Composable Semantics for Model-Based Notations

Jianwei Niu
jniu@uwaterloo.ca
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada
N2L 3G1

Joanne M. Atlee
jmatlee@uwaterloo.ca
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada
N2L 3G1

Nancy A. Day
nday@uwaterloo.ca
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada
N2L 3G1

ABSTRACT

We propose a unifying framework for model-based specification notations. Our framework captures the execution semantics that are common among model-based notations, and leaves the distinct elements to be defined by a set of parameters. The basic components of a specification are non-concurrent state-transition machines, which are combined by composition operators to form more complex, concurrent specifications. We define the step-semantics of these basic components in terms of an operational semantics *template* whose parameters specialize both the enabling of transitions and transitions' effects. We also provide the operational semantics of seven composition operators, defining each as the concurrent execution of components, with changes to their shared variables and events to reflect inter-component communication and synchronization; the definitions of these operators use the template parameters to preserve in composition notation-specific behaviour. By separating a notation's step-semantics from its composition and concurrency operators, we simplify the definitions of both. Our framework is sufficient to capture the semantics of basic transition systems, CSP, CCS, basic LOTOS, ESTELLE, a subset of SDL88, and a variety of statecharts notations. We believe that a description of a notation's semantics in our framework can be used as input to a tool that automatically generates formal analysis tools.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; D.2.1 [**Requirements/Specifications**]: Languages; F.3.2 [**Semantics of Programming Languages**]: Operational semantics

General Terms

Languages, Verification

Keywords

Model-based notations, Operational Semantics, Composition, Concurrency, Communication, Formal analysis

SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

1. INTRODUCTION

Software practitioners find the semantics of model-based notations to be relatively intuitive, and their composition operators provide facilities for decomposing large problems into modules and for expressing concurrency, synchronization, and communication among modules. Examples of model-based notations include state-based notations (e.g., statecharts [15, 16], RSML [21]) and process algebras (e.g., CSP [17], CCS [24], LOTOS [18]).

There are many well-established analysis tools, such as model checkers, for automatically verifying model-based specifications. Either an analyzer is customized for a particular notation [9, 16] or a translator is written that maps the notation to the input language of one or more analyzers [1, 2, 7]. To reduce the number of translators, we can translate specifications into an intermediate language, from which we translate to the input languages for different tools [3, 6]. In all of these cases, an analyzer or translator needs to be written for each notation and rewritten whenever the notation or its semantics changes. To help ease this effort, we and others [10, 12, 27] are working toward generating analyzers and translators automatically from a description of a notation's semantics.

We propose an operational semantics *template* for model-based specification notations, in which the template captures the common behaviour among notations and parameterizes notations' distinct behaviours. The template represents a specification as a concurrent collection of non-concurrent hierarchical transition systems, each of whose operational semantics is its transition relation at both the micro-step and macro-step levels. An instantiation of this template specializes a notation's choice of which transitions occur in a given execution state. We define composition operators separately, as relations that constrain how collections of components can execute in parallel, transfer control to one another, and exchange events and data; the operators' definitions use the template parameters to specialize updates to the components' step-semantics variables. We are able to define most of the semantics of seven popular specification notations (CSP [17], CCS [24], LOTOS [18], basic transition systems (BTS) [22], ESTELLE [19], a subset of SDL88 [20], and several variants on statecharts [15, 16, 21]) as instantiated variants of our template. We have not attempted to handle all of the features of each notation (e.g., we does not address state actions, activities, or history in statecharts), but we believe that we could unify features that map to transition-relation semantics (e.g., we should be able to unify state actions and history, but probably not activities).

Our framework effectively isolates composition as a separate concern of a notation's execution semantics, thereby simplifying the definitions of both step-semantics and composition operators. A secondary effect is progress towards standardizing the semantics of model-based notations: by parameterizing the notations' dis-

tinct behaviour, it is easier to see precisely the semantic differences between two languages. Our main goal is to provide a means to describe a notation's semantics succinctly, so that it can be input to a tool or process that generates notation-specific analysis tools. We believe that our operational semantics template and pattern for defining composition operators may be such a description.

This paper is organized as follows. Section 2 presents the syntax and semantics for hierarchical transition systems (HTS), which is our model for basic components. In Section 3, we give operational semantics for several interesting composition operators. Section 4 shows how the semantics of different model-based notations can be described as instantiations of our framework. Related works are discussed in Section 5, and we conclude in Section 6.

2. HIERARCHICAL TRANSITION SYSTEM

In this section, we introduce hierarchical transition systems (HTS) to represent model-based notations minus their composition operators. An HTS is a hierarchical, extended finite state machine, adapted from basic transition systems [22] and statecharts [15, 16]. An HTS supports no concurrency; that is, in statecharts terminology, it supports OR-state hierarchy but not AND-state hierarchy. Concurrency is introduced by the composition operators, which are defined in the next section. We use HTSs to model the basic components of a composite model-based system.

2.1 Syntax of HTS

A hierarchical transition system (HTS) is an 8-tuple, $< S, H, I, F, E, V, V_I, T >$. S is a finite set of states, H is the state hierarchy, and each state $s \in S$ is either a basic state or a super state that contains other states. $I \subseteq S$ is the non-empty set of initial states. $F \subseteq S$ is the set of final basic states. No transition can exit a final state. E is a finite set of events, including both internal and external events. We assume that event names are distinct across components. V is a finite set of data variables, with an initial value assignment of V_I. T is a finite set of transitions, each of which has the form,

$$< src, trig\_ev \wedge cond, act, dest, prty >$$

where $src, dest \in S$ are the transition's source and destination states, respectively; $trig\_ev \subseteq E$ is zero or more triggering events; *cond* is an optional 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 optional priority. We use identifiers S, H, I, F, E, V, V_I, T throughout the paper to refer to these HTS elements.

The state hierarchy consists of two kinds of states: super states and basic states. A **super state** is a state that contains other states, whereas a **basic state** contains no other states. Each super state has a default state, so that its default state is entered if the super state is the destination state of a transition. A state hierarchy H defines a partial ordering on states, with the root state of an HTS as the maximal element and basic states as minimal elements. For example, in the HTS in Figure 1, S_0, S_1, S_2, S_3 are super states, and the others are basic states. The top state S_0 is the root state of the HTS, and its default state is S_1. The function *rank* assigns a number to a state based on the HTS's hierarchy:

$$rank(s) = rank(parent(s)) + 1$$

where $rank(root) = 0$. In Figure 1, the rank of S_7 is 3.

To access the elements of a transition τ, we use helper functions:

- $source(\tau)$ is the source state of τ.
- $exited(\tau)$ are the states exited when τ executes, including the source's ancestor and descendant states that are also exited.
- $entered(\tau)$ are the states entered when τ executes, including the destination's ancestor and descendant states and all relevant default states that are also entered.

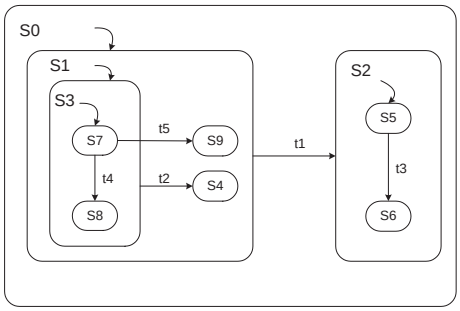

Figure 1: The state hierarchy of an HTS

- $ev\_trig(\tau)$ are the events that trigger τ.
- $ev\_gen(\tau)$ are the events generated by τ's actions.
- $asn(\tau)$ are variable-value assignments in τ's actions.
- $scope(\tau)$ is the lowest common ancestor state of the transition's source and destination states.
- $priority(\tau)$ is the transition's priority over other transitions.

We will also apply these functions to sets of transitions. Their meanings are the same, but those functions that return a single result (e.g., *source*) will return a set of results.

2.2 Semantics of HTS

We define the semantics of an HTS as a **snapshot relation**, which relates two snapshots ss and ss' if the system can move from ss to ss' in a step. Informally, a **snapshot** is an observable point in an HTS's execution. We define two types of steps between snapshots: a **micro-step** is an HTS's incremental response to some external event(s), and a **macro-step** is a sequence of zero or more micro-steps. At the beginning of each macro-step, a new set of external events $EE \subseteq E$ is sensed from the environment, and these events remain active for that macro-step.

We formally define *micro-step* and *macro-step* semantics as templates that are instantiated with specifier-provided *parameter functions* to reflect the semantics of a particular notation. We also populate the snapshot with a small set of history variables that the parameter functions can use and manipulate. Specifically, these history variables and parameter functions capture the semantic differences among notations, by parameterizing how to determine which transitions are enabled in a snapshot and how to select an enabled transition to execute. The parameter functions are also used in Section 3 to help define the semantics of composition operators.

2.2.1 Snapshots

A snapshot is an 8-tuple $< CS, IE, AV, EE_g, CS_h, IE_h, AV_h, EE_h >$. 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. $IE \subseteq E$ is the set of events generated by transitions executing in the previous micro-step. AV is a function that maps each data variable in V to its current value. EE_g is the set of external events that were generated by past transitions in the current macro-step; EE_g will not be important until we discuss composition operators. Elements CS_h, AV_h, IE_h and EE_h are history variables that accumulate data about the states, the variable values, and the internal and external events, respectively, that were used in past transitions. The template parameters use these history variables to derive the set of enabling states CS_e, the set of enabling variable values AV_e, and the sets of enabling internal events IE_e and external events EE_e, which in turn determine the transitions that are enabled in the current snapshot. External events EE are not part of the snapshot because they lie outside of the system. Instead, the

	Resulting Snapshot	Start of Macro-step		Next Micro-step
		$init_{\text{simple}}(ss, EE)$	$init_{\text{stable}}(ss, EE)$	$apply(ss, \tau)$
Fixed	$CS =$	$ss.CS$	$ss.CS$	$(ss.CS \setminus exited(\tau)) \cup entered(\tau)$
	$IE =$	$ss.IE$	$\varnothing$	$ev\_gen(\tau)$
	$AV =$	$ss.AV$	$ss.AV$	$assign(ss.AV, resolve\_conflicts(asn(\tau)))$
	$EE_g =$	$\varnothing$	$\varnothing$	$(ss.EE_g \cup ev\_gen(\tau)) \setminus ev\_trig(\tau)$
Parametric	$CS_h =$	$init\_states\_his(ss)$		$n\_states\_his(ss, \tau)$
	$IE_h =$	$init\_int\_ev\_his(ss, EE)$		$n\_int\_ev\_his(ss, \tau)$
	$AV_h =$	$init\_var\_val\_his(ss)$		$n\_var\_val\_his(ss, resolve\_conflicts(asn(\tau)))$
	$EE_h =$	$init\_ext\_ev\_his(ss, EE)$		$n\_ext\_ev\_his(ss, \tau)$
	$CS_e =$	$en\_states(ss)$		
	$IE_e =$	$en\_int\_ev(ss)$		
	$EE_e =$	$en\_ext\_ev(ss)$		
	$AV_e =$	$en\_var\_val(ss)$		

Table 1: Definitions of *init* and *apply* based on parameter functions. Sample parameter functions are shown in Tables 2-5.

snapshot maintains history variable EE_h, which helps to define how long external events remain enabling events. If a notation does not need some snapshot element (e.g., some process algebras have no variables), then the template parameters must be defined so that the extraneous element does not affect the enabling of transitions.

2.2.2 Micro-step Semantics

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

$$N_{micro}(ss, ss', \tau) = \\ \tau \in pri(enabled\_trans(ss, T)) \wedge ss' = apply(ss, \tau)$$

where

- The function *enabled_trans* returns the set of transitions that are enabled in a snapshot ss:

$$enabled\_trans(ss, T) = \\ \{\tau \in T \mid \\ \tau =< src, trig\_ev \wedge cond, act, dest, prty >, \\ \text{where } (src \in CS_e) \ \wedge \ (trig\_ev \subset IE_e \cup EE_e) \\ \wedge \ (AV_e \models cond)\}$$

  CS_e, IE_e, EE_e, and AV_e are computed by the parameter functions listed in the last four rows of Table 1; each is described in more detail below in a separate sub-section. $x \models y$ means condition y is true given variable assignments x.
- The function *pri* finds the maximal subset of transitions with the highest relative priority. Function *pri* is a template parameter and is also described in more detail below. If *pri* returns a set with more than one transition, then one transition τ is nondeterministically selected from this set to execute.
- The function *apply* determines the next snapshot based on a current snapshot ss and a transition τ; it is defined in the first eight rows of the column labelled "Next Micro-step" of Table 1.

The rows labelled "Fixed" of Table 1 define the functions for snapshot elements CS, IE, AV, and EE_g; these definitions apply to all notations. The rows labelled "Parametric" in Table 1 list the parameter functions that a specifier provides to instantiate the step-semantics of his or her model-based notation.

The functions in the column labelled "Start of Macro-step" of Table 1 are used at the beginning of a macro-step. These functions clear accumulated information about transitions that executed in the previous macro-step. In Section 2.2.3, we distinguish between two different kinds of macro-steps: simple and stable.

The definition of *apply* used for calculating the next micro-step depends on both fixed functions and parameter functions. In the snapshot resulting from an application of *apply*, the set of current states is updated according to the states that the executing transition τ exits and enters. The set of internal events is exactly those events that were generated by the previous transition. Variable values are updated by function $assign(X, Y)$, which takes two variable-value assignments X and Y and updates the assignments in X with the assignments in Y. Any assignments in Y to variables not in X are ignored. The function *resolve_conflicts* resolves any conflicts arising from multiple assignments to the same variable. The set of generated external events accumulates events that are generated by transitions executing in the current macro-step, minus any event that subsequently triggers a transition within the same macro-step (thereby making the event an internal event).

The parameter functions specialize the sets of states, events, and variables values that can *enable* transitions and define how these sets change as an HTS executes. For example, three parameter functions work together to determine which states can enable transitions:

- *en_states* computes the set of enabling states CS_e, using information in current states CS and/or history variable CS_h
- *init_states_his* resets CS_h at the beginning of a macro-step
- *n_states_his* updates CS_h after a micro-step

There are similar parameter functions for each of the other enabling elements (internal events, external events, and variable values). There are thirteen (13) parameter functions in total, the twelve functions listed in Table 1 plus the priority function *pri*.

The following five sub-sections describe how a specifier can use these parameter functions to define the step-semantics for some popular notations. Tables 2–6 provide example definitions. The abbreviations "n/a" means "not applicable".

Enabling States

Table 2 defines parameter functions for deriving the set of enabling states. In RSML and STATEMATE, the enabling states are always the current states. Unfortunately, these semantics allow infinite loops in a macro-step. Harel et al.'s [15] original formulation of statecharts avoids infinite loops by monotonically decreasing the set of enabling subsets: at the beginning of each macro-step, CS_h is initialized to the set of current states CS, and with every transition the sets of exited and entered states are removed from CS_h. We can

Notations	$init\_states\_his(ss) =$	$n\_states\_his(ss, \tau) =$	$en\_states(ss) =$
STATEMATE, RSML, LOTOS, SDL, CCS, CSP, BTS	n/a	n/a	CS
statecharts [15]	CS	$CS_h \setminus (exited(\tau) \cup entered(\tau))$	CS_h

Table 2: Sample Definitions for State Semantics. CS and CS_h mean $ss.CS$ and $ss.CS_h$ respectively.

Notations	$init\_int\_ev\_his(ss, EE) =$	$n\_int\_ev\_his(ss, \tau) =$	$en\_int\_ev(ss) =$
STATEMATE, RSML LOTOS, CSP, CCS	n/a	n/a	IE
statecharts [15]	$\emptyset$	$IE_h \cup ev\_gen(\tau)$	IE_h
SDL	$append(IE_h, EE)$	$append(remove(IE_h, \{IE_h[1], .., IE_h[j]\}), ev\_gen(\tau))$, where $j = min(i).\ \{IE_h[i]\} = IE_e$	$\{IE_h[j] \mid j = min(i).\ \exists\, \tau \in T.\ IE_h[i] = ev\_trig(\tau) \wedge source(\tau) \in CS_e\}$

Table 3: Sample Definitions for Internal Event Semantics. IE and IE_h mean $ss.IE$ and $ss.IE_h$ respectively.

Notations	$init\_ext\_ev\_his(ss, EE) =$	$n\_ext\_ev\_his(ss, \tau) =$	$en\_ext\_ev(ss) =$
STATEMATE,RSML, LOTOS, CSP, CCS	EE	$\emptyset$	EE_h
statecharts [15]	EE	EE_h	EE_h

Table 4: Sample Definitions for External Event Semantics. EE and EE_h mean $ss.EE$ and $ss.EE_h$ respectively.

Notations	$init\_var\_val\_his(ss) =$	$n\_var\_val\_his(ss, a) =$	$en\_var\_val(ss) =$
STATEMATE, RSML, SDL, BTS	n/a	n/a	AV
statecharts [15]	AV	AV_h	AV_h

Table 5: Sample Definitions for Variable Value Semantics. AV and AV_h mean $ss.AV$ and $ss.AV_h$ respectively.

Priority Scheme	$pri(\Gamma) =$
scope_O	$\{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(scope(\tau)) \leq rank(scope(t))\}$
scope_I	$\{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(scope(\tau)) \geq rank(scope(t))\}$
source_O	$\{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(source(\tau)) \leq rank(source(t))\}$
source_I	$\{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ rank(source(\tau)) \geq rank(source(t))\}$
explicit	$\{\tau \in \Gamma \mid \forall\, t \in \Gamma.\ priority(\tau) \leq priority(t)\}$
no priority	Γ

Table 6: Options and Semantics for Priority Schemes

envision a simpler state semantics that prevents infinite loops by allowing no state to be exited more than once in a macro-step.

Enabling Internal Events

Table 3 defines parameter functions for deriving the set of enabling internal events. In STATEMATE and RSML, only internal events generated in the previous micro-step can trigger a transition, whereas in the original statecharts semantics, internal events generated by transitions remain enabling events throughout the macro-step. The last row shows how to model SDL "input queues" as ordered lists that contain both internal and external events: at the start of each macro-step, the new external events are appended to the end of the event list; the enabling event is the event closest to the head of the list that triggers an enabled transition; and an executing transition removes from the head of the list all events up to and including the enabling event (thereby losing events to which the HTS was not ready to react), and appends all of the events that the transition generates.[1]

[1] SDL's save and priority-event constructs could be incorporated into the above semantics. Note that these features would add save and priority-event to the list of transition elements.

Enabling External Events

Table 4 defines parameter functions for deriving the set of the enabling external events. At the beginning of each macro-step, all of the new external events *EE* are enabling events. In RSML and STATEMATE, external events can trigger transitions only in the first micro-step of a macro-step. In statecharts, external events remain enabling events throughout the macro-step. We can also imagine a notation in which an external event remains an enabling event until it triggers a transition or until the macro-step ends.

Enabling Variable Values

Table 5 defines parameter functions for variable value semantics. This function takes a snapshot and set of assignments (a) as input. In most notations, the current variable-value assignments are used to evaluate transitions' enabling conditions. But in the original statecharts semantics, only those variable assignments that hold at the beginning of the macro-step can enable transitions.

Priority

Table 6 provides sample definitions for parameter function *pri*, which returns the subset of transitions of highest priority. In scope-based priority, the priority of a transition is the *rank* of its *scope*,

where *rank* and *scope* are defined in Section 2.1. The priority of transition t_2 in Figure 1 is the rank of $scope(t_2) = S_1$, which is 1. Lower-ranked scopes may have priority over higher-ranked scopes ($scope_O$, for "outer" transitions), or vice versa ($scope_I$, for "inner" transitions). In source-based priority, the priority of a transition is the *rank* of its *source* state. In this case, the priority of transition t_2 is the rank of $source(t_2) = S_3$, which is 2. Source-based priority favours either lower-ranked source states ($source_O$) or higher-ranked source states ($source_I$). Priority schemes $scope_O$ and $source_O$ favour super-state behaviour, whereas schemes $scope_I$ and $source_I$ favour sub-state behaviour. In explicit priority, a transition's priority is explicitly defined by the specifier, where a low rank is considered a high priority.

Some notations use a combination of priority schemes. For example, a language may have a scope-based or source-based priority scheme that can be overridden by explicit priorities.

2.2.3 Macro-Step Semantics

We have identified two macro-step semantics, which we call "simple" and "stable". A simple macro-step is equal to one micro-step. A "stable" macro-step is a maximal sequence of micro-steps, such that the sequence ends only when there are no more enabled transitions.

Simple Macro-Step

In simple macro-step semantics, an HTS takes a micro-step if one is enabled, and otherwise makes no change to the snapshot. Each step is a reaction to a new set of external events *EE*. (Recall that external events are not part of the snapshot.)

$$N_{macro}(ss, ss', EE) =$$
$$\text{let } startss := init_{\text{simple}}(ss, EE) \text{ in}$$
$$(\exists\, \tau \,.\, N_{micro}(startss, ss', \tau)) \ \vee \ (startss = ss')$$

The first line in this definition creates a new starting snapshot *startss* for the macro-step. The function $init_{\text{simple}}$ removes accumulated information about transitions that executed in the last macro-step. In simple macro-step semantics, there is no need to accumulate information about micro-steps within a macro step, which means snapshot elements CS_h, AV_h, and EE_g can be ignored. The enabling states CS_e is the set of current states *CS*, and the enabling variable values AV_e is the current variable values *AV*. We still distinguish between current and enabling events because, even in simple macro-step semantics, a notation may allow transitions to be enabled by past events that have not yet triggered transitions (e.g., as in SDL).

Stable Macro-Step

In stable macro-step semantics, a macro-step is a maximal sequence of micro-steps. A snapshot with no enabled transitions is called a **stable snapshot**. When a stable snapshot is reached, a new macro-step starts with a new set of external events *EE*. We define a relation (N^i) that is true for a pair of snapshots if there is a sequence of length *i* of micro-steps from the first snapshot to the second, where the second snapshot is a stable snapshot.

$$N^0(ss, ss') =$$
$$\neg(\exists\, ss'', \tau \,.\, N_{micro}(ss, ss'', \tau)) \ \wedge \ (ss = ss')$$
$$N^{i+1}(ss, ss') =$$
$$\exists\, ss'', \tau \,.\, N_{micro}(ss, ss'', \tau) \wedge N^i(ss'', ss')$$

We can now define $N_{macro}(ss, ss')$ for stable macro-steps.

$$N_{macro}(ss, ss', EE) =$$
$$\text{let } startss := init_{\text{stable}}(ss, EE) \text{ in}$$
$$\text{if } (\exists\, i > 0 \,.\, \exists\, ss'' \,.\, N^i(startss, ss''))$$
$$\text{then } (\exists\, i > 0 \,.\, N^i(startss, ss'))$$
$$\text{else } startss = ss'$$

There is a macro-step from snapshot *startss* to ss' only if there is a non-empty sequence of micro-steps starting from the initialized snapshot and ending in stable snapshot ss'. Otherwise, the only possible macro-step is the idle step ($startss = ss'$).

Initial Snapshot

The initial state of an HTS is the same for both kinds of macro-step semantics:

$$ss_I = < I, \varnothing, V_I, \varnothing, I, \varnothing, V_I, \varnothing >$$

where I and V_I are the HTS's initial set of states and variable assignments, and the sets of internal events and generated events are initially empty. The history elements are initialized by their respective parameter functions at the start of the first macro step. This definition could easily be generalized if there are multiple possible initial states or multiple possible initial variable assignments.

3. COMPOSITION OPERATORS

In this section, we describe the semantics of a number of well-used composition operators. The operands of a composition operator are components. A *component* may be either a basic component (i.e., an HTS) or a collection of components that have been composed using a composition operator. Operators use the components' next-step relations rather than constructing a new machine, which means that abstraction techniques like partial order reduction [14] can examine the basic components.

We specify each composition operator at the micro-step level, and infer the semantics at the macro-step level as a sequence of zero or more composed micro-steps. For operators that can compose at the macro-step level, we provide explicit definitions for composition for stable macro-step semantics only. At the beginning of such a macro-step, the external events generated by each component in the previous macro-step must be added to the set of external events sensed by the other component.

The composition operators make intricate changes to the elements of the components' snapshots to represent inter-machine communication. For example, the operator must ensure that assignments to shared values made by any component machine are reflected in all appropriate snapshots. In addition, the composition operator is responsible for "message passing" among components, making events generated by one component visible to the other component. We modify snapshots using substitution, where $ss \,|^x_y$ refers to a snapshot that is equal to *ss* except for element *x*, which has value *y*. Substitution over a set of snapshots $\vec{ss} \,|^x_y$ defines a substitution to each snapshot in $\vec{ss}$. For example,

$$\vec{ss} \,|^{AV}_{assign(\vec{ss}.AV,\ resolve\_conflicts(asn(\vec{\tau_1} \cup \vec{\tau_2})))}$$

updates the variable assignments (*AV*) in each snapshot in $\vec{ss}$ with the variable values assigned in transition sets $\vec{\tau_1}$ and $\vec{\tau_2}$, letting function *resolve_conflicts* handle multiple assignments to the same variable.

In Figure 2, we introduce two abbreviations to describe how components communicate. Function *update* is used to update the snapshot of a non-executing component with changes to shared variables and events made by transitions taken in the other, executing component. Function *communicate* is similar, but it is used when both components execute so it starts from an intermediate snapshot $i\vec{ss}$ that reflects the effects of one component's transitions but not yet the shared effects of the other component's transitions. In both functions, the set of internal events *IE* in the returned snapshot is the set of events generated by the executing transitions. The history of internal events (IE_h) is updated using the parameter function *n_int_ev_his*. The variable values are updated using *resolve_conflicts* and *assign*, and the history of variable values is

updated according to the parameter function *n_var_val_his*. By using the parameter functions described in the previous section, we ensure consistency with the choices for how internal and external events and variable assignments persist within a micro-step.

In all cases but two (interrupt and sequence composition), the initial snapshot for the composed machine is the composite of the component machines' initial snapshots:

$$\vec{ss}_I = (\vec{ss}_{I1}, \vec{ss}_{I2})$$

We assume that shared variable assignments are initially consistent in different components.

We present three composition operators in detail: parallel, environmental synchronization, and interrupt. The semantics for the remaining composition primitives are described informally in the text with their formal definitions in Appendix A.

3.1 Parallel

Two components executing in parallel means that their transitions execute simultaneously if both are enabled, otherwise only one component executes and the other updates shared variables and events (Figure 3). If both components execute, then their next snapshots should satisfy N^1_{micro} and N^2_{micro}, except for the values of shared variables and events that must be communicated to each other. We introduce intermediate snapshots $\vec{iss}_1$ and $\vec{iss}_2$ that are reachable in the components' N_{micro} relations, and we use the function *communicate* to describe how the components' next snapshots in the composed machine differ from these intermediate snapshots.

If only one component can execute, then the other component's snapshot stays the same, except for updating shared variables and events. The case where both components do not change is not a possible micro-step.

In parallel composition at the macro-step level N^{para}_{macro}, both components must take macro steps (although one or both may take idle steps) and must agree on the shared variables and events in the resultant snapshots. Because the history variables are only relevant for micro-steps, it is not necessary to update their values at the macro-step level.

3.2 Environmental Synchronization

Our environmental synchronization composition operator captures LOTOS's and CSP's parallel compositions ($\|$). These operators do not distinguish between events triggering a transition and events generated by a transition, so we consider only triggering events in the semantics of this operator.

There are two cases for micro-step environmental synchronization (Figure 4). In the first case, all transitions are triggered by the same event *e*, which is in the set of events, *sync_events*, on which the two components synchronize (line 1). All processes that listen for this event must 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 second case, there are no events on which the two components can synchronize, so one or the other component takes a step in isolation; this step cannot include transitions triggered by synchronization events. We call this second case the **unsynch case** and will refer to it in subsequent definitions.

3.3 Rendezvous Synchronization

Rendezvous synchronization (Appendix A, Figure 6) is similar to environmental synchronization, except that in the synchronized case exactly one transition in the sending component generates a synchronization event that triggers exactly one transition in the receiving component.

3.4 Interleaving

In interleaving composition, only one component can execute its transitions at a time. At the micro-step level, it is equivalent to the unsynch case in synchronization composition. The two synchronization operators above degenerate to interleaving semantics when their set of synchronization events is empty. At the macro-step level (Appendix A, Figure 7), interleaving composition interleaves the macro steps of each machine.

3.5 Sequence

In sequence composition (Appendix A, Figure 8), 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.) We use the function *basic_states* to isolate the basic states from the set of current states. There are three stages to a sequence composition. In the first stage, component one executes and the shared variables of component two are updated. In the second stage, component one has reached its final states: the composition operator updates component two's current snapshot with appropriate information from component one so that component two can take a step, and it clears the states from component one's snapshot so that component one can no longer execute. In the third stage, component two executes and, for consistency, the snapshot of component one is updated.

3.6 Choice

In choice composition (Appendix A, Figure 9), the composition operator nondeterministically chooses one component to execute in isolation, and the other component never executes. This choice is made in the initial snapshot, and thereafter the composite machine behaves only like the chosen component. We capture these semantics by clearing the set of current states from the unchosen component's snapshot to keep it from executing. For consistency, we continue to update the unchosen component's snapshot.

3.7 Interrupt Composition Primitive

Interrupt composition (Figure 5) combines two components via a pre-defined set of **interrupt transitions** (T_{interr}). These transitions may have sources and destinations that are sub-states of the components rather than their root states. Interrupt transitions are similar to HTS transitions in our basic components, except that they transition between components that may have concurrent sub-components.

There are four cases in interrupt composition. In the first case, component one can take transitions and any enabled interrupt transition has lower priority than these transitions. Therefore, component one takes transitions and component two's shared variables are updated. We introduce the function $higher\_pri(x, y)$ to test if the highest priority transition in the set *x* has equal or higher priority than the highest priority transition in the set *y*; this function is defined in terms of the priority semantics given in Table 6 and in terms of the ranks of states, which increase as components are composed. In the second case, one of the interrupt transitions is enabled and has priority over enabled transitions in component one, which means that control passes from component one to component two. The composition operator clears the current states in component one, so that it will not execute, and it applies the actions of the executing transition to component two's snapshot. Because the semantics of these actions depends on each component's step semantics, we use the functions from Section 2 to compute the actions' affects on component two's snapshot. We introduce function *ent_comp* to

$$update(\vec{ss}, \vec{\tau}) \;=\; \vec{ss} \;|^{IE}_{ev\_gen(\vec{\tau})}|^{IE_h}_{n\_int\_ev\_his(\vec{ss},\vec{\tau})}|^{AV}_{assign(\vec{ss}.AV,\; resolve\_conflicts(asn(\vec{\tau})))}|^{AV_h}_{n\_var\_val\_his(\vec{ss},resolve\_conflicts(asn(\vec{\tau})))}$$

$$communicate(\vec{iss}, \vec{ss}, \vec{\tau_1}, \vec{\tau_2}) \;=\; \vec{iss} \;|^{IE}_{ev\_gen(\vec{\tau_1}\cup\vec{\tau_2})}|^{IE_h}_{n\_int\_ev\_his(\vec{ss},\vec{\tau_1}\cup\vec{\tau_2})}|^{AV}_{assign(\vec{ss}.AV,\; resolve\_conflicts(asn(\vec{\tau_1}\cup\vec{\tau_2})))}|^{AV_h}_{n\_var\_val\_his(\vec{ss},\; resolve\_conflicts(asn(\vec{\tau_1}\cup\vec{\tau_2})))}$$

Figure 2: Abbreviations used in the semantics

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

if $(\exists\, \vec{ss}, \vec{\tau}.\ N^1_{micro}(\vec{ss}_1, \vec{ss}, \vec{\tau}))\ \wedge\ (\exists\, \vec{ss}, \vec{\tau}.\ N^2_{micro}(\vec{ss}_2, \vec{ss}, \vec{\tau}))$ then (* both can take a step *)

$$\exists\, \vec{iss}_1, \vec{iss}_2.\ \left[\begin{array}{ll} & N^1_{micro}(\vec{ss}_1, \vec{iss}_1, \vec{\tau}_1)\ \wedge\ \vec{ss}'_1 = communicate(\vec{iss}_1, \vec{ss}_1, \vec{\tau}_1, \vec{\tau}_2) \\ \wedge & N^2_{micro}(\vec{ss}_2, \vec{iss}_2, \vec{\tau}_2)\ \wedge\ \vec{ss}'_2 = communicate(\vec{iss}_2, \vec{ss}_2, \vec{\tau}_1, \vec{\tau}_2) \end{array}\right] \quad \text{(* both take a step *)}$$

$$\text{else}\ \left[\begin{array}{ll} \vee & N^1_{micro}(\vec{ss}_1, \vec{ss}'_1, \vec{\tau}_1)\ \wedge\ \vec{\tau}_2 = \emptyset\ \wedge\ \vec{ss}'_2 = update(\vec{ss}_2, \vec{\tau}_1) \\ & \text{(* symmetric case *)} \end{array}\right] \quad \text{(* only one executes; the other changes shared variables and events *)}$$

$$N^{para}_{macro}((s\vec{s}_1, s\vec{s}_2),\ (\vec{ss}'_1, \vec{ss}'_2),\ EE) =$$

$$\exists\, \vec{iss}_1, \vec{iss}_2.\ \left[\begin{array}{ll} & N^1_{macro}(\vec{ss}_1, \vec{iss}_1, EE \cup \vec{ss}_2.EE_g)\ \wedge\ \vec{ss}'_1 = \vec{iss}_1\ |^{AV}_{assign(\vec{ss_1}.AV,\; resolve\_conflicts(\vec{iss}_1.AV \cup \vec{iss}_2.AV))} \\ \wedge & N^2_{macro}(\vec{ss}_2, \vec{iss}_2, EE \cup \vec{ss}_1.EE_g)\ \wedge\ \vec{ss}'_2 = \vec{iss}_2\ |^{AV}_{assign(\vec{ss_2}.AV,\; resolve\_conflicts(\vec{iss}_1.AV \cup \vec{iss}_2.AV))} \end{array}\right] \quad \text{(* both take a step *)}$$

Figure 3: Semantics of parallel composition for micro- and macro-steps

$$N^{env\text{-}sync}_{micro}((\vec{ss}_1, \vec{ss}_2),\ (\vec{ss}'_1, \vec{ss}'_2),\ (\vec{\tau}_1, \vec{\tau}_2))\ sync\_events =$$

$$\exists\, \vec{iss}_1, \vec{iss}_2, e.\ \left[\begin{array}{ll} & ev\_trig(\vec{\tau}_1 \cup \vec{\tau}_2) = \{e\}\ \wedge\ e \in sync\_events \quad \text{(* line 1 *)} \\ \wedge & \forall\, T \in \vec{T_1} \cup \vec{T_2}.\ (\exists\, \tau \in T.\ ev\_trig(\tau) = \{e\}) \Longrightarrow ((\vec{\tau}_1 \cup \vec{\tau}_2) \cap T \neq \emptyset) \quad \text{(* line 2 *)} \\ \wedge & N^1_{micro}(\vec{ss}_1, \vec{iss}_1, \vec{\tau}_1)\ \wedge\ \vec{ss}'_1 = communicate(\vec{iss}_1, \vec{ss}_1, \vec{\tau}_1, \vec{\tau}_2) \\ \wedge & N^2_{micro}(\vec{ss}_2, \vec{iss}_2, \vec{\tau}_2)\ \wedge\ \vec{ss}'_2 = communicate(\vec{iss}_2, \vec{ss}_2, \vec{\tau}_1, \vec{\tau}_2) \end{array}\right] \quad \text{(* sync on event in multiple processes *)}$$

$$\vee\ \left[\begin{array}{ll} \underline{\vee} & \left(\begin{array}{ll} & N^1_{micro}(\vec{ss}_1\ |^{IE}_{\vec{ss_1}.IE \setminus sync\_events}|^{IE_h}_{\vec{ss_1}.IE_h \setminus sync\_events}|^{EE_h}_{\vec{ss_1}.EE_h \setminus sync\_events},\ \vec{ss}'_1,\ \vec{\tau}_1) \\ \wedge & \vec{\tau}_2 = \emptyset\ \wedge\ \vec{ss}'_2 = update(\vec{ss}_2, \vec{\tau}_1) \end{array}\right) \\ & \text{(* symmetric case of above replacing 1 with 2 and 2 with 1 *)} \end{array}\right] \quad \text{(* unsync case: only one component executes. Reused in Figure 6 *)}$$

Figure 4: Semantics of environmental synchronization for micro-steps

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

$$\exists\, \vec{iss}_1, \vec{t}.\ \left[\begin{array}{ll} & \vec{ss}_1.CS \neq \emptyset\ \wedge\ N^1_{micro}(\vec{ss}_1, \vec{iss}_1, \vec{t})\ \wedge\ higher\_pri(\vec{t}, pri(enabled\_trans(\vec{ss}_1, T_{interr}))) \\ \wedge & N^1_{micro}(\vec{ss}_1, \vec{ss}'_1, \vec{\tau}_1)\ \wedge\ \vec{ss}'_2 = update(\vec{ss}_2, \vec{\tau}_1) \\ & \vec{\tau}_2 = \emptyset\ \wedge\ higher\_pri(\vec{\tau}, \vec{t}) \end{array}\right] \quad \text{(* component 1 steps *)}$$

$$\vee$$

$$\exists\, \tau.\ \left[\begin{array}{ll} & \tau \in pri(enabled\_trans(\vec{ss}_1,\ T_{interr}))\ \wedge\ (\forall\, \vec{iss}_1, \vec{t}.\ N^1_{micro}(\vec{ss}_1, \vec{iss}_1, \vec{t}) \Longrightarrow higher\_pri(\{\tau\}, \vec{t})) \\ \wedge & \vec{ss}'_1 = update(\vec{ss}_1, \tau)\ |^{CS}_{\emptyset}\ \wedge\ \vec{\tau}_1 = \emptyset\ \wedge\ \vec{\tau}_2 = \emptyset \\ \wedge & \vec{ss}'_2 = update(\vec{ss}_2, \tau)\ |^{CS}_{ent\_comp(\tau)}|^{CS_h}_{n\_states\_his(\vec{ss_2}, \tau)}|^{EE_h}_{n\_ext\_ev\_his(\vec{ss_1},\ \tau)} \end{array}\right] \quad \text{(* transition to component 2 *)}$$

$\vee$ (* symmetric cases of two above replaced 1 with 2 and 2 with 1 *)

Figure 5: Semantics of interrupt semantics for micro-steps

determine the current states of component two; this function 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 sub-components). 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. Notice that only one component can ever have enabled transitions.

The initial composite snapshot for interrupted 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 empty.

4. SEMANTICS OF NOTATIONS

We can describe the semantics of several specification notations concisely as instantiations of our framework. Tables 2–5 instantiate the parameter functions of several popular model-based notations. In this section, we list each of those notations' choice of macro-step type and transition priority, and we match its composition operators to our composition operators.

All of basic transition systems (BTS) [22], CSP [17], CCS [24] LOTOS [18], and SDL88 [20] use the "no priority" priority scheme and the "simple" option for macro-step semantics, which dictates the definitions of the parameter functions for step semantics. None of these languages has a state hierarchy.

BTS's interleaving ($|||$), concatenation (;) and selection (OR) are our interleaving, sequence, and choice operators, respectively.

CSP's communication $(c!v \rightarrow P \| c?x \rightarrow Q)$ is our rendezvous synchronization. CSP's parallel ($\|$), interleaving ($|||$), sequential composition (;), and general choice ($[]$) are our environmental synchronization (with the synchronization set consisting of all shared events), interleaving, sequence, and choice operators, respectively. We have not yet formalized CSP's interrupt ($\hat{}$) 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.

CCS's summation (+), composition ($|$), sequential (;), and conjunction ($\|$) operators are our choice, rendezvous, sequence, and environmental synchronization operators, respectively.

ESTELLE hierarchical modules [19] use the "simple" option for macro-step semantics and the "scope$_O$" priority scheme, combined with "explicit" priorities to resolve among transitions that have the same scope. Components collected into a "process" parent map to our parallel composition, and components collected in an "activity" parent map to our interleaving composition.

LOTOS's parallel composition ($\|$), pure interleaving ($|||$), sequential composition ($\gg$), and choice ([ ]) are our environmental synchronization, interleaving, sequence, and choice operators, respectively. We do not yet handle LOTOS's disabling ($[>$) construct. It would be a modification of our sequence composition.

SDL's composition has been described as either our interleaving or parallel composition. Because SDL88 does not allow shared variables, these two choices are equivalent for any properties that do not dependent on a specific time interval.

A statecharts' state with only OR-substates is an HTS. Statecharts' AND-states are formed using our parallel composition. Statecharts' OR-state composition is our interrupt composition. All statecharts dialects use stable macro-step semantics, but they have different micro-step semantics as depicted in Tables 2–5. Harel et al's [15] original formulation of statecharts corresponds to the priority scheme of "no priority". The semantics of STATEMATE [16] have a priority scheme of "scope$_O$". RSML [21] corresponds to a priority scheme of "no priority'.

5. 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 [8, 13], the most famous of which is von der Beeck's comparison of statecharts variants [29]. These taxonomies 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 behaviour. We also express variations in step-semantics as parameters, which makes it easier to define new notations and to identify both major and subtle differences among notations' semantics.

A number of researchers have proposed translating specification notations into more fundamental modelling notations, such as first order logic [30, 31], hierarchical state machines [23], labelled transition systems [4], and hybrid automata [2]. 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 modelling notation can be applied to the translated specification. Researchers have also proposed general semantic models that are capable of capturing the semantics of multiple notations and extended features of these notations [11].

Researchers have introduced *intermediate languages*, such as SAL [3] and IF [5], that are designed to be elegant yet expressive target languages that ease translations between notations. In the case of SAL and IF, 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 language and needs to be modified whenever the language's semantics change.

Work with similar goals to ours is that of Day and Joyce [10], Pezzè and Young [26, 27], and Dillon and Stirewalt [12, 28]. 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. Embedding avoids the translation step and the effort to construct and maintain translators. Notations have also been embedded in the theorem prover PVS [25], 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 components that are specified in different notations can also be described.

Dillon and Stirewalt define operational semantics for process-algebra and temporal-logic notations, and semi-automatically 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 configurations, 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.

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.

6. CONCLUSIONS

We have introduced an operational semantics template for model-based specification notations that parameterizes a notation's step-semantics. Using this template, one can define the semantics of a new notation simply by (1) instantiating the template's thirteen parameter functions and by (2) defining how composition operators control components' executions and change snapshot elements to reflect inter-component communication and synchronization. We expect the degree of parameterization to grow as we survey more notations, but we believe it will grow to support new snapshot elements (e.g., to augment states with entry or exit actions) rather than grow in the number of parameter functions on existing snapshot elements. That said, one candidate for future parameterization is the function *resolve_conflicts*: a specifier may want to specialize how a notation resolves conflicting, concurrent variable assignments.

We believe that our template, with its separation of fixed and parameteric semantics, may be a suitable input language for a tool or process that generates notation-specific analysis tools. To evaluate this hypothesis, we are currently embedding into higher-order logic the formal definitions that we presented in this paper. In this implementation of our framework, an instantiated template would define a theory for reasoning about specifications in the defined notation, and a finite specification could be transformed into a BDD representation for model checking.

7. ACKNOWLEDGMENTS

We thank Dan Berry, Andrew Malton, and John Thistle of the University of Waterloo for helpful discussions on this topic.

8. REFERENCES

[1] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. on Soft. Eng.*, 19(1):24–40, 1993.

[2] G. Avrunin, J. Corbett, and L. Dillon. Analyzing partially-implemented real-time systems. In *Int. Conf. on Soft. Eng.*, pages 228–238. ACM Press, 1997.

[3] S. Bensalem et al. An overview of SAL. In *Langley Formal Methods Workshop*, pages 187–196, 2000.

[4] L. Blair and G. Blair. Composition in multi-paradigm specification techniques. In *Proc. of Int. Workshop on Formal Methods for Open Object-based Dist. Syst.*, 1999.

[5] 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. In *Proc. of SDL-FORUM 1999*, 1999.

[6] M. Bozga, S. Graf, L. Mounier, and J. Sifakis. The intermediate representation IF. Technical report, Verimag, 1998.

[7] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin. Model checking large software specifications. *IEEE Trans. on Soft. Eng.*, 24(7):498–519, 1998.

[8] P. Chou and G. Borriello. An analysis-based approach to composition of distributed embedded systems. In *Proc. of the Int. Workshop on Hardware/Software Co-Design*, 1998.

[9] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Trans. on Prog. Lang. and Syst.*, 15(1):36–72, 1993.

[10] N. A. Day and J. J. Joyce. Symbolic functional evaluation. In *Theorem Proving in Higher Order Logics*, number 1690 in LNCS, pages 341–358. Springer, 1999.

[11] P. Degano and C. Priami. Enhanced operational semantics: A tool for describing and analyzing concurrent systems. *ACM Computing Surveys*, 33(2):135–176, 2001.

[12] L. K. Dillon and K. Stirewalt. Lightweight analysis of operational specifications using inference graphs. In *Int. Conf. on Soft. Eng.*, pages 57–67. IEEE Comp. Soc, 2001.

[13] S. Easterbrook and M. Chechik. Personal Communication, February, 2002.

[14] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *Computer Aided Verification*, volume 531 of *LNCS*, pages 176–185. Springer, 1990.

[15] D. Harel et al. On the formal semantics of statecharts. In *Symp. on Logic in Comp. Sci.*, pages 54–64, 1987.

[16] D. Harel and A. Naamad. The Statemate semantics of statecharts. *ACM Trans. on Soft. Eng. Meth.*, 5(4):293–333, 1996.

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

[18] ISO8807. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical report, ISO, 1988.

[19] ISO9074. ESTELLE - a formal description technique based on an extended state transition model. Technical report, ISO, 1989.

[20] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Technical Report Z-100, International Telecommunication Union - Standardization Sector, 1999.

[21] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Trans. on Soft. Eng.*, 20(9), 1994.

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

[23] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Proc. of Asian Comp. Sci. Conf.*, number 1345 in LNCS. Springer, 1997.

[24] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[25] S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, SRI International, Computer Science Laborabory, 1995.

[26] M. Pezzè and M. Young. Creating of multi-formalism state-space analysis tools. In *Int. Symp. on Soft. Testing and Analysis*, pages 172–179. ACM Press, 1996.

[27] M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools. In *Int. Conf. on Soft. Eng.*, pages 239–249. ACM Press, 1997.

[28] K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *Int. Conf. on Soft. Eng.*, pages 167–176. IEEE Comp. Soc., 2001.

[29] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real Time and Fault-Tolerant Systems*, number 863 in LNCS, pages 128–148. Springer, 1994.

[30] P. Zave and M. Jackson. Conjunction as composition. *ACM Trans. on Soft. Eng. Meth.*, 2(4):379–411, 1993.

[31] P. Zave and M. Jackson. Where do operations come from? A multiparadigm specification technique. *IEEE Trans. on Soft. Eng.*, 22(7):508–528, 1996.

Appendix A

$$N^{\textit{rend-sync}}_{\textit{micro}}((s\vec{s}_1, s\vec{s}_2),\ (s\vec{s}'_1, s\vec{s}'_2),\ (\vec{\tau}_1, \vec{\tau}_2))\ \textit{sync\_events}\ =$$

$$\exists\, i\vec{ss}_1, i\vec{ss}_2, e.\ \left[\begin{array}{ll} & e \in \textit{sync\_events} \wedge\ |\ \vec{\tau}_1\ | = 1 = |\ \vec{\tau}_2\ |\ \wedge \textit{ev\_gen}(\vec{\tau}_1) = \textit{ev\_trig}(\vec{\tau}_2) = \{e\} \\ \wedge & N^1_{\textit{micro}}(s\vec{s}_1, i\vec{ss}_1, \vec{\tau}_1)\ \wedge\ s\vec{s}'_1 = \textit{communicate}(i\vec{ss}_1, s\vec{s}_1, \vec{\tau}_1, \vec{\tau}_2) \\ \wedge & N^2_{\textit{micro}}(s\vec{s}_2, i\vec{ss}_2, \vec{\tau}_2)\ \wedge\ s\vec{s}'_2 = \textit{communicate}(i\vec{ss}_2, s\vec{s}_2, \vec{\tau}_1, \vec{\tau}_2) \end{array}\right]$$

(* sync. on one sync event generated by one transition in component 1, triggering only one transition in component 2 *)

$\vee$ (* symmetric case of above replacing 1 with 2 and 2 with 1 *)

$\vee$ (* unsync case *)

Figure 6: Semantics of rendezvous synchronization for micro-steps

$$N^{\textit{intl}}_{\textit{macro}}((s\vec{s}_1, s\vec{s}_1),\ (s\vec{s}'_1, s\vec{s}'_2), EE) =$$

$$\left[\begin{array}{ll} \underline{\vee} & N^1_{\textit{macro}}(s\vec{s}_1, s\vec{s}'_1, EE \cup s\vec{s}_2.EE_g)\ \wedge\ s\vec{s}'_2 = s\vec{s}_2\ |^{AV}_{\textit{assign}(s\vec{s}_2.AV,\ s\vec{s}'_1.AV)} \\ & N^2_{\textit{macro}}(s\vec{s}_2, s\vec{s}'_2, EE \cup s\vec{s}_1.EE_g)\ \wedge\ s\vec{s}'_1 = s\vec{s}_1\ |^{AV}_{\textit{assign}(s\vec{s}_1.AV,\ s\vec{s}'_2.AV)} \end{array}\right]$$

(* either component can take a step, but not both *)

Figure 7: Semantics of interleaving composition for macro-steps

$$N^{\textit{seq}}_{\textit{micro}}((s\vec{s}_1, s\vec{s}_2),\ (s\vec{s}'_1,\ s\vec{s}'_2), (\vec{\tau}_1, \vec{\tau}_2)) =$$

$$\left[\begin{array}{ll} \wedge & \textit{basic\_states}(s\vec{s}_1.CS) \not\subseteq F_1\ \wedge\ \vec{\tau}_2 = \varnothing \\ & N^1_{\textit{micro}}(s\vec{s}_1, s\vec{s}'_1, \vec{\tau}_1)\ \wedge\ s\vec{s}'_2 = \textit{update}(s\vec{s}_2, \vec{\tau}_1) \end{array}\right]$$

(* Component 1 steps *)

$$\vee \left[\begin{array}{ll} \wedge & \textit{basic\_states}(s\vec{s}_1.CS) \subseteq F_1\ \wedge\ s\vec{s}_1.CS \neq \varnothing\ \wedge\ s\vec{s}'_1 = \textit{update}(s\vec{s}_1, \vec{\tau}_2)\ |^{CS}_{\varnothing}\ \wedge\ \vec{\tau}_1 = \varnothing \\ & N^2_{\textit{micro}}(s\vec{s}_2\ |^{EE_h}_{s\vec{s}_1.EE_h},\ s\vec{s}'_2,\ \vec{\tau}_2) \end{array}\right]$$

(* Component 2 starts and steps *)

$$\vee \left[\ s\vec{s}_1.CS = \varnothing\ \wedge\ \vec{\tau}_1 = \varnothing\ \wedge\ N^2_{\textit{micro}}(s\vec{s}_2, s\vec{s}'_2, \vec{\tau}_2)\ \wedge\ s\vec{s}'_1 = \textit{update}(s\vec{s}_1, \vec{\tau}_2)\ \right]$$

(* Component 2 steps *)

Figure 8: Semantics of sequence composition for micro-steps

$$N^{\textit{choice}}_{\textit{micro}}((s\vec{s}_1, s\vec{s}_2),\ (s\vec{s}'_1, s\vec{s}'_2),\ (\vec{\tau}_1, \vec{\tau}_2)) =$$

$$\left[\begin{array}{ll} & s\vec{s}_1.CS \subseteq I_1\ \wedge\ s\vec{s}_2.CS \subseteq I_2\ \wedge\ s\vec{s}_1.CS \neq \varnothing\ \wedge\ s\vec{s}_2.CS \neq \varnothing \\ \wedge & \left(\left[\begin{array}{ll} & N^1_{\textit{micro}}(s\vec{s}_1, s\vec{s}'_1, \vec{\tau}_1)\ \wedge\ \vec{\tau}_2 = \varnothing \\ \wedge & s\vec{s}'_2 = \textit{update}(s\vec{s}_2, \vec{\tau}_1)\ |^{CS}_{\varnothing} \end{array}\right] \underline{\vee} \left[\begin{array}{ll} & N^2_{\textit{micro}}(s\vec{s}_2, s\vec{s}'_2, \vec{\tau}_2)\ \wedge\ \vec{\tau}_1 = \varnothing \\ \wedge & s\vec{s}'_1 = \textit{update}(s\vec{s}_1, \vec{\tau}_2)\ |^{CS}_{\varnothing} \end{array}\right]\right) \end{array}\right]$$

(* choose a component *)

$$\vee \left(\left[\begin{array}{ll} & s\vec{s}_2.CS = \varnothing\ \wedge\ N^1_{\textit{micro}}(s\vec{s}_1, s\vec{s}'_1, \vec{\tau}_1) \\ \wedge & s\vec{s}'_2 = \textit{update}(s\vec{s}_2, \vec{\tau}_1)\ \wedge\ \vec{\tau}_2 = \varnothing \end{array}\right] \vee \text{(* symmetric case of left replacing 1 with 2 and 2 with 1 *)}\right)$$

(* chosen component steps *)

Figure 9: Semantics of choice composition for micro-steps