left\{ t \;\middle|\; \begin{array}{l} (u \in hist(S) \wedge CS_a(parent(u))=t) \vee \\ (\exists r \in Y.[r \in hist^*(S) \wedge u \in descend(parent(r))] \wedge \\ \quad u \in super(S) \wedge CS_a(u)=t) \vee \\ (\neg\exists r \in Y.[r \in hist^*(S) \wedge u \in descend(parent(r))] \wedge \\ \quad u \in super(S) \wedge (child(u) \cap Y=\emptyset) \wedge default(u)=t) \end{array} \right\} \end{array}\right].$$

That is,

- The state s is entered.
- Any ancestor of s is entered.
- If entered state u is a history state $\textcircled{H}$, then its parent's most recently current substate is entered.
- If entered superstate u descends from the parent of an entered deep-history state $\textcircled{H}^*$, then s's most recently current substate is entered.
- If entered state u is a superstate not descended from the parent of an entered deep-history state, and if none of u's substates is designated as entered, then u's default state is entered.

The fixed-point computation terminates when it finds no more new states to be entered.

Predicate *ent_comp*, which was introduced by the interrupt composition operator, is similar to the above definition of *entered*. Predicate *ent_comp* includes an additional case: If entered state u is an AND-state, then u's sibling states are also entered.

5.3 Template Extension

Some language features fit within state-transition semantics but are orthogonal to states, events, and variables. Examples of such features include real-time conditions and constraints, or any alternative enabling condition on transitions, assertable and retractable constraints, dynamic creation/destruction of processes, and inherited and polymorphic behaviors. We cannot accommodate these features without extending the template, which means adding new elements to the snapshot, adding new template parameters, or extending the template definitions. Fortunately, such extensions are *incremental*, in that they can be appended to the template definitions without overriding the existing definitions.

Consider SDL timers, which are set and reset by transitions and which generate events when the timers time out. This construct requires extensions to the HTS syntax, to the set of snapshot elements, to the set of template parameters, and to a subset of the template definitions.[1] The HTS syntax is extended to include a set of clocks C and to include SET(t,a) and RESET(a) as allowable transition actions in $asn(\tau)$ for a transition τ. The snapshot is extended to include two new elements, TM, representing the current absolute time, and $AL : C \rightarrow Integer$, the set of activated timers (alarm clocks) and their respective settings. The template is extended to include new template parameters for updating TM and AL at the start of each macrostep and at the end of each microstep, respectively. In addition, a template-extending language feature may also affect the values of existing template parameters, as in the case of SDL timers, which affect the event-related parameters: When a timer times out, a signal is generated and is inserted into the process's event queue:

$$reset\_TM(ss, I) \equiv I.Time$$
$$next\_TM(ss, \tau, TM') \equiv ss.TM = TM'$$
$$reset\_AL(ss, I) \equiv ss.AL \ominus \{(a,t) | (a,t) \in ss.AL \wedge t \leq I.Time\}$$
$$\begin{aligned} next\_AL(ss, \tau, AL') \equiv \\ AL' = ss.AL \oplus \{(a,t) | SET(t,a) \in asn(\tau)\} \ominus \\ \{(a,t) | (a,t) \in AL \wedge RESET(a) \in asn(\tau)\} \end{aligned}$$
$$\begin{aligned} reset\_IE(ss, I) \equiv \\ ss.IE^\frown(I.Events)^\frown \underline{\{a \mid (a,t) \in ss.AL \wedge t \leq I.Time\}} \end{aligned}$$
$$\begin{aligned} next\_IE(ss, \tau, IE') \equiv \\ IE' = \underline{[remove(}tail(ss.IE), \\ \underline{\{a | RESET(a) \in asn(\tau)\})]}^\frown(gen(\tau)), \end{aligned}$$

where $\oplus$ is a function-override operator that updates its first operand with new elements and new functional mappings, $\ominus$ is a set difference operator, and $\frown$ concatenates two sequences. In the above definitions, system time TM is changed only by sensing a new time from the environment; microsteps do not affect the passage of time. An alarm clock will time-out at the start of a macrostep if time TM reaches or exceeds its setting; if an alarm clock a times out, the signal a is appended to the process's event queue by template parameter $reset\_IE$, and the timer is removed from AL. A transition may SET an inactive timer, which causes the clock and its setting to be added to AL; it may SET an already active timer, which causes the timer entry in AL to be updated; or it may RESET an active timer, which causes the timer entry to be removed from AL. If an expired timer a is RESET when the event queue contains time-out signal a (i.e., if a timer times-out, but is RESET before its time-out signal is processed), this signal is removed from the event queue. The underlined clauses are those that are added to existing template-parameter definitions to accommodate SDL timers.

Of the six template definitions (*enabled_trans*, *apply*, *reset*, *stable*, N_{micro}, and N_{macro}), only *apply* and *reset* need to be modified. Function *reset* is modified to include additional function calls $reset\_TM(ss, I)$ and $reset\_AL(ss, I)$, so that the new snapshot elements are appropriately reset at the start of a macrostep; and predicate *apply* is modified to include additional conjuncts $next\_TM(ss, \tau, TM')$ and $next\_AL(ss, \tau, AL')$, so that the new snapshot elements are appropriately updated after the execution of every transition τ.

If timing conditions were a new way to enable transitions, then in addition to the above extensions, a new *enabling* template parameter would be needed, and the template definition *enabled_trans* would be modified to include this predicate as an additional conjunct. Because such modifications involve only appending new clauses to

1. SDL timers could be modeled as HTS variables, by stretching the notion of what a variable is. However, we model SDL timers as a new syntactic and semantic construct, to show how to extend the template to incorporate new constructs.

the existing template definitions, we say that these template extensions are incremental.

5.4 Limitations

Language features whose steps are coarser- or finer-grained than our microsteps and macrosteps, or are features that modify the snapshot in ways that cannot be described by transitions, cannot be described using template semantics. Examples of such features include operations, methods, and statecharts activity states (all of which can have their own triggering conditions and can span multiple macrosteps), features that need to observe future snapshots (e.g., Pnueli and Shalev's nonfailure global consistency semantics requires advanced knowledge of all the transitions executing in the macrostep [34]), and continuous-time behavior. Accommodating such features would at best require major modifications to the template beyond appending clauses to existing definitions and, thus, lie outside the scope of our template-semantics framework.

6 Related Work

To the best of our knowledge, there has been no comparable attempt to classify formally the step-semantics and composition semantics for model-based specifications. There has been work on informally classifying the semantics of specifications languages [10], the most famous of which is von der Beeck's comparison of statecharts variants [35]. These catalogues of composition operators (parallel, interleaving, etc.) and communication operators (synchronous, asynchronous, etc.) are similar to ours, but we go further and define formally how each operator affects a model's behavior. We also express variations in step-semantics as parameters, which makes it easier to define new notations and to highlight subtle differences among notations' semantics.

There has been substantial related work on formalizing the semantics of individual specification languages, such as defining the operational semantics of LOTOS [20]. Usually, the purpose of such work is to document a language's precise semantics, possibly as a first step towards developing verification tools. Such formalizations tend to be language specific, making it difficult to compare the semantics of different languages and difficult to generalize the semantics or resulting tools to accommodate multiple languages.

A number of researchers have proposed translating specification notations into more fundamental modeling notations, such as first-order logic [36], hierarchical state machines [26], labeled transition systems [6], and hybrid automata [3]. Such notations are general enough to represent a variety of specification notations and can even accommodate specifications written in multiple notations. The verification tools and techniques associated with the more fundamental modeling notation can be applied to the translated specification. Degano and Priami proposed general semantic models that are capable of capturing the structured operational semantics of notations by enhancing the transition labels [14]. In Rosetta [1], multiple views of a system are composed using different kinds of composition operators that support renaming and instantiation.

Researchers have introduced *intermediate languages*, such as SAL [4], IF [7], and Action Language [8], that are expressive target languages that ease translations between notations. In the case of IF and SAL, there exist translators between several specification notations and the intermediate language, between the intermediate language and the input languages of several verification tools, and vice versa. These approaches allow the specification to be analyzed using multiple verification tools. However, a translator needs to be built for each specification notation and needs to be continuously modified as the notation's semantics evolves. With our approach, we hope to reduce substantially the effort involved in updating analysis tools when a notation changes because we have decomposed the definition of the semantics into smaller, more cohesive concerns that are likely to change independently of one another.

Work with similar goals to ours is that of Day and Joyce [13], Pezzè and Young [33], and Dillon and Stirewalt [15]. The goal of these works is to generate analysis tools automatically from a description of a notation's semantics. Day and Joyce embed the semantics of a notation in higher-order logic and automatically compile a next-state relation from the notation's semantics and specification using symbolic functional evaluation. Embedding avoids the translation step and the effort to construct and maintain translators. Notations have also been embedded in the theorem prover PVS [31], and PVS's connection to a model checker has been used to analyze these specifications.

Pezzè and Young embed the semantics of model-based notations into hypergraph rules, which specify how enabled transitions are selected, and how executing transitions affect the specification's hypergraph model. The composition semantics of heterogeneous components can also be described.

Dillon and Stirewalt define operational semantics for process-algebra and temporal-logic notations and semiautomatically translate these semantic descriptions into a tool that accepts a specification and generates an *inference graph*. This inference graph calculates all of the specification's possible next snapshots, expressed as specifications, which, in turn, can be fed into the tool to produce their respective inference graphs. Their approach cannot accommodate model-based notations with data variables. Similarly, Cleaveland and Sims [12] have developed a semantics-based compiler for translating process algebra notations into CCS, the input language of the Concurrency Workbench [11].

In contrast to these approaches, our paper separates step-semantics and composition operators. This separation allows us to simplify each of these aspects of semantics, to the point where one can define the semantics of a new notation as parameter values rather than an embedding. Also, our work is expressed using traditional state-transition relations rather than introducing a new execution model.

7 Conclusions

We have introduced an operational-semantics template for model-based specification notations, which predefines behavior that is common among notations and parameterizes a notation's distinct semantics. We have defined a set of composition operators as the concurrent execution of components, with appropriate changes to snapshot elements to reflect inter-component communication and synchronization. Using this template, one can define the semantics of a new notation simply by 1) instantiating the template's parameters and by 2) defining how composition operators control components' executions and change snapshot elements.

We are able to express as instantiations of our template most of the semantics of eight popular specification notations: CSP [19], CCS [27], LOTOS [20], basic transition systems (BTS) [24], a subset of SDL88 [21], and three variants on statecharts [16], [17], [22]. Template-semantics definitions for SDL blocks and channel communications [21], SCR [18], Petri Nets [32], and additional statecharts variants and aspects of UML appear elsewhere [29].

There are many model-based notations that we have not yet used template semantics to describe (e.g., value-passing CCS [27]). As explained in Section 5, many specification features can be represented using the existing HTS syntax and template parameters. Support for additional features may involve adding parameters and snapshot elements, but, as is the case for supporting SDL timers, if the additions are orthogonal to the original template model, they may be accommodated by appended clauses to the template definitions without overriding the existing definitions. Template semantics provide enough flexibility that the addition of nonorthogonal features, such as history states, can usually be incorporated just by changing the parameter definitions.

We plan to implement our template definitions, to generate automatically notation-specific, formal analysis tools from the description of a notation's template semantics. We are particularly interested in *model-compiler generators*, where a **model compiler** is a program that *compiles* a specification into a more primitive representation. Example target computation models include Kripke structures, binary decision diagrams (BDDs), and logic formulae. The use of template semantics should substantially reduce the effort involved in creating an analysis tool for a model-based notation.

ACKNOWLEDGMENTS

The authors would like to thank Dan Berry, Yun Lu, Andrew Malton, and John Thistle of the University of Waterloo for feedback on early versions of this work. They also thank Mats Heimdahl and Michael Whalen of the University of Minnesota for discussions on RSML semantics and the reviewers for many helpful comments.

REFERENCES

[1] P. Alexander and C. Kong, "Rosetta: Semantic Support for Model-Centered Systems-Level Design," *Computer,* vol. 34, no. 11, pp. 64-70, 2001.

[2] J.M. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *IEEE Trans. on Software Eng.,* vol. 19, no. 1, pp. 24-40, 1993.

[3] G. Avrunin, J. Corbett, and L. Dillon, "Analyzing Partially-Implemented Real-Time Systems," *Proc. Int'l Conf. Software Eng.,* pp. 228-238, 1997.

[4] S. Bensalem et al., "An Overview of SAL," *Proc. Langley Formal Methods Workshop,* pp. 187-196, 2000.

[5] G. Berry and G. Gonthier, "The Synchronous Programming Language ESTEREL: Design, Semantics, Implementation," *Science of Computer Programming,* vol. 19, no. 2, pp. 87-152, 1992.

[6] L. Blair and G. Blair, "Composition in Multi-Paradigm Specification Techniques," *Proc. Int'l Workshop Formal Methods for Open Object-based Distributed Systems,* pp. 401-417, 1999.

[7] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J. Krimm, L. Mounier, and J. Sifakis, "IF: An Intermediate Representation for SDL and Its Applications," *Proc. SDL-Forum '99,* pp. 423-440, 1999.

[8] T. Bultan, "Action Language: A Specification Language for Model Checking Reactive Systems," *Proc. Int'l Conf. Software Eng.,* pp. 335-344, 2000.

[9] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin, "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.,* vol. 24, no. 7, pp. 498-519, July 1998.

[10] P. Chou and G. Borriello, "An Analysis-Based Approach to Composition of Distributed Embedded Systems," *Proc. Int'l Workshop Hardware/Software Co-Design,* 1998.

[11] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 1, pp. 36-72, 1993.

[12] R. Cleaveland and S. Sims, "Generic Tools for Verifying Concurrent Systems," *Science of Computer Programming,* vol. 41, no. 1, pp. 39-47, Jan. 2002.

[13] N.A. Day and J.J. Joyce, "Symbolic Functional Evaluation," *Proc. 12th Int'l Conf. Theorem Proving in Higher Order Logics,* Sept. 1999.

[14] P. Degano and C. Priami, "Enhanced Operational Semantics: A Tool for Describing and Analyzing Concurrent Systems," *ACM Computing Surveys,* vol. 33, no. 2, pp. 135-176, 2001.

[15] L. Dillon and R. Stirewalt, "Inference Graphs: A Computational Structure Supporting Generation of Customizable and Correct Analysis Components," *IEEE Trans. Software Eng.,* vol. 29, no. 2, pp. 133-150, Feb. 2003.

[16] D. Harel et al., "On the Formal Semantics of Statecharts," *Proc. Symp. Logic in Computer Science,* pp. 54-64, 1987.

[17] D. Harel and A. Naamad, "The Statemate Semantics of Statecharts," *ACM Trans. Software Eng. Methods,* vol. 5, no. 4, pp. 293-333, 1996.

[18] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. Methods,* vol. 5, no. 3, pp. 231-261, 1996.

[19] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[20] International Organization for Standardization, "LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior,"Technical Report ISO8807, 1988.

[21] International Telecommunications Union, "Recommendation Z.100. Specification and Description Language (SDL)},," Technical Report Z-100, Standardization Sector, 1999.

[22] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 9, pp. 684-707, Sept. 1994.

[23] A. Maggiolo-Schettini, A. Peron, and S. Tini, "Equivalences of Statecharts," *Proc. Int'l Conf. Concurrency Theory,* 1996.

[24] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.

[25] K. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic, 1993.

[26] E. Mikk, Y. Lakhnech, and M. Siegel, "Hierarchical Automata as Model for Statecharts," *Proc. Asian Computer Science Conf.,* 1997.

[27] R. Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[28] J. Niu, J.M. Atlee, and N.A. Day, "Composable Semantics for Model-Based Notations," *Proc. Int'l Symp. Foundations of Software Eng. (FSE-10),* pp. 149-158, 2002.

[29] J. Niu, J.M. Atlee, and N.A. Day, "Understanding and Comparing Model-Based Soecification Notations," *Proc. IEEE Int'l Requirements Eng. Conf.,* pp. 188-199, 2003.

[30] Object Management Group, *Unified Modelling Language (UML),* version 1.4, 2001, www.omg.org.

[31] S. Owre, J. Rushby, and N. Shankar, "Analyzing Tabular and State-Transition Requirements Specifications in PVS," Technical Report CSL-95-12, SRI Int'l, Computer Science Laborabory, 1995.

[32] J.L. Peterson, "Petri Nets," *ACM Computing Surveys,* vol. 9, no. 3, pp. 223-252, 1977.

[33] M. Pezzè and M. Young, "Constructing Multi-Formalism State-Space Analysis Tools," *Proc. Int'l Conf. Software Eng.*, pp. 239-249, 1997.
[34] A. Pnueli and M. Shalev, "What is in a Step: On the Semantics of Statecharts," *Proc. Symp. Theoretical Aspects of Computer Software,* 1991.
[35] M. von der Beeck, "A Comparison of Statecharts Variants," *Formal Techniques in Real Time and Fault-Tolerant Systems,* 1994.
[36] P. Zave and M. Jackson, "Conjunction as Composition," *ACM Trans. Software Eng. Methods,* vol. 2, no. 4, pp. 379-411, 1993.

Jianwei Niu received the BSc degree in computer science from Jilin University, Changchun, China. She is a PhD candidate in the School of Computer Science at the University of Waterloo. She is a member of the Waterloo Formal Methods (WatForm) research group. Her research interests include formal methods and requirements engineering. She is a student member of the ACM, ACM SIGSOFT, and the IEEE.

Joanne M. Atlee received the BS degree in computer science and physics from the College of William and Mary in 1985, and the MS and PhD degrees in computer science from the University of Maryland in 1988 and 1992, respectively. She is an associate professor in the School of Computer Science at the University of Waterloo, where she is a member of the Waterloo Formal Methods (WatForm) research group and is the director of software engineering. Her research interests include software requirements and specification, model checking, feature interactions, and software engineering education. She is a member of the ACM, ACM SIGSOFT, and the IEEE Computer Society.

Nancy A. Day received the MSc and PhD degrees in computer science from the University of British Columbia in 1993 and 1998 respectively, and the BSc degree in computer science from the University of Western Ontario in 1991. From 1998 to 2000, she was a postdoctoral research associate at the Oregon Graduate Institute. She is currently an assistant professor in the School of Computer Science at the University of Waterloo and is a member of the Waterloo Formal Methods (WatForm) research group. Her research interests include formal methods, requirements engineering, and hardware verification. She is a member of the ACM.