Understanding and Comparing Model-Based Specification Notations

Jianwei Niu, Joanne M. Atlee, Nancy A. Day
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada
N2L 3G1
{*jniu,jmatlee,nday*}*@uwaterloo.ca*

Technical Report CS-2003-01
February 7, 2003

Abstract

Specifiers must be able to understand and compare the specification notations that they use. Traditional means for describing notations' semantics (e.g., operational semantics, logic, natural language) do not help users to identify the essential differences among notations. In previous work, we presented a template-based approach to defining model-based notations, in which 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. In this paper, we demonstrate the template's expressivity by using it to document the semantics of Petri Nets, SDL, and SCR. We also show how the template can be used to compare notation variants. We believe template definitions of notations ease a user's effort in understanding and comparing model-based notations.

1. Introduction

A requirements writer must have a solid understanding of specification notations to be able to choose appropriate notations and to use them properly. However, it can be difficult to acquire this expertise from published descriptions of notations. Notation developers tend to write either for a formal-methods audience (providing an operational semantics or logic definition), or for a software engineering audience (providing a pseudo-code or natural language definition). Formal definitions are precise, enabling tool development and inviting comparisons between notations; but such definitions are complex, involving multiple inter-dependent mathematical relations. Pseudo-code definitions provide a more intuitive understanding of semantics, describing when computations are performed and when variables change value; but such definitions tend to be less precise and less helpful for comparing notations.

We have developed a template approach to describe the semantics of model-based notations [13]. Our goal with the template is to make it easier to document the semantics of notations by focussing on how notations differ and requiring a user to describe only these differences. In our method, a parameterized template pre-defines the semantics that are common to all notations, and users specify the notation-specific semantics via parameter values. For example, the template defines the notion of enabled transitions in terms of enabling states, enabling events, and enabling variable values; parameters specify these predicates. Composition operators are defined separately from the execution semantics, and are parameterized by the same template parameters. The result is a semantics definition that isolates a notation's distinctive semantics, making it easier for requirements writers and students to compare the essential differences among model-based notations.

Our template was developed after surveying the execution semantics of seven popular specification notations: CSP [5], CCS [11], LOTOS [6], basic transition systems (BTS) [10], ESTELLE [7], a subset of SDL88 [8], and three variants of statecharts [2, 3, 9]. In this paper, we demonstrate the template's expressivity by using it to describe the semantics of Petri Nets, SCR, and a larger subset of SDL, whose semantics are quite different from the notations in our original survey and from each other. We also show how to use our template approach to compare notation variants (e.g., statecharts, RSML [9], UML state models [14]). We assume that the reader is familiar with all of these notations.

2. Overview of the Template

In this section, we give an overview of our template approach to defining model-based notations. The approach separates the definition of a notation's step semantics from the definition of its composition operators. We define a notation's step semantics in terms of the semantics of a single, sequential hierarchical transition system (HTS) – an extended finite state machine, adapted from basic transition systems [10] and statecharts [1]. An HTS supports no concurrency. Composition operators specify how collections of HTSs execute concurrently, transferring control to one another and exchanging events and data. Our presentation in this section is slightly more general than in our original work [13].

2.1. Syntax of HTS

A hierarchical transition system (HTS) is an 8-tuple, $\langle S, H, S_I, F, E, V, V_I, T\rangle$. S is a finite set of states, H is the state hierarchy, $S_I \subseteq S$ is the set of initial states, and $F \subseteq S$ 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. V is a finite set of data variables, with an initial value assignment of V_I. We assume that the names of unshared events and variables are distinct across HTSs. T is a finite set of transitions, each with the form,

$$\langle \mathit{src}, \mathit{trig\_ev}, \mathit{cond}, \mathit{act}, \mathit{dest}, \mathit{prty}\rangle$$

where *src*, *dest* $\subseteq S$ 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 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 **super states**, which contain other states, and **basic states**, which contain no other states. Each super state has a default child state, such that this default state is entered if the super state is a transition's destination state. A state hierarchy H defines a partial ordering on states, with the root state as the minimal element and basic states as maximal elements. The function *rank* assigns a number to a state based on the HTS's hierarchy:

$$\mathit{rank}(s) = \mathit{rank}(\mathit{parent}(s)) + 1$$

where $\mathit{rank}(\mathit{root}) = 0$.

Helper functions access the elements of a transition τ:

- $\mathit{source}(\tau)$ are the source states of τ.
- $\mathit{exited}(\tau)$ are the states exited when τ executes, including the source's ancestor and descendant states that are also exited.
- $\mathit{entered}(\tau)$ are the states entered when τ executes, including the destination's ancestor and descendant states and relevant default states that are also entered.
- $\mathit{trig}(\tau)$ are the events that trigger τ.
- $\mathit{pos}(\tau)$ are the positive events that trigger τ.
- $\mathit{neg}(\tau)$ are the negative events (*i.e.,* lack of event) that trigger τ.
- $\mathit{cond}(\tau)$ is τ's predicate guard condition
- $\mathit{gen}(\tau)$ are the events generated by τ's actions.
- $\mathit{asn}(\tau)$ are variable-value assignments in τ's actions.
- $\mathit{scope}(\tau)$ is the lowest common ancestor state of the transition's source and destination states.
- $\mathit{priority}(\tau)$ is τ's priority value

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., scope*) will return a set of results.

2.2. Step Semantics

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 consecutive snapshots. Formally, a snapshot is an 8-tuple $\langle CS, IE, AV, O, CS_a, IE_a, AV_a, EE_a\rangle$, where CS, IE, and AV are the sets of current states ($CS \subseteq S$), current internal events ($IE \subseteq E$), and current variable values, respectively; the set AV is a function that maps each variable in V to its current value. O is the set of current outputs to be communicated to concurrent components. Snapshot elements CS_a, AV_a, IE_a, and EE_a are auxiliary variables that accumulate data about states, variable values, and internal and external events, respectively. Inputs to an HTS (*e.g.,* external events) are not part of the snapshot because they lie outside of the system. Instead, the template parameters will incorporate input events and data into the auxiliary snapshot elements. If a notation does not need some snapshot element (*e.g.,* process algebras have no variables), then the related template parameters need not be provided.

A **step** moves an HTS from one snapshot to a successor snapshot. A **micro-step** results from executing exactly one transition. A **macro-step** is a sequence of zero or more micro-steps that is initiated by new input I from the environment. In **simple macro-step semantics**, new input from the environment is sensed at the start of every step, and a macro-step is either a single micro-step or an idle step (*i.e.,* the snapshot does not change). In **stable macro-step semantics**, input from the environment is sensed only at the start of a macro-step, and a macro-step is a sequence of micro-steps that represents the HTS's response to the environmental input; the macro-step ends with a **stable snapshot**, in which no transition is enabled.

The step semantics of a notation are a parameterized macro-step, N_{macro}, defined formally in [13]. The template parameters describe the values of the snapshot elements at

Snapshot Element	Start of Macro-step 1	Micro-step 2	General 3
$CS' =$	*init_states*(ss, I)	*n_cur_states*(ss, τ)	*en_states*(ss, τ)
$IE' =$	*init_int_ev*(ss, I)	*n_int_ev*(ss, τ)	*en_events*(ss, τ)
$AV' =$	*init_var_val*(ss, I)	*n_var_val*(ss, τ)	*en_cond*(ss, τ)
$O' =$	*init_gen_ev*(ss, I)	*n_gen_ev*(ss, τ)	*macro_semantics*
$CS'_a =$	*init_states_aux*(ss)	*n_states_aux*(ss, τ)	*pri*(Γ)
$IE'_a =$	*init_int_ev_aux*(ss, I)	*n_int_ev_aux*(ss, τ)	
$AV'_a =$	*init_var_val_aux*(ss)	*n_var_val_aux*(ss, τ)	
$EE'_a =$	*init_ext_ev_aux*(ss, I)	*n_ext_ev_aux*(ss, τ)	

Table 1. Set of parameter functions to be provided by template user

the beginning of the macro-step, the change to the snapshot elements as a result of following a transition τ in snapshot ss, and determine whether a transition is enabled based on its source state, triggering events and triggering condition. Parameters also determine the type of macro-step and how priority is handled. There are 21 parameters, which are listed in Table 1.

Start of Macro-step (Column 1) These functions execute at the start of each macro-step, clearing snapshot information about transitions that executed in the previous macro-step, and recording inputs I. For example in statecharts, the set of current internal events IE is reset to be empty at the start of a macro-step.

Micro-step (Column 2) These functions apply the effects of a transition τ to the snapshot elements. For example in statecharts, variable values are updated with assignments made by τ.

Enabling Predicates (Column 3) These predicates are used to determine whether a transition τ is enabled in the snapshot ss. The predicates test states, events, and conditions to see if they enable a transition τ in snapshot ss. For example in SDL, events are maintained in a queue and only the front-most event that can enable a transition.

Macro-step semantics (Column 3) Macro-semantics are either *simple* or *stable*, as described above; *simple* semantics are either *diligent*, meaning that enabled transitions have priority over idle steps, or *non-diligent*.

Priority (Column 3) The function *pri*(Γ) for a set of transitions Γ implements the notation's priority scheme. For example in scope-based priority, a transition's priority is the *rank* of its *scope*. A priority scheme that favours super-state behaviour over sub-state behaviour would give priority to transitions with the lowest-ranked scope.

2.3. Composition Operators

Composition operators specify how HTSs execute concurrently. The operands of a composition operator are **components**, where a component is either a single HTS or a collection of HTSs that have been composed via some composition operator(s). Semantically, a composition operator specifies how the components' snapshots change when the components take a collective step.

We define composition operators as parameterized, composite micro-step and/or macro-step relations that relate pairs of consecutive snapshot collections. For example, the composite micro-step relation for an operator op is

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

which relates snapshot collections $\vec{ss}_1$ and $\vec{ss}'_1$ and transitions $\vec{\tau}_1$ in component one, and $\vec{ss}_2$ and $\vec{ss}'_2$ and transitions $\vec{\tau}_2$ in component two in a single micro-step. The definition of N^{op}_{micro} is parameterized by the micro-step semantics of the two components, N^1_{micro} and N^2_{micro}, modulo the side effects on each component's snapshots from the other component's execution (shared events and assignments to shared variables) and modulo effects of the operator op (*e.g.,* transfer of control from one component to the other). We represent these effects by using substitution to modify snapshots: $ss\,|^x_y$ is a snapshot that is equal to snapshot ss, except that element x has value y. Substitution over a set of snapshots $\vec{ss}\,|^x_y$ defines a substitution to each snapshot in $\vec{ss}$. For example, functions *update* and *communicate* in Figure 1 update the snapshot elements IE, IE_a, AV, AV_a in snapshots $\vec{ss}$, with events and variable assignments from a set of transitions $\vec{\tau}$. Function *update* is used to update the snapshots of a non-executing component with the side effects from the executing component. Function *communicate* is used when both components execute. It starts from an intermediate snapshot that reflects the effects of one component's transitions but not yet the side effects of the other component's transitions. Composition operators use the template parameters, so that they adhere to their components' semantics for updating snapshot elements.

$$update(\vec{ss}, \vec{\tau}) \;=\; \vec{ss} \;|^{IE}_{n\_int\_ev(\vec{ss},\vec{\tau})}|^{IE_a}_{n\_int\_ev\_aux(\vec{ss},\vec{\tau})}|^{AV}_{n\_var\_val(\vec{ss},\vec{\tau})}|^{AV_a}_{n\_var\_val\_aux(\vec{ss},\vec{\tau})}$$

$$communicate(\vec{ss}, \vec{ss}, \vec{\tau}) \;=\; \vec{ss} \;|^{IE}_{n\_int\_ev(\vec{ss},\vec{\tau})}|^{IE_a}_{n\_int\_ev\_aux(\vec{ss},\vec{\tau})}|^{AV}_{n\_var\_val(\vec{ss},\vec{\tau})}|^{AV_a}_{n\_var\_val\_aux(\vec{ss},\vec{\tau})}$$

Figure 1. 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 *)

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

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

Figure 2. Micro-step semantics for parallel composition

Below, we review the definition of parallel composition. A more detailed presentation of a richer set of operators can be found in [13]. Because they use the template parameters and the snapshot relations of the two components, all the composition operators have a similar form.

In parallel composition, in each micro-step, the two components execute simultaneously if they are both enabled (Figure 2): 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, because it implies that a stable snapshot has been reached, initiating a new macro-step.

3. Petri Nets

In this section, we show how to represent the semantics of Petri Nets using our template. Petri Nets are a well-used formal notation for modelling and analyzing software systems (e.g., concurrent systems, distributed systems, communication protocols) [12, 15]. Many extensions of Petri Nets notation have been developed for modelling different applications by researchers, however, we only consider traditional Petri Nets in this paper.

A Petri net, usually represented as a directed graph, contains five types of elements: places, transitions, arcs, a weight function, and an initial marking. A place, drawn as a circle, contains zero or more tokens (dots). A transition is drawn as a bar or a box. An arc, drawn as a directed line, represents an input-output relation between a place and a transition. A pair (p_i, t) is used to denote an arc from a source place p_i to a transition t, and a pair (t, p_o) is used to denote an arc from a transition t to a destination place p_o. In Petri Nets, each place (or each transition) can have more than one input transitions (or more than one input places), and more than one output transitions (or more than one output places). A weight is an integer attached to each arc, and a weight function, w, maps an arc, (p_i, t) or (t, p_o), to its weight. If the weight is 1, it is usually omitted. A distribution of tokens in a Petri net is called a marking, which represents a state of the net.

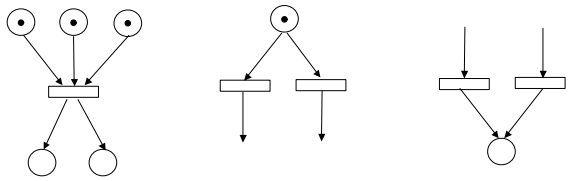

Figure 3. Example Petri Nets

First, we describe how a Petri net can be represented as an HTS. Each place of a Petri net is represented as a unique variable of an HTS, whose type is integer and whose value is the number of tokens in the place. An initial marking of a Petri net determines the HTS's initial variable-value assignment. Since a Petri net has no control states, the HTS's set of states is empty ($S = \emptyset$). Similarly, the set of events is empty ($E = \emptyset$). A transition is expressed in the form,

$$\langle cond, act \rangle$$

where *cond* is a predicate on the set of place variables that are the input places of the transition and the weight con-

Snapshot Element	Start of Macro-step	Executing Transition τ
$AV' =$	AV	$assign(AV, asn(\tau))$

en_states(ss, τ)	=	true
en_events(ss, τ)	=	true
en_cond(ss, τ)	=	$AV \models$ *cond*(τ)
macro_semantics	=	simple, diligent
pri(Γ)	=	Γ

Table 2. Template parameters for Petri Nets

stants, and *act* is zero or more actions that assign values to some place variables for the output places of the transitions.

The transitions of a Petri net determine its behaviour. A transition t is enabled if each of its input places i contains at least $w(p_i, t)$ tokens, where (p_i, t) denotes an arc from place p_i to transition t. The firing of transition t removes $w(p_i, t)$ tokens from each input place p_i of t and adds $w(t, p_o)$ tokens to each output place p_o of t. Consider case (a) in Figure 3, a transition with three input places p_1, p_2, p_3, two output places p_4, p_5, three input arcs with weights $w(p_1, t) = w(p_2, t) = w(p_3, t) = 1$, and two output arcs with weight $w(t, p_4) = w(t, p_5) = 1$. The transition is enabled since each of its three input places contains a token. After firing, a token is removed from each input place and a token is added to the output place. The transition is represented as an HTS transition with,

$$\begin{aligned} \textit{cond} &= v_1 \geq w(p_1, t) \\ &\quad \wedge\ v_2 \geq w(p_2, t) \\ &\quad \wedge\ v_3 \geq w(p_3, t) \\ \textit{asn} &= v_1 = v_1 - w(p_1, t);\ v_2 = v_2 - w(p_2, t); \\ &\quad v_3 = v_3 - w(p_3, t);\ v_4 = v_4 + w(t, p_4); \\ &\quad v_5 = v_5 + w(t, p_5); \end{aligned}$$

where variables v_1, v_2, v_3, v_4 and v_5 are the number of tokens in the places p_1, p_2, p_3, p_4, p_5 respectively.

At most one transition executes at a time in a Petri net and its firing changes the marking of a net, which is captured by the semantics of an HTS. Petri Nets do not use any composition operators. This description as an HTS captures all the cases of Petri Nets. For example, in case (b) of Figure 3, two transitions are in conflict. They share one input place and both are enabled, but only one of them can execute in a step. The one that executes disables the other until its input place is populated again.

A place can have an upper limit b on the number of tokens it can hold, which can affect the behaviour of cases such as (c) of Figure 3, where the firing of one transition may disable the other because they share a common output place. We can capture this in an HTS by adding the extra condition $v_i + w(t, p_i) \leq b$ to each transition that has an output with a bound.

Petri Nets use the simple, diligent option for macro-step semantics, since they do not have any events. There is no priority to the transitions ($pri(\Gamma) = \Gamma$). Table 2 shows the values for any relevant template parameters for Petri Nets. AV is the only snapshot element in use and the variable values are modified by the executing transition.

4. SDL

In this section, we present the template semantics for the Specification and Description Language (SDL), as defined in SDL88 [8]. An SDL specification has three types of components. SDL processes, the most basic components, are extended finite state machines that send and react to signals. SDL blocks contain multiple, concurrent processes, which are inter-connected by non-delaying, signal-passing routes; more abstract SDL blocks compose lower-level blocks that are inter-connected by delaying communication channels. An SDL system, the root component, is like an abstract SDL block that communicates with the environment.

An SDL process consists of states, variables, signals, decisions, and transitions. A transition has a source state; is triggered by an input signal; and has multiple possible actions and destination states, depending on decision points in the transition. We model each process as an HTS, whose states, variables, and events represent the process's states, variables, and signals, respectively. We model each conditional path through an SDL transition's actions as a sequence of HTS transitions: we add an auxiliary state for each decision construct, the initial transition is triggered by the input signal and leads to the first auxiliary state, and each subsequent transition is enabled by its source-state's decision-construct's condition and leads to a subsequent auxiliary state or to a destination state. Because these conditions are disjoint and complete, the HTS transitions are guaranteed to terminate in a distinct destination state.

Each process has an unbounded input queue to store the signals it receives from its signal routes. A signal is removed from the head of the queue (if not empty) when the process is in an SDL state. If the signal can trigger a transition, the process executes the transition; otherwise, the signal is discarded. To model the latter semantics, we add from every state a transition whose event is any signal that does not trigger an SDL transition from that state and whose sole effect on the snapshot is to remove the signal from the input queue. Variables are local to processes in SDL, and variable values are passed among processes only via signals.

The template semantics for an SDL's process is described in Table 3. SDL has stable macro-step semantics

Snapshot Element	Start of Macro-step	Executing Transition τ
CS'	CS	$(CS \setminus \mathit{exited}(\tau)) \cup \mathit{entered}(\tau)$
IE'	$enQ(IE, I)$	$enQ(deQ(IE, head(IE)), \mathit{gen}(\tau))$
AV'	AV	$assign(AV, asn(\tau))$
O'	ϕ	$O \cup \mathit{gen}(\tau)$

en_states(ss, τ)	=	*src* $\in CS$
en_events(ss, τ)	=	*trig*$(\tau) = head(IE)$
en_cond(ss, τ)	=	$AV \models$ *cond*
macro_semantics	=	stable
pri(Γ)	=	Γ

Table 3. Template parameters for SDL Process

$$N^{\mathit{sdl\_para}}_{\mathit{macro}}((\vec{ss}_1, \vec{ss}_2),\ (\vec{ss}'_1, \vec{ss}'_2),\ (C\vec{H}_1, C\vec{H}_2),\ (C\vec{H}'_1, C\vec{H}'_2),\ (C\vec{H}_{12}, C\vec{H}_{21}),\ (C\vec{H}'_{12}, C\vec{H}'_{21})) =$$

$$\exists \vec{iss}_1, \vec{iss}_2.\ \left[\begin{array}{ll} & N^1_{\mathit{macro}}(\vec{ss}_1, \vec{iss}_1, front(C\vec{H}_1, m) \cup front(C\vec{H}_{21}, m)) \\ \wedge & C\vec{H}'_2 = enQ(deQ(C\vec{H}_2, front(C\vec{H}_2, n)), \vec{iss}_1.O) \wedge C\vec{H}'_{12} = enQ(deQ(C\vec{H}_{12}, front(C\vec{H}_{12}, n)), \vec{iss}_1.O) \\ \wedge & N^2_{\mathit{macro}}(\vec{ss}_2, \vec{iss}_2, front(C\vec{H}_2, n) \cup front(C\vec{H}_{12}, n)) \\ \wedge & C\vec{H}'_1 = enQ(deQ(C\vec{H}_1, front(C\vec{H}_1, m)), \vec{iss}_2.O) \wedge C\vec{H}'_{21} = enQ(deQ(C\vec{H}_{21}, front(C\vec{H}_{21}, m)), \vec{iss}_2.O) \end{array}\right]$$

Figure 4. Macro-step semantics for SDL parallel composition

(because an SDL transition that has a decision construct maps to a sequence of HTS transitions) and no priority scheme. In every micro-step, the set of current states CS is updated according to the states the transition exits and enters. Variable values AV are updated according to data carried by the trigger event and by transition's sequence of variable assignments. The input queue for each process is modelled as a queue in snapshot element IE. SDL treats internal and external signals as the same, so a process's input queue holds both. Output signals are accumulated in O'. Enabling states (*en_states*) depend on the current states, and enabling conditions (*en_cond*) are conditions that evaluate to *true* given the current variable values. The enabling event (*en_events*) is the signal at the head of the queue.

Processes are composed using the parallel composition operator described in Section 2.3. Our template semantics assumes that events are broadcast to all components. We simulate SDL's point-to-point communication by assuming that every event contains its address, and a process enqueues an input signal only if the signal's address is the process's address. Parallel composition implements non-delaying communication among processes.

One or more blocks are composed into an SDL system specification. Blocks are connected with channels, which may nondeterministically delay signals.

We define a variant parallel-composition operator for composing SDL blocks. $C\vec{H}_1$ and $C\vec{H}_2$ represent the sets of delaying communication channels (queues) that pass signals among blocks in components one and two, respectively. The composition operator adds delaying channels $C\vec{H}_{12}$ that pass signals from blocks in component one to blocks in component two; channels $C\vec{H}_{21}$ act similarly. In each macro-step, some number of signals are removed from each channel set $C\vec{S}_1$ and $C\vec{S}_2$ and are treated as input to their respective components' processes; the components execute; and the output signals from each components processes are enqueued in the corresponding new and old channels.

5. SCR

In this section, we use our template to document the semantics of the Software Cost Reduction (SCR) notation, as defined in [4]. In SCR, a system specification is a collection of mathematical functions, represented as tables. A table specifies the states (called *modes*) and transitions for one state machine (called a *modeclass*), or it specifies assignments to one variable. Variables are partitioned into *monitored variables*, whose values are set by the environment; *controlled variables*, whose values are set by the system and are output to the environment; and *terms*, internal variables whose values are set by the system. An SCR specification defines functions only for modeclasses, terms, and controlled variables.

Each of the specification's functions is applied exactly once in every macro-step. The monitored variables are updated at the start of a macro-step. Each micro-step changes one of the specification's modes and variables based on either values at the beginning of the macro-step or new monitored-variable values and entered modes as a result of micro-steps. The order in which the functions are applied depends on the modes and terms used in their calculations: each function that depends on updated values of modes and terms must execute after the functions that perform those updates. These dependencies impose a partial order on the tables[1], which the composition operator follows when selecting the next function for "execution".

[1] The specification is ill-formed if the dependency graph has a cycle.

Mode	Event		
Off	**X**	@T(Dial=bake) WHEN[Temp<SetT]	@T(Dial=bake) WHEN[Temp≥SetT]
Heat	@T(Dial=off)	**X**	@T(Temp≥SetT)
Maintain	@T(Dial=off)	@T(Temp<(SetT-20))	**X**
Mode'=	Off	Heat	Maintain

Mode	Condition	
Off, Heat, Maintain	Temp≤100	Temp>100
Warning'=	off	on

Table 4. Partial SCR specification of a control system for an oven

Consider a simple control system for an oven. The system's monitored variables are

- *Dial :* $\{$*off, bake*$\}$ – the user-set command
- *SetT : integer* – the user–set temperature
- *Temp : integer* – the air temperature in the oven

The modes are *Off, Heat* (the oven is warming to temperature *SetT*), and *Maintain* (the system is maintaining an oven temperature around *SetT*). The system sets controlled variable *Warning* whenever the oven temperature is above 100° C, to warn children (and parents) when the oven temperature is hot enough to burn.

SCR uses two types of tables to express mathematical functions: *condition tables* and *event tables*. A **condition table** defines a case-based assignment to a variable. Table 4 shows a condition table for controlled variable *Warning*. The header information in the bottom row specifies the variable being assigned and its possible values. The Mode column decomposes the function's cases by mode value. Each table entry defines a transition that is enabled by the system being in one of the modes at the head of the row, and the table entry's condition and by any of the modes in the corresponding Mode-column entry. The action of the transition is to assign the table variable to the value specified at the bottom of the table entry's column. The table's conditions refer to values of modes and variables from the previous micro-step. For example, the *Warning* light is set to *On* whenever the oven temperature is above 100° C, regardless of the system's mode. The table's cases are mutually disjoint and are complete; hence, the function always assigns its variable to exactly one new value.

An **event table** defines a case-based assignment to a variable or a mode, where each case is enabled by being in a mode and by a change (*i.e.*, an event) in the values of modes and variables. Table 4 shows an event table for updating the system's modeclass. Event tables have a similar structure to condition tables, in that every table entry defines a transition that is triggered by the table entry's event when the system is in any of the modes in the corresponding Mode-column entry; the transition's action assigns the table variable (possibly a mode) to the value specified at the bottom of the table entry's column. The Mode column refers mode values from the start of the macro-step. The conditions used in the table entries also refer to variable-value assignments at the beginning of the macro-step. However, the events are any changes that have occurred since the beginning of the macro-step. A simple event $@T(cond)$ occurs if the condition $cond$ becomes true during the prefix of the macro-step that executes before the event table is evaluated. A conditional event *@T(cond1) WHEN [cond2]* occurs if simple event *@T(cond1)* occurs and the value of *cond2* was true at the start of the macro-step. The enabling conditions of an event table's transitions are mutually disjoint but not complete; hence, the table's function includes an implicit idle transition that re-assigns its variable to the variable's current value if none of the diligent transitions' enabling conditions are satisfied.

In instantiating our template to represent SCR semantics, we define separate templates for condition and event tables. In both cases, a table defines an HTS whose set of states S is the set of modes that appear in the table and whose set of variables V consists of the monitored variables, terms, and controlled variables that appear in the table. Each table entry defines a distinct transition whose action updates the variable being assigned or changes the mode (state). Transition events are modelled as changes in conditions between the values at the beginning of a macro-step and the current values after the last micro-step. The event set E is not needed in SCR.

Tables 5 and 6 are the template instantiations for SCR condition tables and event tables, respectively. In both cases, the environment's input EE are monitored-variable assignments, which update the values AV at the start of every macro-step. (Function *assign*(X, Y) takes variable-value assignments X and Y and updates the assignments in X with the assignments in Y.) Similarly, the system's output O are the assignments made to controlled variables. The main difference between the two template definitions is that the instantiation for event tables defines enabling states and enabling conditions in terms of both old and new values of modes and variables, and uses the auxiliary variables CS_a and AV_a to store the modes and variable values that hold at the start of the macro-step. Auxiliary snapshot element IE_a is used to ensure that each table's function executes exactly once per macro-step: We add to the triggering event of every transition the event *enabled*; IE_a is set to $\{$*enabled*$\}$ at the start of a macro-step, thereby enabling the table's transitions; and IE_a is set to $\emptyset$ when any of the table's transitions

Snapshot Element	Start of Macro-step	Executing Transition τ
$CS' =$	CS	$(CS \setminus \mathit{exited}(\tau)) \cup \mathit{entered}(\tau)$
$AV' =$	$assign(AV, EE)$	$assign(AV, asn(\tau))$
$O' =$	$\emptyset$	$O \cup asn(controlled(\tau))$
$IE'_a =$	$\{enabled\}$	$\emptyset$

$$\begin{aligned} \mathit{en\_states}(ss, \tau) &= source(\tau) \subset CS \\ \mathit{en\_events}(ss, \tau) &= \mathit{trig}(\tau) \subset IE_a \\ \mathit{en\_cond}(ss, \tau) &= AV \models \mathit{cond}(\tau) \\ \mathit{macro\_semantics} &= \text{stable} \\ \mathit{pri}(\Gamma) &= \Gamma \end{aligned}$$

Table 5. Template parameters for SCR condition tables

Snapshot Element	Start of Macro-step	Executing Transition τ
$CS' =$	CS	$(CS \setminus \mathit{exited}(\tau)) \cup \mathit{entered}(\tau)$
$AV' =$	$assign(AV, EE)$	$assign(AV, asn(\tau))$
$O' =$	$\emptyset$	$O \cup asn(controlled(\tau))$
$CS'_a =$	CS	CS_a
$AV'_a =$	AV	AV_a
$IE'_a =$	$\{enabled\}$	$\emptyset$

$$\begin{aligned} \mathit{en\_states}(ss, \tau) &= source(\tau) \subset (CS \cup CS_a) \\ \mathit{en\_events}(ss, \tau) &= AV_a \models \neg \mathit{trig}(\tau) \wedge \\ &\quad\; AV \models \mathit{trig}(\tau) \wedge \\ &\quad\; \mathit{trig}(\tau) \subset IE_a \\ \mathit{en\_cond}(ss, \tau) &= (AV \cup AV_a) \models \mathit{cond}(\tau) \\ \mathit{macro\_semantics} &= \text{stable} \\ \mathit{pri}(\Gamma) &= \Gamma \end{aligned}$$

Table 6. Template parameters for SCR event tables

executes, thereby disabling subsequent transitions until the next macro-step. Diligent transitions have priority over the idle transition in event tables; there are no idle transitions in a condition table. Lastly, because transitions' enabling conditions are mutually disjoint, the priority scheme is the identity function.

The composition operator $N_{micro}^{po\_interr}$ interleaves the table specifications, adhering to a given partial order PO on the HTSs, which we describe as a partial order on the transition sets $\vec{T}$. One HTS executes per micro-step, and the executing HTS chosen is such that no other function that has enabled transitions has a lower rank in the partial ordering. With each micro-step, the modes and variables values of all snapshots are updated with the assignment made by the executing transition.

6 Statecharts Variants

Statecharts, first introduced by Harel [2], are one of the most popular model-based specification notations. Many users have redefined subtle aspects of the statecharts semantics to better suit a particular problem, thereby creating a plethora of statecharts variants. For specifiers, it can be very difficult to understand the similarities and differences among these variants. von der Beeck's work comparing statecharts variants [17] is well cited because it provides a number of criteria for comparing variants. Our template parameters highlight the variants' differences in a more formal and succinct manner than previously possible.

Syntactically, all statecharts variants map into HTSs that are composed using parallel composition (*i.e.,* AND composition) and interrupt composition, which combines components via a set of *interrupt transitions* that pass control between the components. Table 7 shows the template parameter values for five popular statecharts variants (Harel's original semantics [2], Pnueli & Shalev [16], RSML [9], STATEMATE [3], and UML [14]). UML has simple, diligent macro-step semantics; the other statecharts variants have stable macro-step semantics.

The CS, CS_a, and *en_states* parameters capture the differences in which states can enable transitions. In Harel and in Pnueli & Shalev, transitions can be taken in a macro-step only if their source state was a current state at the start of the macro step. The snapshot element CS_a contains this set. RSML, STATEMATE, and UML do not have this restriction, so it possible for a macro-step to have an infinite loop. Variable values (AV, AV_a) are handled similarly.

Harel and Pnueli & Shalev allow external events to trigger transitions throughout a macro-step; parameter EE_a holds these events. In RSML and STATEMATE, external events can trigger transitions only in the first micro-step. We assume that timeout events are external events

Similarly, Harel and Pnueli & Shalev allow internal events generated in a micro-step to trigger future transitions in the same macro-step; hence, parameter IE accumulates generated events. RSML and STATEMATE, allow only events generated in the last micro-step to trigger a transition. In UML, each object has an event queue that emits one event per (simple) macro-step. Note that transitions may trigger on implicit internal events, such as the entering and exiting of states or changes in conditions or in variable values. Parameter IE uses function *ev_gen* to return all explicit and implicit internal events generated in the micro-step where transition τ is taken in snapshot ss.

Many startcharts variants allow transitions to trigger on event expressions, such as negated events (*i.e.,* lack of an

$$N^{po\_interr}_{micro}((s\vec{s}_1, s\vec{s}_2),\ (s\vec{s}_1{}', s\vec{s}_2{}'),\ \ (\vec{\tau}_1, \vec{\tau}_2))\ \ PO =$$

$$\left[\begin{array}{l} \quad \exists!\, T.\ \left(\vec{\tau}_1 \in T \in \vec{T}_1\ \wedge\ \left(\begin{array}{l} \neg\, \exists U \in \vec{T}_1\,.\,(\mathit{enabled\_trans}(s\vec{s}_1, U)) \neq \emptyset\ \wedge\ PO(U,T) \quad \wedge \\ \neg\, \exists U \in \vec{T}_2\,.\,(\mathit{enabled\_trans}(s\vec{s}_2, U)) \neq \emptyset\ \wedge\ PO(U,T) \end{array}\right)\right) \\ \wedge \\ \quad N^1_{micro}(s\vec{s}_1, s\vec{s}_1{}', \vec{\tau}_1)\ \ \wedge\ \ \vec{\tau}_2 = \emptyset\ \ \wedge\ \ s\vec{s}_2{}' = s\vec{s}_2\ |^{CS}_{\mathit{n\_cur\_states}(s\vec{s}_2, \vec{\tau}_1)}|^{AV}_{\mathit{n\_var\_val}(s\vec{s}_2, \vec{\tau}_1)} \end{array}\right]$$

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

Figure 5. Micro-step semantics for SCR composition

event) or disjunctions of events. To handle negated events, we distinguish between trigger events *pos*(τ) and negated trigger events *neg*(τ). Pnueli & Shalev do not allow two transitions to execute in the same macro-step, if one is triggered by negated event *not* a and the other is triggered by subsequently generated event a; they call this scenario a *global inconsistency*. To model Pnueli & Shalev's semantics, we use IE_a to accumulate the events that trigger transitions in the macro-step. Subsequent transitions are enabled only if their actions are consistent (*consis*) with this set IE_a (see Table 7's definition for *en_events*). Harel's statecharts and STATEMATE allow *global inconsistencies*; RSML does not allow negated events; UML cannot exhibit *global inconsistency* because it has simple macro-step semantics. We treat disjunctive events as a notational convenience for combining transitions that have similar actions.

Harel, Pnueli & Shalev, and RSML place no priority scheme on transitions. STATEMATE gives priority to transitions whose scope has the lowest rank:

$$\{\tau \in \Gamma | \forall t \in \Gamma.\ rank(\mathit{scope}(\tau)) \leq rank(\mathit{scope}(t))\}$$

UML favours transitions with the highest-rank source state:

$$\{\tau \in \Gamma | \forall t \in \Gamma.\ rank(source(\tau)) \geq rank(source(t))\}$$

Some of the statecharts features not modelled here (*i.e.*, compound transitions, AND/OR tables) are simply notational conveniences. Other features, such as history states, and method calls (UML), would require an extension to our HTS model, which is outside of the scope of this paper.

7 Conclusion

We have demonstrated that our template approach for describing model-based notations is expressive enough to define the semantics of Petri Nets, SDL, and SCR. A notation's template definition is succinct and better facilitates comparison among notations than traditional descriptions, because the template separates the different concerns of the step semantics. This separation is particularly helpful when comparing notation variants, as demonstrated for the statecharts variants. Template definitions of some notations (*e.g.,* SCR) stretch the intended uses of some of the snapshot elements, and it is not clear whether the resulting definition is easier to understand than a traditional definition, although we can compare the semantics more easily.

We are currently working on using template definitions of notations to generate notation-specific analysis tools, such as model checkers. We believe this work will make it possible to create formal analysis tools for custom notations with considerably less effort than previously possible.

References

[1] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.

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

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

[4] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Soft. Eng. Meth.*, 5(3):231–261, 1996.

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

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

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

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

[9] 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), September 1994.

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

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

[12] T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.

[13] J. Niu, J. M. Atlee, and N. A. Day. Composable semantics for model-based notations. In *FSE*, pages 149–158, 2002.

[14] Object Management Group. Unified Modelling Language (UML), v1.4, 2001. Internet: www.omg.org.

[15] J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[16] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Proceedings of the Symposium on Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264. Springer-Verlag, 1991.

[17] 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.

Parameter	Harel [2]	Pnueli [16]	RSML [9]	STATEMATE [3]	UML [14]
CS(start)	CS	CS	CS	CS	CS
CS'	$(CS \setminus \mathit{exited}(\tau)) \cup \mathit{entered}(\tau)$				
IE(start)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$enQ(IE, I)$
IE'	$IE \cup \mathit{ev\_gen}(ss, \tau)$	$IE \cup \mathit{ev\_gen}(ss, \tau)$	$\mathit{ev\_gen}(ss, \tau)$	$\mathit{ev\_gen}(ss, \tau)$	$enQ(deQ(IE, head(IE)), \mathit{ev\_gen}(\tau))$
AV(start)	AV	AV	AV	AV	AV
AV'	$\mathit{assign}(AV, \mathit{resolve\_conflicts}(\mathit{asn}(\tau)))$				
O(start)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
O'	$(O \cup \mathit{ev\_gen}(ss, \tau)) \setminus \mathit{trig}(\tau)$				$O \cup \mathit{ev\_gen}(ss, \tau)$
CS_a(start)	CS	CS	n/a	n/a	n/a
CS'_a	$CS_a \setminus (\mathit{exited}(\tau) \cup \mathit{entered}(\tau))$	$CS_a \setminus (\mathit{exited}(\tau) \cup \mathit{entered}(\tau))$	n/a	n/a	n/a
IE_a(start)	$\emptyset$	$\emptyset$	n/a	n/a	n/a
IE'_a	n/a	$IE_a \cup \mathit{trig}(\tau)$	n/a	n/a	n/a
AV_a(start)	AV	AV	n/a	n/a	n/a
AV'_a	AV_a	AV_a	n/a	n/a	n/a
EE_a(start)	I	I	I	I	n/a
EE'_a	EE_a	EE_a	$\emptyset$	$\emptyset$	n/a
$\mathit{en\_states}(ss, \tau)$	$\mathit{src}(\tau) \in CS_a$	$\mathit{src}(\tau) \in CS_a$	$\mathit{src}(\tau) \in CS$	$\mathit{src}(\tau) \in CS$	$\mathit{src}(\tau) \in CS$
$\mathit{en\_events}(ss, \tau)$	$\mathit{pos}(\tau) \subseteq IE \cup EE_a \ \wedge$ $(\mathit{neg}(\tau) \cap (IE \cup EE_a)) = \emptyset$	$\mathit{pos}(\tau) \subseteq IE \cup EE_a \ \wedge$ $(\mathit{neg}(\tau) \cap (IE \cup EE_a)) = \emptyset$ $\wedge \mathit{consis}(IE_a \cup \mathit{ev\_gen}(ss, \tau))$	$\mathit{trig\_ev}(\tau) \subseteq IE \cup EE_a$	$\mathit{trig\_ev}(\tau) \subseteq IE \cup EE_a$	$\mathit{trig\_ev}(\tau) = head(IE)$
$\mathit{en\_cond}(ss, \tau)$	$AV_a \models \mathit{cond}(\tau)$	$AV_a \models \mathit{cond}(\tau)$	$AV \models \mathit{cond}(\tau)$	$AV \models \mathit{cond}(\tau)$	$AV \models \mathit{cond}(\tau)$
macro_semantics	stable	stable	stable	stable	simple, diligent
$\mathit{pri}(\Gamma)$	no priority	no priority	no priority	outer scope higher	inner source higher

Table 7. Template parameters for statecharts variants ("n/a" means not applicable)