

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

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 distinctive semantics (e.g., which states can enable transitions) are specified as parameters. In this paper, we demonstrate the template's generality by using it to document the semantics of SCR, SDL, and Petri Nets. 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 [14, 15]. Our goal with the template is to make it easier to document the semantics of notations by focusing on how notations differ and by requiring a user to describe only these differences. In our method, a parameterized template pre-defines the semantics that are common among 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 latter three predicates. Composition operators are defined separately from the step 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 eight popular specification notations: CSP [7], CCS [12], LOTOS [8], basic transition systems (BTS) [11], a subset of SDL88 [9], and three variants of statecharts [3, 4, 10]. In this paper, we demonstrate the template's generality by using it to describe the semantics of Petri Nets, SCR, and a larger subset of SDL; these three languages 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 statecharts variants (original statecharts [3], Pnueli and Shalev's statecharts [18], RSML [10], STATEMATE [4], and UML state models [16]). 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 describing model-based notations. The approach separates the definition of a notation's step semantics from the definitions 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 [11] and statecharts [2]. An HTS supports no concurrency. Composition operators specify how collections of HTSs execute concurrently, transferring control to one another and exchanging events and data.

2.1. Syntax of HTS

A hierarchical transition system (HTS) is an 8-tuple, $\langle S, S^I, S^F, S^H, E, V, V^I, T \rangle$. S is a finite set of states, $S^I \subseteq S$ is the set of initial states, $S^F \subseteq S$ is the set of final basic states, and S^H is the state hierarchy¹. E is a finite set of events, including both internal and external events. V is a finite set of typed data variables, whose initial values satisfy predicate 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,

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

where $src, dest \subseteq S$ are the transition’s source and destination states, respectively; $trig \subseteq E$ is zero or more triggering events; $cond$ is a predicate over V ; act is zero or more actions that generate events and assign values to some variables in V ; and $prty$ is the transition’s priority. A notation need not use all of the transition elements. We use identifiers $S, S^I, S^F, S^H, E, V, V^I$, and 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 S^H defines a partial ordering on states, with the root state as the minimal element and basic states as maximal elements. We use helper functions to access static information about an HTS:

- $src(\tau)$ are the source states of τ .
- $dest(\tau)$ is the destination state of τ .
- $trig(\tau)$ are the events that trigger τ .
- $pos(\tau)$ are events whose occurrence trigger τ .
- $neg(\tau)$ are negative events whose non-occurrence trigger τ .
- $cond(\tau)$ is τ ’s predicate guard condition
- $gen(\tau)$ 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.

¹A *state* in an HTS refers to a control state from which a transition can originate or can end. It differs from the concept of a state in a classical finite state machine (FSM) [1], in that a traditional FSM is always in exactly one state, whereas an HTS is always in a non-empty set of states.

- $rank(s)$ is the distance between state s and the root state, where $rank(root) = 0$.
- $entered(s)$ are the states entered when s is entered, including s ’s ancestors and its descendants’ default states.

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 $\langle CS, IE, AV, O, CS_a, IE_a, AV_a, I_a \rangle$, where CS is the set of current states ($CS \subseteq S$), IE is the set of current internal events ($IE \subseteq E$), and AV is the set of 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 I_a are auxiliary variables that accumulate data about states, variable values, internal events, 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 a 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 upon reaching a **stable snapshot**, in which no transition is enabled; a stable snapshot ends the HTS’s reaction to one set of input I and begins the start of a new macro step.

The step-semantics of a notation is a parameterized macro-step, N_{macro} , defined formally in [14]. The template parameters specify how enabled transitions are determined, how an executing transition affects the value of the snapshot, and how the snapshot is initialized at the beginning of a macro-step. Parameters also determine the type of macro-step and how priority among transitions is handled.

The template parameters are organized in Table 1 by language construct. For example, the five state-related parameters work together to record the set of current states and to determine the set of enabling states: *reset_CS*, *reset_CS_a*

Affected Snapshot Element	Start of Macro-step	Micro-step τ
states	$reset_CS(ss, I) : CS^i$	$next_CS(ss, \tau, CS^i)$
	$reset_CS_a(ss, I) : CS_a^i$	$next_CS_a(ss, \tau, CS_a^i)$
	$en_states(ss, \tau)$	
events	$reset_IE(ss, I) : IE^i$	$next_IE(ss, \tau, IE^i)$
	$reset_IE_a(ss, I) : IE_a^i$	$next_IE_a(ss, \tau, IE_a^i)$
	$reset_I_a(ss, I) : I_a^i$	$next_I_a(ss, \tau, I_a^i)$
	$en_events(ss, \tau)$	
variables	$reset_AV(ss, I) : AV^i$	$next_AV(ss, \tau, AV^i)$
	$reset_AV_a(ss, I) : AV_a^i$	$next_AV_a(ss, \tau, AV_a^i)$
	$en_cond(ss, \tau)$	
outputs	$reset_O(ss, I) : O^i$	$next_O(ss, \tau, O^i)$
Additional Parameters	$macro_semantics$ $pri(\Gamma) : 2^T$ $resolve(AV_1, AV_2, asnAV)$	

Table 1. Parameters to be provided by template user

clean out irrelevant state information accumulated in the previous macro-step, and incorporate any state information from the inputs I ; en_states tests whether transitions are enabled with respect to the state information; and $next_CS$, $next_CS_a$ specify how a transition τ 's actions affect the possible next current states (CS^i) and next auxiliary states (CS_a^i). The parameters related to events and variables play comparable roles, with enabling events depending not only on internal event information, but also on external events.

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*. The function $pri(\Gamma)$ for a set of transitions Γ implements the notation's priority scheme (e.g., sub-state behaviour could override super-state behaviour). The template parameter $resolve(AV_1, AV_2, asnAV)$ captures different notations' policies for resolving conflicts among variable-value assignments (e.g., non-deterministic choice). This predicate is true if $asnAV$ is a non-conflicting set of variable-value assignments resulting from resolution of the assignments in AV_1 and AV_2 .

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

$$N_{micro}^{op}((\vec{s}\vec{s}_1, \vec{s}\vec{s}_2), (\vec{\tau}_1, \vec{\tau}_2), (\vec{s}\vec{s}'_1, \vec{s}\vec{s}'_2))$$

which relates snapshot collections $\vec{s}\vec{s}_1$ and $\vec{s}\vec{s}'_1$ if component one executes transitions $\vec{\tau}_1$, and relates snapshot collections $\vec{s}\vec{s}_2$ and $\vec{s}\vec{s}'_2$ when component two executes transitions $\vec{\tau}_2$, all in the same micro-step. The definition of N_{micro}^{op} is parameterized by the micro-step semantics of the two components, N_{micro}^1 and N_{micro}^2 . The definition of the composition operator determines how the effects of the changes in each component are shared, and how transfer of control from one component to another can occur. We represent these effects by using substitution to modify snapshots: $ss \mid_y^x$ is a snapshot that is equal to snapshot ss , except that element x has value y . Substitution over a set of snapshots $\vec{s}\vec{s} \mid_y^x$ defines a substitution to each snapshot in $\vec{s}\vec{s}$.

We use helper predicates *update*, *communicate*, and *communicate_vars* to perform common substitutions that realize the sharing of events and variable assignments. Predicate *update* is used when only one component executes in the micro-step: it passes to the non-executing component those events generated by executing component. Predicate *communicate* is used when both components execute in the micro-step: it passes to one component those events generated by the other component. Predicate *communicate_vars* handles the communication of shared variable values, such that conflicts among variable assignments are resolved, and all resulting snapshots have the same value associated with a shared variable. Predicate *communicate_vars* uses template parameter *resolve* to implement the notations' policy for resolving variable-value assignments. In general, composition operators use the template parameters, so that they adhere to their components' semantics for updating snapshot elements.

Condition Table

Mode	Condition	
	True	X
Off	Temp < 175	Temp ≥ 175
Heat, Maintain	Temp < 175	Temp ≥ 175
DisplayTemp' =	{blank}	[Temp/25] × 25

Event Table

Mode	Event		
	X	@T(Dial=bake) WHEN[Temp < SetT]	@T(Dial=bake) WHEN[Temp ≥ SetT]
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

Figure 2. Partial SCR specification of a control system for an oven

condition *cond* becomes *true* in that step.² A **simple conditioned event**, expressed as $@T(cond1) \text{ WHEN } [cond2]$, occurs in a step if its basic event $@T(cond1)$ occurs in the step and if its **WHEN condition** *cond2* evaluates to *true* with respect to variable values that held at the start of the step. A **conditioned event** is the conjunction and disjunction of multiple simple conditioned events.

We will use a simple specification of an oven-control system to help describe the SCR notation. The oven's monitored variables are

- *Dial* : {*off*, *bake*} – the user-set command
- *SetT* : [0..550] – the user-set temperature
- *Temp* : [0..600] – the air temperature in the oven

The modes are *Heat* (the oven is warming to temperature *SetT*), *Maintain* (the system is maintaining an oven temperature around *SetT*), and *Off*. The system displays the oven temperature via controlled variable *DisplayTemp*, whenever the oven is on and its temperature is 175° F or greater. To avoid rapid fluctuations in the displayed value, the value is rounded down to the nearest number divisible by 25.

SCR uses two types of tables to express mathematical functions: *condition tables* and *event tables*³. A **condition table** defines a case-based assignment to a variable. Figure 2 shows is a condition table for controlled variable *DisplayTemp*. The bottom row of the table specifies the variable being assigned and its possible values. The Mode column decomposes the function's definition by mode value. Each of the table entries defines a conditional assignment to the variable: if the current mode is one of the modes listed in the row's Mode-column entry, and if the table entry's condition evaluates to *true*, then the variable is assigned to the value listed at the bottom of the table entry's column. For example, *DisplayTemp* displays nothing if the oven is in mode *Off* or if the oven temperature is below 175° F. A table entry

²Basic event $@F(cond)$, representing a condition becoming *false*, can be expressed as $@T(\neg cond)$. Basic event $@C(cond)$ is equivalent to the expression $@T(cond) \vee @F(cond)$. Basic event $@C(x)$, where variable *x* is not necessarily boolean, occurs in a step if variable *x* changes value in that step. Henceforth, we assume that all basic events are of type $@T(cond)$ or $@C(x)$, without loss of generality, although for simplicity we'll refer in general discussion only to basic events of type $@T(cond)$.

³A *mode transition table*, which defines assignments to a modeclass variable, is a type of event table.

of **X** denotes an impossible case in the function definition. Condition-table expressions always refer to current values of modes and variables (*cf.* expressions in event-table entries). A condition table's cases must be both mutually disjoint and complete; hence, each condition table defines a total function.

An **event table** is similar to a condition table, in that the table entries define mode-partitioned, conditional assignments to a single variable. Figure 2 shows an event table for mode transitions in the oven control-system specification. Unlike in condition tables, event-table entries are conditioned events over previous and current variable values, and the modes in the Mode column are considered additional WHEN conditions in these events. A particular table entry applies if one of the modes listed in the row's Mode-column entry held at the start of the step, if the entry's basic events occur in the step, and if the entry's WHEN conditions held at the start of the step; then the table's variable is assigned to the value listed at the bottom of the table entry's column. A table entry of **X** denotes an impossible case in the function definition. Thus, the event table in Figure 2 is equivalent to the following, more traditional representation of a mathematical function, where unprimed variables refer to values at the start of the macro-step and primed variables refer to values in the current snapshot:

$$Mode' = \begin{cases} Off & \text{if } (Mode=Heat \wedge Dial \neq off \wedge Dial' = off) \vee \\ & (Mode=Maintain \wedge Dial \neq off \wedge Dial' = off) \\ Heat & \text{if } (Mode=Off \wedge Dial \neq bake \wedge Dial' = bake \wedge \\ & Temp < SetT) \vee \\ & (Mode=Maintain \wedge Temp \not< (SetT-20) \wedge \\ & Temp < (SetT'-20)) \\ \dots & \dots \end{cases}$$

In the last case above, the modeclass transitions to mode *Heat* if the previous mode was *Maintain*, the oven temperature was within 20 degrees of the user-set temperature at the start of the step, and the oven temperature is now at least 20 degrees cooler than the user-set temperature. The cases in an event table must be mutually disjoint, but are not necessarily complete; hence, the function includes an implicit *else clause* that re-assigns the function's variable to the variable's current value if none of the specified cases is satisfied. Thus event-table functions are also total functions.

In mapping the syntax of SCR to the syntax of HTS, we

Snapshot Element	Start of Macro-step	Micro-step τ
AV	$assign(ss.AV, I)$	$AV' = assign(ss.AV, eval(ss, asn(\tau)))$
O	\emptyset	$O' = controlled(V) \triangleleft eval(ss, asn(\tau))$

$$\begin{aligned}
en_states(ss, \tau) &= true \\
en_events(ss, \tau) &= true \\
en_cond(ss, \tau) &= ss.AV \models cond(\tau) \\
macro_semantics &= simple, diligent \\
pri(\Gamma) &= \Gamma
\end{aligned}$$

Table 2. Template parameters for SCR condition tables

Snapshot Element	Start of Macro-step	Micro-step τ
AV	$assign(ss.AV, I)$	$AV' = assign(ss.AV, eval(ss, asn(\tau)))$
O	\emptyset	$O' = controlled(V) \triangleleft eval(ss, asn(\tau))$
AV_a	$ss.AV$	$AV'_a = ss.AV_a$

$$\begin{aligned}
en_states(ss, \tau) &= true \\
en_events(ss, \tau) &= \\
&\forall e \in trig(\tau). [\\
&\quad (e = @T(c) \Rightarrow (ss.AV_a \models \neg c \wedge ss.AV \models c)) \\
&\quad \wedge (e = @C(x) \Rightarrow (ss.AV_a(x) \neq ss.AV(x)))] \\
en_cond(ss, \tau) &= ss.AV_a \models cond(\tau) \\
macro_semantics &= simple, diligent \\
pri(\Gamma) &= \Gamma
\end{aligned}$$

Table 3. Template parameters for SCR event tables

map each SCR table to a distinct HTS, where

- V is the set of specification variables that appear in the table, including modeclasses. V is partitioned into sets *monitored*, *terms*, and *controlled*.
- V_I is a predicate specifying initial modes.
- T is the table's set of conditional assignments. In condition tables, an entry defines zero or one HTS transition, whose trigger event is empty, whose enabling conditions are the entry's condition plus the entry's mode set, and whose action is the entry's assignment to the table's variable. In event tables, an entry defines zero or more HTS transitions, one for each disjunct in the entry's conditioned event; the transition's trigger events are the disjunct's basic events, its enabling conditions are the disjunct's WHEN conditions plus the entry's mode set, and its action is the entry's assignment to the table's variable. A table entry of \mathbf{X} maps to zero HTS transitions.

In the event table in Figure 2, the mode transition from mode *Off* to mode *Heat* maps to HTS transition

$\langle -, @T(Dial=bake), Mode=Off \wedge Temp < SetT, Mode:=Heat, -, - \rangle$

Our mapping of SCR syntax does not use HTS states (S, S_I, S_F, S_H), or named events E .

To express SCR step-semantics, we instantiate separate templates for condition tables and event tables. In our template instantiation for condition tables (Table 2),

- AV is the set of current variable values
- Environment input I is a monitored-variable assignment that updates the variable values in AV at the start of every step. (Function $assign(X, Y)$ takes variable-value assignments X and Y , and it updates the assignments in X with the assignments in Y , ignoring assignments in Y to variables not in X . Function $eval(ss, a)$ evaluates assignment a using variable values $ss.AV$.)

- System outputs O are assignments to controlled variables. (Function $controlled(V) \triangleleft eval(asn(\tau))$ uses the Z domain-restriction operator \triangleleft to ensure that the HTS outputs a variable assignment only if the variable is among the specification's controlled variables.)
- Predicates en_states and en_events are vacuously *true*, because there are no states or events.
- Predicate en_cond evaluates a transition enabling conditions with respect to current variable values in AV .
- The step-semantics is simple, diligent macro-step semantics, where a macro-step is one micro-step (*i.e.*, the execution of one of the HTS's transitions).
- There is no priority scheme among transitions (*i.e.*, pri is the identity function) because the transitions in an HTS are mutually disjoint.

The template instantiation for event tables (Table 3) is similar, except that events and conditions are evaluated with respect to both previous and current variable values. Hence,

- Auxiliary variable AV_a is used to store variable values that hold at the start of the step.
- Predicate en_events tests whether a transition's triggering events have all occurred since the start of the step. With respect to events of type $@C(x)$, it tests whether variable x has changed value since the start of the step. (Notation $ss.AV(x)$ returns the value of variable x in $ss.AV$.)
- Predicate en_cond tests whether a transition's WHEN conditions evaluated to *true* at the start of the step.

SCR's sole composition operator is functional composition: each micro-step of an SCR specification is the composition of HTS micro-steps, one from each of the tables' HTSs; the order of composition is with respect to a provided, static, total order TO on the HTSs. The provided total

$$\begin{aligned}
N_{micro}^{fun_comp}(\vec{s}\vec{s}, \vec{\tau}, \vec{s}\vec{s}') TO = \\
\vec{s}\vec{s}' = N_{micro}^{TO[n]} \left(\dots \left(N_{micro}^{TO[2]} \left(N_{micro}^{TO[1]}(\vec{s}\vec{s}, \tau_1), \tau_2 \right), \dots \right), \tau_n \right) \wedge \\
\vec{\tau} = \{\tau_1, \tau_2, \dots, \tau_n\} \wedge \forall i \leq n. \left[\tau_i = \text{enabled_trans} \left(N_{micro}^{TO[i-1]} \left(\dots \left(N_{micro}^{TO[1]}(\vec{s}\vec{s}, \tau_1), \dots \right), \tau_{i-1} \right), T_{TO[i]} \right) \right]
\end{aligned}$$

Figure 3. Micro-step semantics for SCR composition

order TO must adhere to the partial order on the specification's functions, which is imposed by the functions' variable dependencies. Any topological sort will do, since all will result in equivalent compositions. The tables' partial order, a representative total order TO , and cycle detection are all calculated off-line using def-use analysis.

Our modelling of SCR's composition operator $N_{micro}^{fun_comp}$, shown in Figure 3, determines whether a set of transitions $\vec{\tau}$ executing with respect to a collection of snapshots $\vec{s}\vec{s}$ (one snapshot per component HTS) will result in a collection of next snapshots $\vec{s}\vec{s}'$. In this definition, we use a functional representation of the HTSs' N_{micro} steps (the HTSs' micro-step semantics are guaranteed to be functions, because the SCR tables are all functions). Let TO be indexed 1 through n , which is the number of tables being composed. The first line says that the collection of next snapshots $\vec{s}\vec{s}'$ results from applying each HTS's micro-step function N_{micro} in the total order specified by TO , starting from snapshots $\vec{s}\vec{s}$. The second line ensures that the transition set $\vec{\tau}$ contains exactly one transition from each of the HTS's set of transitions T , and that each transition τ_i is enabled when its HTS executes.

We note that there are multiple ways to structure a notation's template semantics, just as there are multiple ways to structure a notation's operational semantics. For example, SCR users who view modeclasses as state machines might have represented modes as states. Alternatively, a user could view any dependent variable defined by an SCR event table as a stable machine. We chose the above representation because it more closely matches existing descriptions of the semantics of SCR, which treat modeclasses as distinguished variables. In general, we have not yet explored the methodological issues of how best to structure a notation's template semantics.

4. SDL

In this section, we present the template semantics for the Specification and Description Language (SDL), as defined in SDL88 [9]. 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 an abstract SDL block that communicates with the environment.

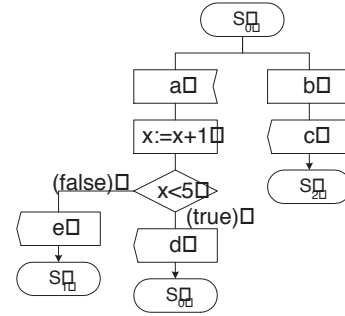


Figure 4. An SDL process example

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 (variable assignments and output signals) and destination states, depending on decision points in the transition. 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. Figure 4 shows a simple example of an SDL process. In state s_0 , an input signal a at the head of the queue can trigger a transition that increments the value of variable x by 1, outputs signal d if the condition $x > 5$ is true, or outputs a signal e if the condition is false. Another transition from state s_0 to state s_3 executes if the input signal is b , sending output signal c .

Syntactically, we map each SDL process to one 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 set of HTS transitions: we create an auxiliary state for each decision construct, and we create an HTS transition for every segment of an SDL transition between an SDL source state and a decision construct, between two decision constructs, or between a decision construct and an SDL state. The initial transition is triggered by the input signal and leads to

Snapshot Element	Start of Macro-step	Micro-step τ
CS	CS	$CS' = dest(\tau)$
IE	$append(ss.IE, I)$	if $trig(\tau) = \emptyset$ then $IE' = append(ss.IE, gen(\tau))$ else $IE' = append(tail(ss.IE), gen(\tau))$
AV	AV	let $AVp = assign(ss.AV, parm(trig(\tau)))$ in $AV' = assign(AVp, eval(ss, asn(\tau)))$
O	\emptyset	$O' = ss.O \cup gen(\tau)$

$en_states(ss, \tau)$	$\equiv src(\tau) \subseteq ss.CS$
$en_events(ss, \tau)$	$\equiv trig(\tau) \subseteq \{head(ss.IE)\}$
$en_cond(ss, \tau)$	$\equiv ss.AV \models cond(\tau)$
$macro_semantics$	$\equiv stable$
$pri(\Gamma)$	$\equiv \Gamma$

Table 4. Template parameters for SDL Process

$$\begin{aligned}
& N_{macro}^{SDL_block}((s\bar{s}_1, s\bar{s}_2), I, (s\bar{s}'_1, s\bar{s}'_2)), (\bar{C}H_{E1}, \bar{C}H'_{E1}), (\bar{C}H_{E2}, \bar{C}H'_{E2}), (\bar{C}H_{1E}, \bar{C}H'_{1E}), (\bar{C}H_{2E}, \bar{C}H'_{2E}), (\bar{C}H_{12}, \bar{C}H'_{12}), (\bar{C}H_{21}, \bar{C}H'_{21}) = \\
& \exists m, n, p, q. \left[\begin{array}{l}
N_{macro}^1(s\bar{s}_1, front(\bar{C}H_{E1}, m) \cup front(\bar{C}H_{21}, n), s\bar{s}'_1) \wedge \bar{C}H'_{21} = append(last(\bar{C}H_{21}, n), s\bar{s}'_2.O) \\
\wedge \bar{C}H'_{E1} = append(last(\bar{C}H_{E1}, m), I) \wedge \bar{C}H'_{1E} = append(\bar{C}H_{1E}, s\bar{s}'_1.O) \\
\wedge N_{macro}^2(s\bar{s}_2, front(\bar{C}H_{E2}, p) \cup front(\bar{C}H_{12}, q), s\bar{s}'_2) \wedge \bar{C}H'_{12} = append(last(\bar{C}H_{12}, q), s\bar{s}'_1.O) \\
\wedge \bar{C}H'_{E2} = append(last(\bar{C}H_{E2}, p), I) \wedge \bar{C}H'_{2E} = append(\bar{C}H_{2E}, s\bar{s}'_2.O)
\end{array} \right] \quad (* \text{ both take a step } *)
\end{aligned}$$

Figure 5. Macro-step semantics for SDL parallel composition

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. For example, the transition triggered by signal a in Figure 4 is split into the HTS transitions t_1, t_2 , and t_3 , where t_1 is from state s_0 to an auxiliary state s_0^a (at decision point $x < 5$), t_2 is from state s_0^a to s_1 , and t_3 is from state s_0^a to s_0 . Because the decision construct's conditions are disjoint and complete, the HTS transitions are guaranteed to terminate in a distinct destination state.

To correctly model the removal of signals from the input queue, for each $\langle \text{SDL state, event} \rangle$ pair that does not trigger an SDL transition, we create an HTS transition whose sole effect on the snapshot is to remove the signal from the queue. We use the following operations on queues:

- $head(Q)$ returns the first element of the queue
- $tail(Q)$ removes the first element from the queue
- $front(Q, k)$ returns the first k elements of the queue
- $last(Q, k)$ removes the first k elements of the queue
- $append(Q, e)$ adds events in e to the end of the queue

The template semantics for an SDL process is provided in Table 4.

- In every micro-step, the new current state is the transition's destination state.
- The input queue for each process is modelled as a queue in snapshot element IE . SDL processes do not distinguish between internal and external signals, so a process's input queue holds both. If an executing transition has a trigger event, the trigger event is removed from the head of the input queue and the transition's generated events are appended to the end of the queue;

otherwise (it is a transition from a decision construct), the only change to the input queue is the addition of the transition's generated events.

- Variables are local to processes in SDL, but their values may be passed among processes via signals' parameters. Variable values AV are updated according to data carried by the trigger event (function $parms$ returns a set of mappings from local variables to signal parameters); the variables are subsequently updated by the transition's sequence of variable assignments.
- Output signals are accumulated in O .
- Predicate (en_states) tests that transition's source states are in the set of the current states CS .
- Predicate en_cond tests a transition's enabling conditions with respect to current variable values in AV .
- Predicate (en_events) tests that a transition's trigger event matches the signal at the head of the input queue.
- SDL has stable macro-step semantics (*i.e.*, a macro-step is a sequence of HTS micro steps from a source SDL state through auxiliary decision-point states to an SDL destination state).
- SDL has no priority scheme.

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.

We define a variant parallel-composition operator for composing SDL blocks in Figure 5. The user provides the composition operator with channel sets $\vec{C}H_{E1}$, $\vec{C}H_{E2}$, $\vec{C}H_{1E}$, $\vec{C}H_{2E}$, $\vec{C}H_{12}$, and $\vec{C}H_{21}$. Each channel set is a collection of uni-directional, delaying, communication channels (queues) whose source is denoted by the left subscript and whose destination is denoted by the right subscript. Channel sets $\vec{C}H_{E1}$ and $\vec{C}H_{E2}$ represent the channels that pass signals from the environment to blocks in components one and two, respectively. Channel sets $\vec{C}H_{1E}$ and $\vec{C}H_{2E}$ represent the channels that pass signals from blocks in components one and two to the environment, respectively. Channel sets $\vec{C}H_{12}$ pass signals from blocks in component one to blocks in component two; channel sets $\vec{C}H_{21}$ act similarly. In each macro-step, the inputs I are appended to the ends of every channel in $\vec{C}H_{E1}$ and $\vec{C}H_{E2}$ (recall that the leaf processes will enqueue an event only if the signal's address is the process's address); some number of signals are removed from the fronts of the channels in $\vec{C}H_{E1}$ and $\vec{C}H_{E2}$, $\vec{C}H_{12}$, and $\vec{C}H_{21}$ and are treated as input I to their respective components' processes; the components execute; and the output signals from each component's processes are enqueued in the components' output channels.

5. 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) [13, 17]. 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 a relation between a place and a transition. A pair (p_i, t) denotes an arc from a source place p_i to a transition t , and a pair (t, p_o) denotes an arc from a transition t to a destination place p_o . In Petri Nets, each place (or each transition) can have one or more input transitions (or one or more input places), and one or more output transitions (or one or more output places). A weight is an integer attached to an 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 places in a Petri Net is called a marking, which represents a state of the net.

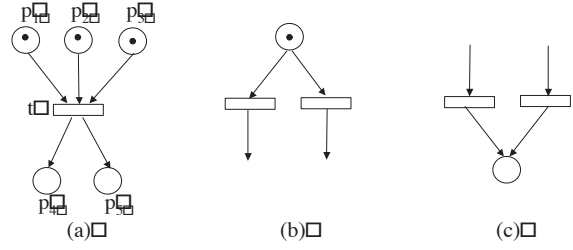


Figure 6. Example Petri Nets

At most one transition executes at a time in a Petri Net. A transition t is enabled if each of its input places p_i contains at least $w(p_i, t)$ tokens. 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 Figure 6 (a), depicting a transition with three input places p_1, p_2, p_3 ; two output places p_4, p_5 ; all of whose arcs have a weight of 1. Transition t is enabled because all three of its input places contain a token. After firing, a token is removed from each input place and a token is added to each output place.

We map a Petri Net to one HTS. Each place in the Petri Net is represented as a unique variable of the 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. Because a Petri Net has no control states or events, the HTS state and event elements are empty. Each Petri Net transition is represented as an HTS transition with the form $\langle cond, act \rangle$, where $cond$ is a predicate that tests that the transition's input places have the necessary number of tokens with respect to weights on the input arcs, and act adds zero or more tokens to the output places, according to the weights on the output arcs. For example, Petri Net transition t in Figure 6(a) would be represented as the following HTS transition:

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

where variables $v_1, v_2, v_3, v_4,$ and v_5 are the numbers of tokens in the places $p_1, p_2, p_3, p_4,$ and p_5 , respectively. In Figure 6(b), two transitions are in conflict: they share one input place, both are enabled, but only one of them can execute in a step. The transition that executes disables the other until its input place is populated again. In Figure 6(c), if the transitions' output place has an upper limit b on the number of tokens it can hold, then the firing of one transition may disable the other. We can capture this in an HTS by adding the extra enabling condition $v_o + w(t, p_o) \leq b$ to each transition that has an output with a bound.

Snapshot Element	Start of Macro-step	Micro-step τ
AV	AV	$AV' = assign(ss.AV, eval(ss, asn(\tau)))$

$en_states(ss, \tau)$	\equiv	true
$en_events(ss, \tau)$	\equiv	true
$en_cond(ss, \tau)$	\equiv	$ss.AV \models cond(\tau)$
$macro_semantics$	\equiv	simple, diligent
$pri(\Gamma)$	\equiv	Γ

Table 5. Template parameters for Petri Nets

Table 5 shows the relevant template parameters for Petri Nets. AV is the only snapshot element used, and the variable values are modified by the actions of executing transitions. Petri Nets use simple, diligent, macro-step semantics. There is no priority among the transitions ($pri(\Gamma) = \Gamma$). Petri Nets do not use any composition operators.

6 Statecharts Variants

In this section, we show how to use our template approach to compare notational variants, in particular statecharts variants. Statecharts, first introduced by Harel [3], are one of the most popular model-based specification notations. Many users have redefined subtle aspects of the statecharts semantics to better suit their 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 [19] 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; interrupt transitions can originate from within a component. Table 6 shows the template parameter values for five popular statecharts variants (Harel’s original semantics [3], Pnueli & Shalev [18], RSML [10], STATEMATE [4], and UML [16]). UML has simple, diligent macro-step semantics; the other statecharts variants have stable macro-step semantics.

The template parameters CS , CS_a , and en_states parameters capture the differences in which states can enable transitions. In Harel’s and in Pnueli & Shalev’s semantics, each non-concurrent HTS can execute only one transition per macro-step. We model this using CS_a , which stores the set of enabling states and is set to the empty set after a transition is taken. RSML and STATEMATE do not have this restriction, and it is possible for a macro-step to be an infinite loop of one or more HTSs’ transitions.

Template parameters AV , AV_a , and en_cond capture the differences in which variables can enable transitions. The current variable values (AV) are updated by executing tran-

sitions. In STATEMATE, a single transition may assign multiple values to the same variable, but only the last assignment has an effect. The abbreviation $assign_{last}$ used in Table 6 has the following meaning:

$$AV' = assign(ss.AV, lasts(asn(\tau)))$$

where $lasts$ returns the last assignment made to each variable. STATEMATE allows multiple transitions in the same micro-step to assign conflicting values to variables. The conflicts are non-deterministically resolved:

$$\begin{aligned} resolve(vv_1, vv_2, vv) &\equiv resolve_{STM} \equiv \\ vv &= \{(a, b) \mid (a, b) \in vv_1 \vee (a, b) \in vv_2\} \\ \wedge [\forall (a, b) \in vv. \forall (c, d) \in vv. (a = c \implies b = d)] \end{aligned}$$

Harel and Pnueli & Shalev allow external events to trigger transitions throughout a macro-step; parameter I_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. For notations that differentiate syntactically between internal events and external events (*e.g.*, RSML), we use function $intern_{ev}(E)$ to refer to the set of internal events and the function $extern_{ev}(E)$ for the set of external events.

Similarly, Harel and Pnueli & Shalev allow internal events generated in a micro-step to trigger any future transition in the same macro-step; parameter IE accumulates generated events. RSML and STATEMATE allow only events generated in the previous micro-step to trigger a transition. In UML, each object has an event queue that emits one event per (simple) macro-step. Because transitions may trigger on implicit internal events, such as the entering and exiting of states or changes in conditions or in variable values, parameters IE and O use a macro function ev_gen that returns all explicit and implicit events generated when transition τ executes in snapshot ss .

Many statecharts variants allow transitions to trigger on event expressions, such as negated events (*i.e.*, lack of an event) or disjunctions of events. We treat disjunctive events as a notational convenience for combining transitions that have similar actions. To handle negated events, we distinguish between trigger events $pos(\tau)$ and negated trigger events $neg(\tau)$. 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 generates event a ; they call this scenario a *global inconsistency*. To model Pnueli & Shalev’s semantics, we use IE_a to accumulate the

Parameter	Harel [3]	Pnueli [18]	RSML [10]	STATEMATE [4]	UML [16]
$reset_CS(ss, I)$	ss.CS	ss.CS	ss.CS	ss.CS	ss.CS
$next_CS(ss, \tau, CS')$			$CS' = entered(dest(\tau))$		
$reset_CS_a(ss, I)$	ss.CS	ss.CS	n/a	n/a	n/a
$next_CS_a(ss, \tau, CS'_a)$	$CS'_a = \emptyset$	$CS'_a = \emptyset$	n/a	n/a	n/a
$en_states(ss, \tau)$	$src(\tau) \subseteq ss.CS_a$	$src(\tau) \subseteq ss.CS_a$	$src(\tau) \subseteq ss.CS$	$src(\tau) \subseteq ss.CS$	$src(\tau) \subseteq ss.CS$
$reset_IE(ss, I)$	\emptyset	\emptyset	\emptyset	\emptyset	$append(ss.IE, I)$
$next_IE(ss, \tau, IE')$	$IE' = ss.IE \cup ev_gen(ss, \tau)$	$IE' = ss.IE \cup ev_gen(ss, \tau)$	$IE' = ev_gen(ss, \tau) \cap intern_{ev}(E)$	$IE' = ev_gen(ss, \tau)$	$IE' = append(tail(ss.IE), ev_gen(ss, \tau))$
$reset_IE_a(ss, I)$	n/a	\emptyset	n/a	n/a	n/a
$next_IE_a(ss, \tau, IE'_a)$	n/a	$IE'_a = ss.IE_a \cup neg(\tau)$	n/a	n/a	n/a
$reset_I_a(ss, I)$	I	I	I	I	n/a
$next_I_a(ss, \tau, I'_a)$	$I'_a = ss.I_a$	$I'_a = ss.I_a$	$I'_a = \emptyset$	$I'_a = \emptyset$	n/a
$en_events(ss, \tau)$	$pos(\tau) \subseteq ss.IE \cup ss.I_a \wedge (neg(\tau) \cap (ss.IE \cup ss.I_a)) = \emptyset$	$pos(\tau) \subseteq ss.IE \cup ss.I_a \wedge (neg(\tau) \cap (ss.IE \cup ss.I_a)) = \emptyset$	$trig(\tau) \subseteq ss.IE \cup ss.I_a$	$trig(\tau) \subseteq ss.IE \cup ss.I_a$	$trig(\tau) = head(IE)$
$reset_O(ss, I)$	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
$next_O(ss, \tau, O')$	$O' = ss.O \cup ev_gen(ss, \tau)$	$O' = ss.O \cup ev_gen(ss, \tau) \cap extern_{ev}(E)$	$O' = ss.O \cup (ev_gen(ss, \tau) \cap extern_{ev}(E))$	$O' = ev_gen(ss, \tau)$	$O' = ss.O \cup ev_gen(ss, \tau)$
$reset_AV(ss, I)$	ss.AV	ss.AV	ss.AV	ss.AV	ss.AV
$next_AV(ss, \tau, AV')$		$AV' = assign(ss.AV, eval(ss, asm(\tau)))$		$assign_{last}$	$AV' = assign(ss.AV, eval(ss, asm(\tau)))$
$reset_AV_a(ss, I)$	ss.AV	ss.AV	n/a	n/a	n/a
$next_AV_a(ss, \tau, AV'_a)$	$AV'_a = ss.AV_a$	$AV'_a = ss.AV_a$	n/a	n/a	n/a
$en_cond(ss, \tau)$	$ss.AV, ss.AV_a \models cond(\tau)$	$ss.AV, ss.AV_a \models cond(\tau)$	$ss.AV \models cond(\tau)$	$ss.AV \models cond(\tau)$	$ss.AV \models cond(\tau)$
$macro_semantics$	stable	stable	stable	stable	simple, diligent
$pri(\Gamma)$	no priority	no priority	no priority	$lowest \rightarrow ranked\ scope$	$highest \rightarrow ranked\ source$
$resolve(vv_1, vv_2, vv)$		n/a		$resolve_{STM}$	n/a

Table 6. Template parameters for statecharts variants ("n/a" means not applicable)

negated events that trigger transitions in the macro-step. Subsequent transitions are enabled only if their generated events are disjoint with this set IE_a (see Table 6'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.

With respect to outputs, in Harel's and in Pnueli & Shalev's semantics, all events generated during the micro-step are communicated as outputs. In RSML, all generated external events are communicated as outputs. In STATEMATE, only the events generated in the last micro-step of the macro-step are communicated as outputs.

With respect to priorities on transitions, Harel's, Pnueli & Shalev's, and RSML's semantics place no priority scheme on transitions. STATEMATE gives priority to transitions whose *scope* has the lowest *rank*, where *scope* and *rank* are defined in Section 2. UML favours transitions with the highest-ranked source state.

Some of the statecharts features not modelled here (*i.e.*, compound transitions) are simply notational conveniences that can be accommodated by using the features' expanded definitions. Some features (*e.g.*, history states) can be modelled with changes to the template parameters (*e.g.*, calculation of current states) without requiring changes to the parameterized template definitions. Other features (*e.g.*, real-time clocks) are best modelled as an extension to our HTS model. Generality of the HTS model is described in [15]

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. 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; but the template semantics representation would make it easier to compare SCR with other dataflow languages.

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.

8 Acknowledgments

We thank Dan Berry, Yun Lu, Andrew Malton, and John Thistle for helpful discussions of early aspects of this work.

We also thank Constance Heitmeyer and Ralph Jeffords for their help with the semantics of SCR and for their feedback on earlier drafts of this paper.

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.
- [2] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, 8:231–274, 1987.
- [3] D. Harel et al. On the formal semantics of statecharts. In *Symp. on Logic in Comp. Sci.*, pages 54–64, 1987.
- [4] D. Harel and A. Naamad. The Statemate semantics of statecharts. *ACM Trans. on Soft. Eng. Meth.*, 5(4):293–333, 1996.
- [5] C. L. Heitmeyer and R. D. Jeffords. The SCR tabular notation: A formal foundation, 2003. NLR/MR/5546-03-8678.
- [6] 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.
- [7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, UK, 1985.
- [8] ISO8807. LOTOS - a formal description technique based on the temporal ordering of observational behaviour. Technical report, ISO, 1988.
- [9] ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union - Standardization Sector, 1999.
- [10] 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.
- [11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
- [12] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.
- [13] T. Murata. Petri Nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989.
- [14] J. Niu, J. M. Atlee, and N. A. Day. Composable semantics for model-based notations. In *Proc. of the ACM SIGSOFT FSE-10*, pages 149–158, 2002.
- [15] J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. Technical Report CS-2003-19, University of Waterloo, School of Computer Science, 2003. Submitted for publication.
- [16] Object Management Group. Unified Modelling Language (UML), v1.4, 2001. Internet: www.omg.org.
- [17] J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.
- [18] 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.
- [19] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.