```
@inproceedings{Lu:2004:MTS:1025115.1025237,
 author = {Lu, Yun and Atlee, Joanne M. and Day, Nancy A. and Niu, Jianwei},
 title = {Mapping Template Semantics to SMV},
 booktitle = {Proceedings of the 19th IEEE International Conference on Automated Software Engineering},
 series = {ASE '04},
 year = {2004},
 pages = {320--325}
}
```

DOI: https://doi.org/10.1109/ASE.2004.1342756

Mapping Template Semantics to SMV

Yun Lu Joanne M. Atlee Nancy A. Day Jianwei Niu
School of Computer Science
University of Waterloo
E-mail: {y4lu, jmatlee, nday, jniu}@uwaterloo.ca

Abstract

We show how to create a semantics-based, parameterized translator from model-based notations to SMV, using template semantics. Our translator takes as input a specification and a set of user-provided parameters that encode the specification's semantics; it produces an SMV model suitable for model checking. Using such a translator, we can model check a specification that has customized semantics. Our work also shows how to represent complex composition operators, such as rendezvous, in the SMV language, in which there is no matching language construct.

1. Introduction

Template semantics [17] is a template-based approach to expressing the semantics of model-based notations, such as statecharts variants and process algebras. The semantics that are common among notations (e.g., the concept of an enabled transition) are pre-defined as a template of parameterized definitions. Users instantiate the template into a complete semantics by providing notation-specific parameter values (e.g., predicates on how states, events, and variables enable transitions). Composition operators are parameterized constraints on how components execute and share information. The result is a definition for a notation's semantics that is finely decomposed into separate parameterized concerns, making it much easier to read, write, and compare notations' semantics.

In this paper, we describe how to use the semantic decomposition provided by template semantics to facilitate notation-specific analysis. In particular, we use template semantics to parameterize the translation from a requirements notation to the input language of the SMV family of model checkers: Cadence SMV [15], NuSMV [5]. Our translator takes as input a specification in a notation and a set of template parameters detailing the notation's semantics; the translator combines these inputs with the template's common-semantics definitions, to generate an SMV model of the specification. The generated SMV model preserves the modularity of both the original specification and template semantics. The state space of the SMV model is comparable to that of the original specification, and no extra steps are introduced. Our translator is fully automated and supports multiple options for each template parameter, including the parameter values used in the definitions of many popular requirements notations: CSP [10], CCS [16], basic LOTOS [11], a subset of SDL88 [12], and several statecharts [8] variants. The translator supports also a rich set of composition operators, including rendezvous, environmental synchronization, sequence, choice, and interrupt (a form of goto). Because the supported parameter values describe a variety of ways in which states, events, variables, and priorities affect a notation's semantics, the translator can be used for many more notations (defined by different combinations of parameter values and composition operators) than the notations listed above. We chose to translate to SMV because it is a well-used and general-purpose model checker. Because of the simplicity of the SMV language, our method for representing various forms of composition may be applicable to other model checkers.

Compared to writing a direct translator from one notation to another, or to an intermediate language (e.g., SAL [1], IF [2], Action Language [3]), we can generate new notation-specific translations automatically by simply selecting different combinations of template-parameter values and composition operators. Furthermore, template semantics supports a richer set of composition operators than other intermediate languages. Our work is similar to approaches that generate a model or analysis tool from a notation's semantics (e.g., [6] [18] [7]), except that our use of template semantics allows one to specify semantics by providing parameters to predefined templates, rather than providing a complete semantic description.

2. Template Semantics

The basic computation model of template semantics is a nonconcurrent, hierarchical transition system (HTS). An HTS is an extended state machine (adapted from basic transition systems [14] and statecharts [8]) that consists of transitions and a hierarchical set of states. A transition label can include event and condition triggers, assignments to typed variables, generated events, and a priority. A specification is a hierarchical composition of HTSs; concurrency is introduced via composition operators. The mapping from a notation's syntactic constructs to our HTS syntactic constructs is usually straightforward.

We use **snapshots** to collect information about the system at observable points in its execution. A snapshot is a tuple of eight elements, $\langle CS, IE, AV, O, CS_a, IE_a, AV_a, I_a \rangle$, representing the system's current states CS, current internal events IE, current variable values AV, and current outputs O. CS_a, AV_a, IE_a, I_a are auxiliary elements that accumulate data about states, variables values, and internal and external events, respectively. Each notation collects different information in these auxiliary elements.

Template semantics is a collection of parameterized definitions that, taken together, describe how a snapshot can change in an execution step. These definitions separate the semantics of model-based notations into four phases: *resetting* a snapshot with inputs, determining the set of *enabled* transitions, choosing a set of transitions to *execute* in a step, and *applying* to the snapshot the effects of the executing transitions. The parameterized definitions include:

- ***micro_step***: a step between consecutive snapshots, due to the execution of at most one transition per HTS
- ***macro_step***: a sequence of zero or more micro-steps in response to external input (Macro-steps are used by notations, such as statecharts, that fully respond to one set of inputs before receiving another.)
- ***reset***: resets the current snapshot with new inputs at the beginning of a macro-step
- ***enabled***: computes the set of transitions enabled by the current snapshot's states, events, and variable values
- ***execute***: chooses nondeterministically, from the highest-priority enabled transitions, which transitions to execute
- ***apply***: applies the executing transitions' actions (i.e., generated events and variable assignments) to the current snapshot, to derive the next snapshot

These common semantic definitions are instantiated by 22 template parameters that use, reset, and update the snapshot elements in different ways for different notations. Table 1 shows template instantiations for STATEMATE statecharts [9] and CCS [16] with variables.[1] Column ***resetXX(ss,I)*** lists the parameters used in template definition ***reset***: each parameter resets a snapshot element in snapshot ss (e.g., ***resetCS*** resets snapshot element CS), removing old data and incorporating new system inputs I. Column ***nextXX(ss,τ)*** lists the parameters used in ***apply***: each parameter updates a snapshot element with respect to transition τ's actions. Consider STATEMATE event semantics: at the start of a macro-step, only input events can trigger transitions; and in subsequent micro-steps, only events generated in the previous micro-step can trigger transitions. These semantics are reflected in the values of five event-related parameters (rows 3-5 in Table 1): ***reset*** empties snapshot element IE of old internal events and sets element I_a to the input's events, $I.ev$; when a transition executes, IE is set to the transition's set of generated events, and I_a is emptied; transitions are enabled by any event in IE or I_a.

Parameter ***macro-semantics*** determines the semantics of a macro-step. In **simple** semantics, every macro-step is either a micro-step or an idle step, and the snapshot is reset at the start of every step. Simple semantics can be either **diligent**, in which enabled transitions have priority over an idle step, or **nondiligent**. In **stable** semantics, a macro-step is a maximal sequence of micro-steps, starting with a reset snapshot and ending with a **stable** snapshot, in which no transition is enabled.

Parameter ***pri*** determines the priority scheme among enabled transitions. Parameter ***resolve_conflicts*** sets the notation's policy for resolving conflicting assignments made to the same variable in the same micro-step.

We use composition operators to compose hierarchically a collection of concurrent HTSs. The composition operators control when the HTSs execute and how the HTSs share data (e.g., generated events). We have defined the template semantics for eight composition operators: two kinds of parallel, interleaving, environmental synchronization, rendezvous, sequence, choice, and interrupt. Interrupt composition transfers control between components (HTSs or composed HTSs) via new composition-level transitions; interrupt transitions' source and destination states can be either a component or states within a component.

3. SMV Language

In the SMV language, models are described using variables, and equations that assign initial and next values to variables in every SMV step. SMV also supports the specification of invariants, as boolean expressions following the keyword `INVAR`. Expression operators `!`, `&`, `|`, `->`, and `<->`, represent "not," "and," "or," "implies," and "iff" respectively. Comments follow the symbol "`--`".

We make extensive use of SMV's macros declared after the keyword `DEFINE`. Macros are replaced by their definitions, so they do not increase the system's state space.

SMV provides modules to decompose a model, so that the statements can be reused by creating a module instance, which is declared as a variable. A module can also be used to structure variables into a record that can be passed as a parameter to another module. If `a` identifies an instance of a module, then the expression `a.b` identifies the internal variable or macro named `b` within module instance `a`. We use the terms "module instantiation" and "record" interchange-

[1] Table 1's *CCS with variables* is CCS with shared global variables; this is different from data-passing CCS [16], which allows internal events to carry data parameters.

Parameter		STATEMATE		CCS with variables	
		$resetXX(ss, I)$	$nextXX(ss, \tau)$	$resetXX(ss, I)$	$nextXX(ss, \tau)$
states	$CS =$	$ss.CS$	$entered(dest(\tau))$	$ss.CS$	$dest(\tau)$
	en_states	$src(\tau) \subseteq ss.CS$		$src(\tau) \subseteq ss.CS$	
events	$IE =$	$\emptyset$	$gen(\tau)$	n/a	
	$I_a =$	$I.ev$	$\emptyset$	$I.ev$	$ss.I_a \cup gen(\tau)$
	en_events	$trig(\tau) \subseteq ss.I_a \cup ss.IE$		$trig(\tau) \subseteq ss.I_a$	
	$O =$	$\emptyset$	$gen(\tau)$	$\emptyset$	$gen(\tau)$
variables	$AV =$	$assign(ss.AV, I.var)$	$assign(ss.AV, eval(ss.AV, last(asn(\tau))))$	$assign(ss.AV, I.var)$	$assign(ss.AV, eval(ss.AV, asn(\tau)))$
	en_cond	$ss.AV \models cond(\tau)$		$ss.AV \models cond(\tau)$	
macro-semantics		stable		simple diligent	
pri		lowest-ranked scope		none	
resolve_conflicts		$resolve(vv_1, vv_2, vv)$		n/a	

$entered(s)$: set of states when state s is entered, including s's ancestors and relevant descendants' default states
$assign(X, Y)$: updates assignments X with assignments Y
$eval(X, A)$: evaluates expressions in A with respect to assignments X, and returns variable-value assignments for A
$last(A)$: a sub-sequence of A, comprising only the last assignment to each variable in the sequence of assignments A
$resolve(vv_1, vv_2, vv)$: the conflicting variable assignments in vv_1 and in vv_2 are resolved to vv non-deterministically
$gen(\tau)$, $dest(\tau)$, $src(\tau)$, $trig(\tau)$, $cond(\tau)$, and $asn(\tau)$ are accessor functions on transition τ

Table 1. Template Parameter Definitions for STATEMATE and CCS with variables

ably. Modules can be hierarchical. All statements in all modules run *synchronously* in an SMV step.

4. Translation to SMV

The first key idea of our translation method is to decompose the translation primarily by template-semantics structure and secondarily by specification structure. This approach differs significantly from most translations, in that a translation is not primarily a mapping of specification construct to corresponding SMV construct. Rather, the high-level module structure of the resulting SMV model reflects template semantics' parameterized definitions and parameters; these modules are then structured along the lines of the specification's composition hierarchy. This decomposition structure localizes definitions that are most likely to change (i.e., the specification and template-parameter values) and allows our translator to optimize for SMV the definitions that are least likely to change (i.e., the common semantics and composition operators).

The second key idea is to use characteristic predicates to represent sets, which are prevalent in template-semantics definitions. For example for the set of executing transitions, we introduce for each transition a boolean macro that is true whenever the transition is executed in the current step.

The third key idea is to make extensive use of macros to structure and communicate information about the system, without adding to the state space of the model. We use macros to represent the sets of enabled and executing transitions and components, and use a number of parameter- and composition-specific helper macros to calculate these sets. With these macros, we can represent a rich set of composition operators using SMV's synchronous composition.

The fourth key idea is to use SMV's invariants to constrain when components may execute, as prescribed by the meanings of the various composition operators. Each component has a boolean macro to indicate whether or not it executes. For example, the invariants for interleaving composition constrain these macros so that the components' executions are interleaved. By using constraints rather than specifying which components execute, our translation preserves any nondeterminism in the specification.

Stable and simple macro-semantics differ only in when the system senses inputs from the environment. The fifth and final key idea is to represent macro-steps, for notations that use stable macro-semantics, as micro-steps that have conditional reset statements. Each SMV step is a micro-step. If the system is stable (i.e., no transition is enabled) at the start of a micro-step, new inputs are admitted to the system using the reset function. This method is adapted from the work of Chan et al. [4].

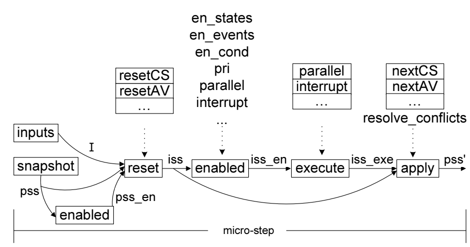

Figure 1. Architecture of SMV Model

Figure 1 shows the module structure of the SMV model that our translator produces for a notation with stable macro-semantics. Boxes represent SMV modules. A solid arrow represents a record, of SMV variables or macros, that is instantiated in the arrow's source module and is passed as a parameter to the destination module. The dashed arrows show sub-modules and encodings of template-parameter values (e.g., template parameter ***pri*** is encoded in the definition of module `enabled`). Entities in the figure that share names with template parameters or with composition operators correspond to those elements. Separate modules capture the template-semantics phases: resetting a snapshot with inputs (`reset`), determining the set of enabled transitions (`enabled`), choosing a set of transitions to execute in a step (`execute`), and applying the effects of the executing transitions on the snapshot (`apply`). Individual parameter values are encapsulated in either modules (e.g., `resetCS`) or macros (e.g., `en_states`). The structure of the specification is realized in the decomposition of the execute module, one submodule for each composition operator and each HTS. In addition (and not shown), there is an SMV module that contains a set of `init` statements, initializing the snapshot elements at the start of execution.

Our translator creates the SMV model by walking over the abstract syntax tree (AST) of the specification, producing output suitable for each parameter value. Next, we briefly describe each of the modules in the figure, discussing how the translation changes for different parameter values. Details on our translation method can be found in Lu [13].

4.1 Snapshot and Inputs

The `snapshot` module has a submodule for each snapshot element used by the specification's semantics. These submodules contain SMV variable declarations for the states, variables, and events of the specification. The state submodules (CS, CS_a) are further decomposed into one module for each HTS. Each of these modules declares an enumerated variable to represent the HTS's current basic state (the enumerated type has one value for each basic state, plus a value `noState` to indicate that the HTS is not active). Information about super-states (e.g., whether a super-state is current) is represented in macros and derived from information about basic states. The representation of CS_a depends on the type of information stored. If CS_a is used to record all previous states visited in the current macro-step, to avoid an infinite macro-step, then a boolean variable is needed for every basic state and super-state, and the set is represented as the characteristic predicate over these variables. Each of the variable submodules (AV, AV_a) declares an SMV variable for each specification variable. We assume all variables are of the primitive types provided by SMV: booleans, enumerated types, and integer ranges. Each of the event submodules (IE, O, IE_a, I_a) declares a boolean variable for each event and represents event sets by their characteristic predicates. The `inputs` module has variable declarations for each input, in a similar manner to the snapshot module.

4.2 Enabled

For stable macro-semantics, the `enabled` module is instantiated twice (see Figure 1): first, to determine if the current snapshot is stable and needs to be reset; and second, to determine which entities are enabled after a reset.

Figure 2 shows a simple system with two HTSs (P and Q) and one composition operator. Figure 3 shows the top-level `enabled` module for this example, where P and Q are composed using parallel composition. This module instantiates, in its VAR section, an `enabled` submodule for each

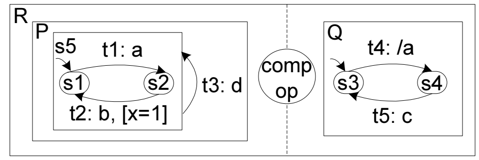

Figure 2. Example Specification

of the specification's HTSs, passing its snapshot argument `ss` to the submodules. The DEFINE section sets a macro called `any` for each component resulting from a composition operator; this macro determines whether the component is enabled, based on the enabled status of its subcomponents. Each HTS submodule sets its own `any` macro (see below). For parallel composition, a component is enabled if any of its subcomponents are enabled.

```
MODULE enabled(ss)
 VAR
  P:  enabled_P(ss);
  Q:  enabled_Q(ss);
 DEFINE
  R.any := P.any | Q.any;
```

Figure 3. Enabled Module

Figure 4 shows part of the `enabled` submodule for the HTS P. Each transition has a macro to indicate whether the transition is enabled (e.g., `ent2`). This macro is the conjunction of helper macros that test whether the transition's source state, triggering events, and guard conditions are enabled, according to the template parameters for enabling states (***en_states***), enabling events (***en_events***), and enabling conditions (***en_cond***), respectively. Our translator produces these helper macros, based on the values of the provided template parameters. References to state names, variables, and events are prefixed with module names that reflect the modular structure of the snapshot (e.g., `ss.AV.x` refers to variable x in element AV of snapshot ss). For each transition, the submodule declares a second macro (e.g., `t2`) that determines whether the transition is *priority-enabled*: the transition is enabled and no higher-priority transition is enabled. Our translator produces these macros, based on the static priority scheme specified in template parameter ***pri***. In this example, the priority scheme gives `t3` priority over `t2`. Finally, the HTS is enabled (macro `any`) if one or more of its transitions are priority-enabled.

```
MODULE enabled_P(ss)
 DEFINE
  enStates_t2 := ss.CS.in_s2 ;
  enEvents_t2 := ss.IE.b ;
  enCond_t2   := ss.AV.x = 1;
  ent2 := enStates_t2 & enEvents_t2 & enCond_t2;
  t2 = ent2 & !ent3 ;
  any := t1 | t2 | t3 ;
```

Figure 4. Enabled Sub-module for HTS P

For compositions such as rendezvous and environmental synchronization, the enabling of a component depends on the occurrence of synchronization events. For the example of Figure 2, if 'rendezvous on event a' is the composition operator, then component R is enabled if Q can generate a (transition $t4$ is enabled) in the same step that P can trigger on a (transition $t1$ is enabled), or if either component is enabled by a nonsynchronization event. As appropriate, our translator adds macros that help determine the enabled status of synchronized components.

4.3 Reset

The `reset` module contains only macros: one for each snapshot element. Depending on whether the system is stable (established by a macro in the `enabled` module), inputs are incorporated into the snapshot. There are submodules that set each snapshot element, according to the values of the provided ***resetXX*** template parameters.

4.4 Execute

Figure 5 shows the `execute` module for the example from Figure 2. The `execute` module uses the enabled status of transitions and components (parameter `en` is an instantiation of the `enabled` module) to constrain which transitions may execute in a step. The module instantiates an operator-specific submodule (e.g., `parallel`) for each composition and an HTS-specific submodule (e.g., `execute_P`) for each HTS. It also declares and sets for each composed component an `.any` macro that indicates whether the component executes in this step. Diligence, when relevant, is enforced in the `execute` module by an invariant: if the top-level component is enabled, it must execute.

```
MODULE execute(en)
 VAR
   P : execute_P(en.P);
   Q : execute_Q(en.Q);
   R : parallel(en.P,en.Q,P,Q)
 INVAR  en.R.any -> R.any
```

Figure 5. Execute Module for R

Each of the HTS submodules declares a variable whose value identifies which of the enabled transitions, if any, is chosen (possibly nondeterministically) to execute in the current step. An invariant asserts that a transition can execute only if it is enabled (as indicated by `enabled` macros).

The submodule for a composition operator uses the enabled and execute status of the subcomponents and the enabled status of the composition to decide whether the composition executes and possibly to constrain further whether the subcomponents execute; the submodule also ensures that subcomponents execute if the operator is diligent. Figure 6 shows the submodule for parallel composition: a parallel component executes (macro `any`) if either subcomponent executes; and an enabled subcomponent must execute.

```
MODULE parallel(enLeft,enRight,exeLeft,exeRight,enMe)
 DEFINE
  any := exeLeft.any | exeRight.any;
 INVAR
  (any ->  ((enLeft.any -> exeLeft.any)
            & (enRight.any -> exeRight.any)))
```

Figure 6. Parallel Composition

Figure 7 shows an example of a rendezvous on event a. Extra `execute` macros determine whether the subcomponents are executing transitions that trigger on a or generate a. The invariant ensures that each subcompont executes a transition triggered by a if and only if the other subcompnent executes a transition that generates a. When a rendezvous occurs, only one transition in each component executes. Finally, if a rendezvous does not occur, then the behaviour of the components is interleaved. Details on the remaining composition operators can be found in Lu [13].

```
MODULE rend_sync_a(enLeft,enRight,exeLeft,exeRight,enMe)
 DEFINE
  any := exeLeft.any | exeRight.any;
  -- rendezvous means one generates and other triggers
  a_rend := (exeLeft.a_trig & exeRight.a_gen)
          | (exeLeft.a_gen & exeRight.a_trig);
 INVAR
   -- left and right are trig/gen on same sync event
    (exeLeft.a_trig <-> exeRight.a_gen)
  & (exeLeft.a_gen  <-> exeRight.a_trig)
   -- if rendezvous, one tran executes in each component
  & ((a_rend) ->
      !(exeLeft.more_than_one | exeRight.more_than_one))
   -- interleaved behaviour
  & (!(a_rend) -> !(exeLeft.any & exeRight.any))
```

Figure 7. Rendezvous Composition on $\{a\}$

4.5 Apply

Module `apply` sets the values of the snapshot elements in the next snapshot, based on the effects of the executing transitions chosen in the `execute` module. It updates each snapshot element *XX* in a separate submodule `nextXX`, which realizes the semantics of template-parameter value ***nextXX***. Submodule `nextAV` uses the template parameter ***resolve_conflicts*** to handle simultaneous assignments to the same variable made by multiple HTSs. Underflow or overflow errors are detected for integer-range variables.

5 Evaluation

To validate our translator, we have tested our translation on every template-parameter value and every composition operator. We assume that the template parameters and composition operators are all separate concerns. This separation of concerns eases the evoluation of the translator in that adding another parameter value or composition operator does not usually affect the behaviour of the others.

Table 2 shows how the size of the SMV model resulting from our translation compares with the original specification by snapshot element. The basic states, internal and external events, and variables of the specification have corresponding SMV elements in CS, IE, O, and AV respectively. When used, the auxiliary snapshot elements, CS_a, IE_a, I_a, and AV_a, contribute to the state space as appropriate for their parameter values. For example, CS_a may be used to record all previous states visited in a micro-step to avoid an infinite macro-step. In this case, the characteristic predicate representation of CS_a must have a boolean variable for every basic state and super-state. There is one variable per HTS to represent which transition is chosen to execute. No variables are added for the composition operators, except for one for each choice composition to record the choice made between components.

Snapshot Element	SMV Variables	
	Worst Case	STATEMATE, CCS with variables
CS	1 with $(b+1)$ values	1 with $(b+1)$ values
CS_a	$b+s$ boolean	n/a
IE	i boolean	i boolean
IE_a	i boolean	n/a
I_a	e boolean	e boolean
O	i boolean	i boolean
AV	v typed	v typed
AV_a	v typed	n/a
transitions (per HTS)	1 with $(t+1)$ values	1 with $(t+1)$ values

Table 2. SMV Model Size for t transitions (per HTS), b basic states, s super states, i internal events, e external events, and v variables

We have performed two case studies: a heating system and a single lane bridge. We simulated and model checked our SMV models with a set of properties to check that the models match the behaviours of the original specifications. NuSMV calculated the reachable state space sizes of these models as: 6.88e+8 (heating system), and 9.216e+5 (single lane bridge). Because we use the same variable, event, and state names (prefixed by snapshot information), and do not add any extra steps, the counterexamples can be easily understood by users. The complete specifications, the SMV models, and the properties checked can be found in [13].

6 Conclusion

We have created a fully automated, semantics-based, parameterized translator from model-based notations to SMV. The translator takes the template-semantics description of a notation and a specification in the notation, and it produces an SMV model whose state space and execution steps are comparable to the original specification. Using our translator, we can model check specifications written in a wide range of model-based notations. The translator handles all of the template-parameter values and the composition operators that were used in [17] to describe the semantics of basic transition systems [14], CSP [10], CCS [16], basic LOTOS [11], and several statecharts [8] variants. Because the modularity of our translation matches the modularity of template semantics, adding a new parameter value or composition operator normally means only creating new SMV module(s). A secondary, but key, contribution of the work is showing how to represent a rich collection of composition operators within the language features provided by SMV.

References

[1] S. Bensalem et al. An overview of SAL. In *Langley Formal Methods Workshop*, pages 187–196. Center for Aerospace Information, NASA, 2000.

[2] M. Bozga, S. Graf, L. Mounier, and J. Sifakis. The intermediate representation IF: syntax and semantics. Technical report, Verimag, Grenoble, 1999.

[3] T. Bultan. Action language: A specification language for model checking reactive systems. In *Int. Conf. on Soft. Eng.*, pages 335–344, 2000.

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

[5] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *Int. Journal on Soft. Tools for Technology Transfer*, 2(4):410–425, 2000.

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

[7] L. Dillon and R. Stirewalt. Inference graphs: a computational structure supporting generation of customizable and correct analysis components. *IEEE Trans. on Soft. Eng.*, 29(2):133–150, Feb 2003.

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

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

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

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

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

[13] Y. Lu. Mapping template semantics to SMV. Master's thesis, School of Computer Science, University of Waterloo, 2004. In preparation.

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

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

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

[17] J. Niu, J. M. Atlee, and N. A. Day. Template semantics for model-based notations. *IEEE Trans. on Soft. Eng.*, 20(10):866–882, Oct. 2003.

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