Springer Copyright Notice

Published in: ***Software and Systems Modeling,*** May 2012

"Code Generation for a Family of Executable Modelling Notations"

Cite as:

Prout, A., Atlee, J.M., Day, N.A., Shaker, P. "Code Generation for a Family of Executable Modelling Notations, in *Software and Systems Modelling* (2012) 11: 251.-272.

BibTex:

@article{ProutSoSym12,
  TITLE = {{Code Generation for a Family of Executable Modelling Notations}},
  AUTHOR = {Prout, A. and Atlee, J.M. and Day, N.A. and Shaker, P.},
  JOURNAL = {{Software and Systems Modelling}},
  PAGES = {251-272},
  YEAR = {2012}
}

DOI: https://doi.org/10.1007/s10270-010-0176-6

Code Generation for Semantically Configurable Modelling Notations

Adam Prout, Joanne M. Atlee, Nancy A. Day, Pourya Shaker

David R. Cheriton School of Computer Science
University of Waterloo, Waterloo Ontario, N2L 3G1, CANADA
e-mail: {aprout,jmatlee,nday,p2shaker}@uwaterloo.ca

The date of receipt and acceptance will be inserted by the editor

Abstract We are investigating *semantically configurable* model-driven engineering, in which specifiers are able to configure the semantics of their models. The goal of this work is a modelling environment that supports flexible, configurable modelling notations, so that specifiers are better able to represent their ideas, and yet still provides the types of analysis tools and code generators normally associated with model-driven engineering.

In this paper, we describe *semantically configurable code generation*, comprising (1) a modelling notation that is semantically configurable via a set of parameter values specifying semantics choices and (2) a code-generator generator that creates a suitable Java-code generator for each derivable modelling notation. Our prototype code-generator generator supports 22 semantics parameters, 57 parameter values, and 7 composition operators. As a result, we are able to produce code generators for a sizable family of modelling notations, though at present the performance of our generated code is about an order of magnitude slower than that produced by commercial-grade generators.

1 Introduction

One of the obstacles to successful model-driven engineering (MDE) is the semantic gap between the modelling notation and the system being modelled. It is because of this gap that there exist multiple variants of modelling notations, such as variants of statecharts and process algebras, and variation points in the Unified Modelling Language. Notations are often customized or tweaked during use, to ease the modelling in a particular domain or of a particular problem.

To help bridge this gap, we propose *semantically configurable* model-driven engineering as a way of accommodating semantic variability among

related modelling notations. Semantically configurable MDE enables specifiers to create or customize their own modelling notations, and yet still have access to the modelling tools – that is, the editing environments, model analyzers, code generators, etc. – that are normally associated with MDE. Note that our work is distinct from complementary efforts on configurable tools [13,15] that support configurability with respect to the abstract syntax of a family of notations [32] but do not support configurable semantics.

In previous work, we introduced *template semantics* [28,29], by which we configure the semantics of a modelling notation. Template semantics structures the operational semantics of a family of modelling notations as a *set of predefined templates* that are instantiated with *user-provided parameter values*. Thus, a member of this family can be succinctly described as a set of template parameter values. The intended scope of template semantics is the family of notations whose semantics can be expressed operationally in terms of execution traces. This family includes process-algebras (e.g., CCS, CSP, basic LOTOS), statecharts variants (e.g., statecharts, STATEMATE, RSML, UML StateMachines), and even more sophisticated notations like SCR [19], SDL88 [21], and BoxTalk [41]. Because a notation's semantics can be expressed as a collection of parameters, it can be easily parsed, and we are starting to develop semantically parameterized tools [26]. For example, Metro [28] is a suite of semantically parameterized analysis tools, and Express [26] is a semantically parameterized translator to the Symbolic Model Verifier (SMV) [27].

As anecdotal evidence of the utility of semantically configurable modelling, we relate our experiences with graduate-student modellers. Students taking a graduate course on computer-aided verification used our configurable modelling notation to specify the behaviour of elevators, an office lighting system, and a hotel-room safe. They used Express to transform their models into representative SMV models and then verified their models using the SMV model checker. The students were free to choose their own modelling semantics, and only one student used a semantics that conforms to the classic semantics of some existing notation. Some specifications had a statecharts-like semantics, but used rendezvous for a critical synchronization. Other specifications had a CCS-like semantics, but used global shared variables. These observations suggest that (student) modellers are comfortable manipulating the semantics of a modelling notation as a strategy to producing a correct model – as opposed to manipulating only the model and correctness properties until the analysis is successful.

In this paper, we provide an extended description of our prototype of a semantically configurable code-generator generator (CGG), which was first presented in [36]. The CGG takes as input a description of a modelling notation's semantics, expressed as a set of template-semantics parameter values, and produces a code generator for that notation. Problems that we addressed in the course of this work include

- **Configurable CGG:** A primary contribution of this work is the code structure of the CGG, which isolates code that pertains to individual

template-semantics parameters and parameter values. This structure eases the task of adding new parameter values that support new semantic choices.
- **Configurable execution semantics:** A second contribution is the run-time architecture of the generated Java programs. The generated programs have a common abstract execution step that is specialized by composition operators and whose details are parameterized by template-semantics parameters (similar to the execution step of SMV models generated by Express [26]).
- **Representing composition operators in Java:** A side effect of the above contribution is Java implementations of a variety of composition operators. The Java scheduler imposes an interleaving semantics on concurrent threads, whereas many modelling notations have composition operators that are more tightly synchronized, such as parallel composition and rendezvous.
- **Resolving nondeterminism:** There are several natural sources of nondeterminism in models (e.g., selecting one of many enabled transitions to execute). While it may be appropriate to leave such nondeterminism unresolved during the modelling phase, nondeterminism in source code is unnatural. The transformation of a model into source code may involve decisions to eliminate nondeterminism.
- **Performance evaluation:** A long-term concern of this work is how efficient CGG-generated code is, given the competing concern that a configurable CGG be general enough to support a family of modelling notations. As a baseline, we compare the performance of our generated code against the performance of code generated by three commercial tools: IBM Rational Rose RT [20], IBM Rational Rhapsody [39], and SmartState [1] – each of which has been optimized for a particular modelling notation.

The rest of this paper is organized as follows. In Section 2, we review template semantics, which we use to configure the semantics of modelling notations. This section includes summaries of the semantics parameters and composition operators that are supported in our approach. In Sections 3 and 4, we describe our code-generator generator and the architecture of the generated code, respectively. We discuss techniques for resolving nondeterminism in Section 5. We present the results of our performance evaluations in Section 6, and we conclude the paper with discussions on related work, limitations, and future work.

2 Semantically Configurable Notations

In this section, we describe the syntax and semantics of our modelling notation. The semantics can be configured via a set of user-specified parameter values, which we will also review. We achieve configurability using template semantics [29,30]. Understanding how we support semantically configurable

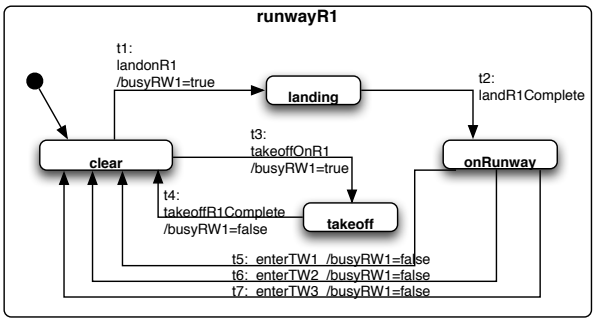
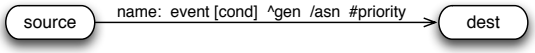


Fig. 1 Example HTS

modelling notations helps in understanding the degree of configurability supported by our CGG and in understanding the user's task of configuring the CGG.

2.1 Syntax

To accommodate a variety of modelling notations, we base the syntax of our notation on a form of extended finite-state machine that we call *hierarchical transition systems (HTS)*, which are adapted from statecharts [18]. An example HTS is shown in Figure 1. It includes control states and state hierarchy, state transitions, events, and typed variables, but not concurrency. Concurrency is achieved by composing multiple HTSs. Transitions have the following form:



Each transition has a source state, is triggered by zero or more events, may have a guard condition (a predicate on variable values), and may have an explicit priority. If a transition executes, it leads to a destination state, may generate events, and may assign new values to variables. An HTS designates an initial state for each hierarchical state and initial variable values (not shown). In UML terminology, an HTS corresponds to a simple-composite state with only one region.

2.2 Semantic Domain

The semantics of an HTS is the set of its possible execution sequences. The semantic domain of HTS execution is *sequences of snapshots*, where a *snapshot* records information about the model's execution at a discrete point in the execution. Snapshot information is stored in distinct elements:

CS	-	the current states
IE	-	the events to be processed
AV	-	the current variable-value assignments
I_a	-	data about inputs to the HTS
O	-	the generated events (to be communicated to other HTSs)

In addition, the snapshot includes auxiliary elements that, for different notations, store different types of information about the HTS's execution:

CS_a	-	data about states, such as enabling or history states
IE_a	-	data about events, such as enabling or nonenabling events
AV_a	-	data about variable values, such as old values

An execution starts with an initial snapshot of initial states and variable values. Consecutive snapshots ss_i and ss_{i+1} represent a "step" in the HTS's execution. There are two levels of granularity for steps: a *micro-step* represents the execution a single transition, and a *macro-step* is a sequence of zero or more micro-steps taken between consecutive inputs I from the environment.

2.3 Parameterized Semantics

Many modelling notations have comparable language constructs (e.g., states, events, typed variables, transitions) and semantic domains (sequences of snapshots), but they vary in how a model's execution affects snapshot values and vice versa. For example, many notations have a notion of *enabled transition*, representing the set of possible "next steps" that a model in a particular execution state is ready to perform, but use different information to decide which of a model's transitions are enabled.

The semantic mapping from our modelling syntax to the semantic domain is defined in terms of functions and relations over snapshot elements. In previous work, we developed a formalism called *template semantics* [29] in which semantic-mapping definitions are *parameterized*, resulting in a definition for a *family* of modelling notations. The parameters effectively represent semantic variation points. We provide two template definitions below, as examples:

$$\textsc{enabled\_trans}(ss, T) \equiv$$
$$\{\tau \in T \mid \mathbf{en\_states}(ss, \tau) \wedge \mathbf{en\_events}(ss, \tau) \wedge \mathbf{en\_cond}(ss, \tau)\}$$

$$\textsc{execute}(ss, \tau, ss') \equiv$$
$$\text{let } \langle CS', IE', AV', O', CS'_a, IE'_a, AV'_a, I'_a\rangle \equiv ss' \text{ in}$$
$$\mathbf{next\_CS}(ss, \tau, CS') \wedge \mathbf{next\_CS}_a(ss, \tau, CS'_a) \wedge$$
$$\mathbf{next\_IE}(ss, \tau, IE') \wedge \mathbf{next\_IE}_a(ss, \tau, IE'_a) \wedge$$
$$\mathbf{next\_AV}(ss, \tau, AV') \wedge \mathbf{next\_AV}_a(ss, \tau, AV'_a) \wedge$$
$$\mathbf{next\_O}(ss, \tau, O') \wedge \mathbf{next\_I}_a(ss, \tau, I'_a)$$

Template ENABLED_TRANS returns the subset of transitions (where T is the HTS's full set of transitions), whose source states, triggering events,

Table 1 Template parameters provided by users

Construct	Start of Macro-step	Micro-step
states	**reset_CS**(ss, I, CS') **reset_CS**$_a(ss, I, CS'_a)$	**next_CS**(ss, τ, CS') **next_CS**$_a(ss, \tau, CS'_a)$
	en_states(ss, τ)	
events	**reset_IE**(ss, I, IE') **reset_IE**$_a(ss, I, IE'_a)$ **reset_I**$_a(ss, I, I'_a)$	**next_IE**(ss, τ, IE') **next_IE**$_a(ss, \tau, IE'_a)$ **next_I**$_a(ss, \tau, I'_a)$
	en_events(ss, τ)	
variables	**reset_AV**(ss, I, AV') **reset_AV**$_a(ss, I, AV'_a)$	**next_AV**(ss, τ, AV') **next_AV**$_a(ss, \tau, AV'_a)$
	en_cond(ss, τ)	
outputs	**reset_O**(ss, I, O')	**next_O**(ss, τ, O')
additional parameters	**macro_semantics** **pri**$(T) : 2^T$ **resolve**	

and guard conditions are all enabled in snapshot ss. Predicates **en_states, en_events,** and **en_cond** are template parameters for how the snapshot elements are used to determine the set of enabled transitions. Template EXECUTE is a relation over consecutive snapshots ss and ss', with unprimed elements referring to snapshot values before the transition τ executes and primed elements referring to snapshot values after the transition executes. The template definition uses parameters **next_X**, one for each snapshot element X, as placeholders for how the execution of a transition τ affects the individual snapshot elements. For example, executing a transition might result in a new set of current control states, updates to variable values, an update to the set of enabling events, and so on. There are five template definitions, including the above, that, taken together, reflect the identification of enabled transitions, the effects of inputs from the environment, and the effects of executing a micro- or macro-step.

2.4 Semantics Parameters

Our template-semantics definitions have a total of 22 parameters that represent variations on how a model's snapshot can change during execution.

There are eight **reset_X**(ss, I) predicates, each specifying how one snapshot element, $ss.X$, is updated to new value X' at the start of a macro-step due to new environmental inputs I. Example values for some of these parameters include

- **reset_IE**: Empty the set of events IE that were generated in the previous macro-step.
- **reset_I$_\mathbf{a}$**: Add inputs I to the snapshot element I_a, which records input events.
- **reset_CS**: Make no change to the set of current states CS.

There are also eight **next_X**(ss, τ, X') predicates, each specifying how a snapshot element, $ss.X$, is updated on execution of a transition τ. Example values for some of these parameters include

Table 2 Event-Related Semantics Parameter Values

Parameter	Parameter Value	Informal Definition
reset_IE(ss, I, IE')	$IE' = \emptyset$	Set IE to be empty
	$IE' = ss.IE$	Make no change to IE
	$IE' = I$	Set IE to the inputs I from the environment
	$IE' = ss.IE \cup I$	Add inputs I to the set of events IE
	$IE' = ss.IE \frown I$	Append inputs I to end of the queue IE of events
next_IE(ss, τ, IE')	$IE' = gen(\tau)$	Set IE to the events generated by τ
	$IE' = ss.IE \cup gen(\tau)$	Add the events generated by τ to the events already in set IE
	$IE' = ss.IE \frown gen(\tau)$	Append the events generated by τ to the end of queue IE
	$IE' = (ss.IE \setminus \{trig(\tau)\}) \cup gen(\tau)$	Remove the trigger event from IE, and add the generated events
	$IE' = tail(ss.IE) \frown gen(\tau)$	Remove head element from IE's queue, and append generated events
reset_IEa(ss, I, IE'_a)	$IE'_a = \emptyset$	Set IEa to be empty
	$IE'_a = \{head(ss.I_a)\}$	Set IEa to be the head element in event queue Ia
	$\exists q \in ss.I_a.[IE'_a = \{(q, head(q))\}]$	Set IEa to be the head element of an arbitirary input queue in Ia's set of queues
next_IEa(ss, τ, IE'_a)	$IE'_a = \emptyset$	Set IEa to be empty
	$IE'_a = ss.IE_a$	Make no change to IEa
	$IE'_a = ss.IE_a \cup gen(\tau)$	Add the events generated by τ to the events already in set IEa
	$IE'_a = ss.IE \frown gen(\tau)$	Append the events generated by τ to the end of queue IEa
reset_Ia(ss, I, I'_a)	$I'_a = \emptyset$	Set Ia to be empty
	$I'_a = I$	Set Ia to the inputs I from the environment
	$I'_a = ss.I_a \cup I$	Add inputs I to the events already in set Ia
	$I'_a = ss.I_a \frown I$	Append input events I to the end of input queue Ia
	$I'_a = tail(ss.I_a) \frown I$	Remove head element from queue Ia, and append the inputs I
	$\forall q \in ss.I_a.[I_a.q' = ss.I_a.q \frown directed(q, I)]$	Append each input i=(queue, event) to its appropriate destination queue
next_Ia(ss, τ, I'_a)	$I'_a = \emptyset$	Set Ia to be empty
	$I'_a = ss.I_a$	Make no change to Ia
	$I'_a = ss.I_a \cup gen(\tau)$	Add the events generated by τ to the events already in set Ia
	$I'_a = ss.I_a \frown gen(\tau)$	Append τ's generated events to the end of input queue Ia
	$I'_a = ss.I_a \setminus \{trig(\tau)\}$	Remove τ's triggering event from Ia
	$\forall q \in ss.I_a.[if\ \ (q, head(q)) \in trig(\tau)$ $then\ \ I_a.q' = tail(ss.I_a.q) \frown directed(q, gen(\tau))$ $else\ \ I_a.q' = ss.I_a.q \frown directed(q, gen(\tau))\ ]$	Remove τ's triggering event, and append each generated event i=(queue, event) to its appropriate destination queue
en_event(ss, τ)	$trig(\tau) \subseteq ss.IEa$	Transition's triggering event(s) must be in IEa
	$trig(\tau) \subseteq (ss.I_a \cup ss.IE)$	Transition's trigger(s) must be in Ia or IE
	$trig(\tau) \subseteq (ss.I_a \cup ss.IE_a)$	Transition's trigger(s) must be in Ia or IEa
	$trig(\tau) = head(ss.IE)$	Transition's trigger matches the head of queue IE

- **next_IE**: Update the set of enabling events IE to be exactly the events generated by τ
- **next_IE**: Add the events generated by τ to the set of events already in IE
- **next_AV**: Update the current variable values in AV based on τ's assignments

There are the three ENABLED_TRANS parameters that specify how the snapshot elements are used to identify enabled transitions. Function **pri** specifies a default priority scheme on transitions (e.g., transitions with higher-ranked source states might have priority over transitions with lower-ranked source states). Parameter **macro_semantics** specifies when new inputs are sensed from the environment (e.g., after every micro step, or when no transition is enabled). Parameter **resolve** specifies how to resolve concurrent assignments to shared variables.

The 22 parameters are listed in Table 1, organized by language construct. Parameters that are associated with the same language construct tend to

have compatible values. For example, the seven event-related parameters work together to determine which events can enable transitions and which events remain to be processed. Example parameter values for event-related semantics parameters are listed in Table 2.

2.5 Composition Operators

So far, we have discussed the execution of a single HTS. Composition operators specify how multiple HTSs execute concurrently and how they share information. Informally, a composition operator defines an "allowable (collective) step" that a collection of HTSs takes. What differentiates one composition operator from another are the conditions under which it allows, or forces, its component HTSs to take a step.

Our prototype code-generator generator supports variations of seven binary composition operators. Each operand is either an HTS or a collection of previously composed HTSs.

- ***interleaving:*** One of the operands takes a step if enabled, but not both.
- ***parallel:*** Both operands execute simultaneously if both are enabled. Otherwise, one or the other operand executes, if enabled.
- ***interrupt:*** Control passes between the operands via a set of interrupt transitions.
- ***sequence:*** One operand executes to completion, then the other operand executes to completion.
- ***choice:*** One operand is chosen to execute, and thereafter only the chosen operand executes.
- ***environmental synchronization:*** All HTS components that have a transition enabled by a specified *synchronization event* execute those transitions simultaneously. Otherwise, the operands' executions are interleaved.
- ***rendezvous:*** A pair of HTS components (one in each operand) execute simultaneously only if (1) one HTS component has an enabled transition that generates a specified *rendezvous event*, and (2) the other HTS component has a transition that is enabled by the same event. Otherwise, the operands' executions are interleaved.

Composition operators may be combined in the same model to affect different types of synchronization and communication among the model's components. We use the term *composition hierarchy* to refer to a model's composition structure, because the structure forms a binary tree (see the left side of Figure 3).

Figure 2 shows the composition hierarchy for a Ground-Traffic Control System [40] that is used throughout the paper to exemplify aspects of our approach. The airport-controller component responds to airplanes' requests to take off, land, and taxi by telling them which runway or taxiway to use. The models for runways (Figure 1) and taxiways keep track of the current

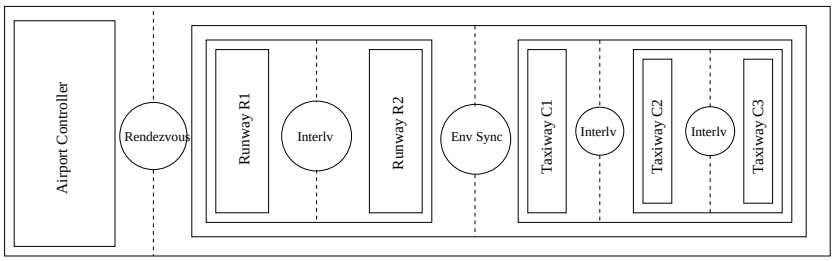


Fig. 2 Compositional Hierarchy for a Ground-Traffic Control System

states of the real-world entities they represent. The runways are interleaved with each other, as are the taxiways. The environmental synchronization operator synchronizes the runways with the taxiways, so that both are aware when an airplane is at the intersection of a runway and a taxiway. The rendezvous operator synchronizes the controller and the runways to ensure that they have a shared understanding of the status of the runways.

3 CGG Architecture

We use template semantics as the basis for realizing semantically configurable code generation. Because semantics is specified as parameter values, we can create a parameterized code-generator generator (CGG) that produces a code generator specialized to a particular modelling notation's semantics; the produced code generator transforms models from that modellng notation into representative Java programs.

Our prototype CGG is implemented using preprocessor directives and conditional compilation as a primitive form of generative programming. Other generative-programming technologies could have been explored [4, 5, 7, 22]. However, preprocessor directives were sufficiently powerful and its technology is stable. Moreover, it was not clear how easy it would be to express one of the semantics parameters, **macro-semantics**, as a plugable component: its values are not predicates or functions, like those of the other parameters, but rather are names that specify the granularity of an execution step.

The CGG source code is annotated with preprocessor directives that indicate the parts of the source that are specific to each supported parameter value. The user provides a file that lists a preprocessor #define declaration for each template parameter, specifying the value of that parameter. Compiling the CGG source code along with the parameter-definition file compiles only the parts of the CGG code that are associated with the specified parameter values, thereby producing a code generator for the user's modelling notation. We then execute the produced code generator on an input model, thereby generating a Java program whose executions match the model's execution traces.

Our prototype CGG supports 57 parameter values, roughly 2-5 values per parameter. We have not attempted to identify or implement a com-

plete set of parameter values, as "completeness" depends on the desired family of notations to be supported. We expect specifiers to devise new semantic variants as they try to model unusual problems. To that effect, we have structured the CGG program such that the preprocessor directives are highly localized, easing the task of adding support for new parameter values.

4 Generated Java Code

Java natively supports concurrency using threads and supports synchronization using monitors. Intuitively, mapping HTSs to Java threads and mapping composition operators to Java monitors seems like a good idea. The problem is that some template-semantics composition operators, such as rendezvous and environmental synchronization, require global knowledge to determine whether concurrent machines ought to execute simultaneously. For example, the environment synchronization operator, borrowed from process algebras, needs to know about all HTS components that have transitions enabled by some synchronization event. Collecting this information effectively synchronizes all of the program's threads, and the savings that would be gained from executing the HTS steps in parallel is not large enough to offset the cost of this synchronization.

Instead, we generate a single-threaded program that controls the execution of the program's concurrent components via method calls. The program's run-time structure resembles the structure of the model from which it is created. For example, the object model of the program generated from our model of the Ground-Traffic Control System (from Figure 2) is shown in Figure 3. Each composition operator and HTS is implemented as a (shaded) Java object, and these classes are organized as a tree that mirrors the model's composition hierarchy. Moreover, every HTS object has member variables that refer to local objects implementing local snapshot elements (CS, IE, IE_a, etc.). The snapshot elements I_a, AV, AV_a are shared, and every HTS object has references to these global objects.

The generated program simulates the "steps" of the model's possible behaviours. A step has two phases. In the first phase, the System object requests information about all enabled transitions in all HTSs. This request is triggered by the sensing of input events from the environment (object Inputs in Figure 3), and is recursively passed down the composition hierarchy, with each operator class requesting information from its operands'. At the leaf nodes of the hierarchy, each of the HTS objects identifies its enabled transitions, stores its results locally in member variables, and passes its results back to its parent node in the composition hierarchy. In turn, each operator class combines its operands' results and passes the information to its parent node, and so on until the System object receives all of the information.

In the second phase, execution decisions in the form of constraints flow from the System object down the composition hierarchy to the HTSs: every

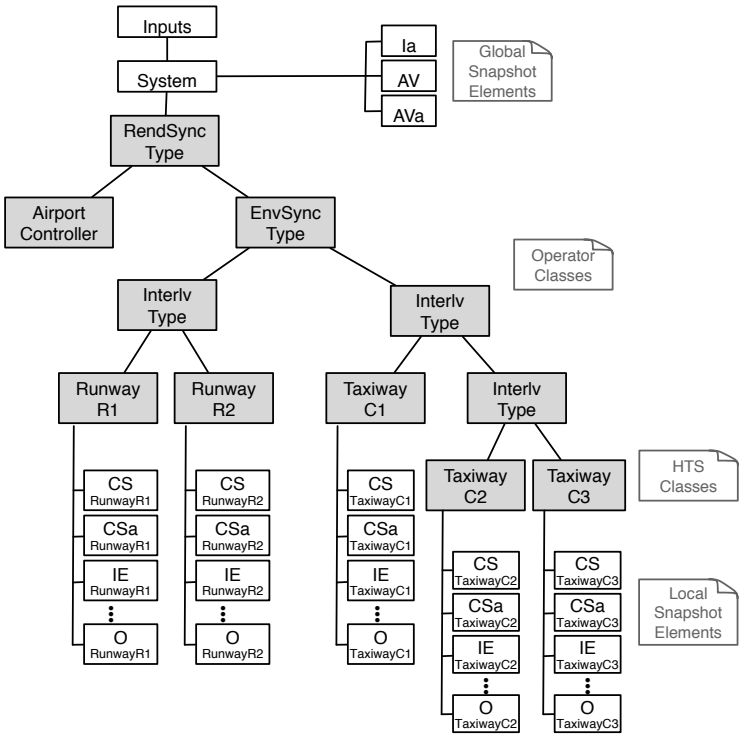


Fig. 3 Code structure for Ground Traffic Control System example. Shaded objects mimic the composition hierarchy of the model from Figure 2

operator object (1) receives constraints from its parent node, restricting which enabled transition(s) should be selected for execution, (2) possibly asserts additional constraints, and (3) recursively sends the cumulation of constraints to one or both of its operands. Constraints may be as specific as stipulating that a particular transition be executed or as general as requiring that some enabled transition execute. Constraints reach only the HTSs that are chosen to execute. Each chosen HTS executes a transition that satisfies its constraints and updates its snapshot.

In the rest of this section, we discuss the generated Java classes in more detail. The discussion is structured in terms of the phased execution described above: we consider each class's contribution to the identification of enabled transitions followed by each class's contribution to the selection and execution of transitions. The generated code preserves any nondeterminism in the model, which is useful for simulation and reasoning about all possible executions. In Section 5, we discuss ways of resolving nondeterminism, to produce deterministic code from nondeterministic models. We conclude with a discussion of how we optimize the generated code.

```
VAR Set enabledTrans
VAR Map syncTrans, rendTrans
VAR Transition exec
VAR EventSnap &Ia = globalIa
VAR VarSnap &AV = globalAV
VAR VarSnap &AVa = global AVa
VAR StateSnap CS
...
VAR EventSnap O
```

```
1: IsEnabled(set syncEv, rendEv; bool enabled)
2: enabled = false
3: for each transition τ in HTS do
4:    if EN_STATE(τ) ∧ EN_COND(τ) ∧ EN_EVENTS(τ) then
5:       enabled = true
6:       if τ is triggered by a rendezvous event e then
7:          rendEv.add(e)
8:          rendTrans.insert(e,τ)
9:       else if τ is triggered by a sync event e then
10:         syncEv.add(e)
11:         syncTrans.insert(e,τ)
12:      else
13:         enabledTrans.add(τ)
14:      end if
15:   end if
16: end for
```

```
1: Execute(event syncEv, rendEv)
2: if rendEv is not null then
3:    exec = rendTrans.lookup(rendEv)
4: else if syncEv is not null then
5:    exec = syncTrans.lookup(syncEv)
6: else
7:    exec = top priority τ ∈ enabledTrans
8: end if
9: {update snapshot elements}
10: Ia.NEXT(exec)
11: AV.NEXT(exec)
12: AVa.NEXT(exec)
13: CS.NEXT(exec)
14: CSa.NEXT(exec)
15: IE.NEXT(exec)
16: IEa.NEXT(exec)
17: O.NEXT(exec)
```

Fig. 4 Pseudocode for Taxiway HTS. Configurable code is highlighted in SMALL CAPS.

4.1 HTS - Enabled Transitions

A separate class is generated for each HTS in the input model. Figure 4 sketches the class generated for the C1Taxiway HTS from the Ground-Traffic Control System example. The Taxiway class contains a member variable for each of the HTS's snapshot elements, some of which are local objects and some of which are references to global snapshot objects. In addition, there are member variables that store information about enabled transitions.

Each HTS object is responsible for determining which of its HTS's transitions are enabled in the current snapshot. It has an IsEnabled() method that identifies the enabled transitions (shown on the left in Figure 4). Much of this task is done by methods that implement the semantic parameters **en_state**, **en_event**, and **en_cond** (lines 4-5). These methods compare a transition's source state, triggering event, and guard against the contents of the snapshot objects and determine whether the transition is currently enabled. IsEnabled() also computes and stores any enabledness information that is needed by any of the composition operators in the model: enabled rendezvous transitions are stored in rendTrans (lines 6-8), enabled transitions that are triggered by a synchronization event are stored in syncTrans (lines 9-11), and ordinary enabled transitions are stored in enabledTrans (lines 12-13). Abstract information about enabled transitions –such as that there exists some enabled transition, or that there exists a transition enabled by a particular synchronization event – are passed back to the HTS's parent node via assignments to the method's parameters (lines 1, 5, 7, 10).

```
VAR bool LEnabled, REnabled
VAR operand compToExecute

1: IsEnabled(bool enabled)
2: left.IsEnabled(LEnabled)
3: right.IsEnabled(REnabled)
4: enabled = LEnabled ∨ REnabled
```

```
1: Execute()
2: if LEnabled ∧ REnabled then
3:    compToExecute = choose left or right child
4:    compToExecute.execute()
5: else if LEnabled then
6:    left.Execute()
7: else if REnabled then
8:    right.Execute()
9: end if
```

Fig. 5 Pseudocode for interleaving composition.

4.2 Composition Operators - Enabled Components

In this section, we describe the Java implementations of composition operators. To ease presentation, we first assume that a model employs only one type of operator. In Section 4.4, we describe how an operator's implementation changes when it is combined with other types of composition. Details beyond the implementation sketches provided below can be found in [35].

A Java class is generated for each operator type used in the model, and an object is instantiated for each operator instance in the model. Thus, the code for our Ground-Traffic Control example includes three operator classes: rendezvous, environmental synchronization, and interleaving. The interleaving class is instantiated three times.

The implementations of composition operators are model independent. Each operator class has an IsEnabled() method that determines whether the operator's components have enabled transitions. This method (1) recursively calls the IsEnabled() methods of its two operands (each of which is either an HTS or a composition operator with its own operands), (2) combines its operands' enabledness information, (3) stores the results in member variables, and (4) passes the results to its parent node via pass-by-reference parameters.

4.2.1 Interleaving and Parallel Composition The IsEnabled() method for the interleaving operator is shown on the left in Figure 5. The parameter encodes the enabledness information that is returned. In interleaving, the only enabledness information is an enabled flag that indicates whether the operator has any descendent HTS with enabled transitions. The method calls the IsEnabled() methods of its two operands and stores the results in member variables (L/R)Enabled (lines 2-3). The method then computes the operator's own enabledness, which is *true* if either of the operands is enabled (line 4), and returns the result via its parameter (line 1). The IsEnabled() method for the parallel composition operator is the same as that for interleaving composition.

4.2.2 Environmental Synchronization The operators that synchronize the execution of multiple transitions have more intricate implementations. Figure 6 presents the pseudocode for the Java class that implements environmental synchronization composition. This operator introduces a set of synchronization events (*syncEvents*), and all enabled transitions in com-

ponent HTSs that are triggered by the same sync event execute simultaneously. Each instance of this composition operator has member variables *(L/R)Enabled* that record whether its left and right operands have enabled transitions. In addition, it has two other member variables, *(L/R)SynEv*, that record *all* synchronization events that trigger enabled transitions in the left and right operands, respectively[1]. The variable *syncEvents* refers to the static set of synchronization events associated with this instance of the operator, as declared in the model.

The IsEnabled() method collects enabledness information from its operands (lines 2-3), and then computes its own enabledness. An environmental synchronization operator is enabled if both of its operands have transitions enabled by one of the operator's *syncEvents* (line 5) or if either of its operands has a transition that is enabled by some *non* sync event (line 4). The operator passes back to its parent a flag indicating whether it is enabled and the set of sync events that enable its components' transitions.

4.3 Composition Operators - Execution Phase

At the end of the first phase, each object in the composition hierarchy is populated with information about the transitions enabled in its HTS or component HTSs. In the second phase, a subset of these transitions is selected for execution The selection process is incremental, with each composition operator contributing to the process by imposing constraints on the final selection of transitions. These constraints can be light constraints (e.g., an arbitrary enabled transition from among its right operand's HTSs) or can be a tight constraint (e.g., transitions enabled by a specific event).

The selection process starts at the top of the composition hierarchy with a call to the root node's Execute() method. Each operator class has an Execute() method that propagates and contributes selection constraints to its operands. In general, the method (1) receives selection constraints via its parameters, (2) possibly asserts additional, operation-specific constraints, and (3) recursively calls the Execute() method of one or both of its operands, providing an augmented set of constraints

At the end of the execution phase, the selection constraints reach the HTS objects, each of which selects and executes one of its enabled transitions that satisfies all imposed constraints.

4.3.1 Interleaving and Parallel Composition The Execute() method for an interleaving operator is shown on the right in Figure 5. If both of operator's operands are enabled, then one is nondeterministically chosen to execute (lines 2-4)[2]. Otherwise, the solely enabled operand is instructed to execute

[1] Actually, all sync events that enable the operands' transitions are passed to the parent node, rather than just those identified on line 5, because other operators in the composition hierarchy may be interested in these other events.

[2] The generated Java program uses random number generators to make such nondeterministic choices.

```
  VAR bool LEnabled, REnabled
  VAR set LSyncEv, RSyncEv
 1: IsEnabled(bool enabled, set syncEv)
 2: left.IsEnabled(LSyncEv,LEnabled)
 3: right.IsEnabled(RSyncEv,REnabled)
 4: un-sync = LEnabled ∨ REnabled
 5: sync = (LSyncEv ∩ RSyncEv ∩ syncEvents)≠ Ø
 6: enabled = un-sync ∨ sync
 7: syncEv = LSyncEv ∪ RSyncEv

 1: Execute(event constraint)
 2: {Case 1: enforce synchronization imposed by ancestor}
 3: if constraints is not null then
 4:    if constraint∈(LSyncEv ∩ RSyncEv ∩ syncEvents) then
 5:          left.Execute(constraint)
 6:          right.Execute(constraint)
 7:    else if constraint ∈(LSyncEv ∩ RSyncEv) then
 8:       compToExecute = choose left or right child
 9:       compToExecute.Execute(constraint)
10:    else if constraint ∈ LSyncEv then
11:       left.Execute(constraint)
12:    else if constraint ∈ RSyncEv then
13:       right.Execute(constraint)
14:    end if
15: else
16:    {Case 2: make local decisions about synchronization}
17:    sub_sync_events = (LSyncEv ∩ RSyncEv ∩ syncEvents)
18:    if sub_sync_events ≠ Ø ∧ LEnabled ∧ REnabled  then
19:       choice = choose sync or un-sync
20:    else if LEnabled ∨ REnabled then
21:       choice = un-sync
22:    end if
23:    if choice==sync {Case 2a } then
24:       event = choose an event in sub_sync_events
25:          left.Execute(event)
26:          right.Execute(event)
27:    else if choice==un-sync {Case 2b} then
28:       if LEnabled ∧ REnabled then
29:          compToExecute = choose left or right child
30:          compToExecute.Execute()
31:       else if LEnabled then
32:          left.Execute()
33:       else if REnabled then
34:          right.Execute()
35:       end if
36:    end if
37: end if
```

Fig. 6 Pseudocode for Environmental Synchronization

(lines 5-8). It is guaranteed that at least one of the operands is enabled, otherwise the operator's Execute() method would not have been invoked. The Execute() method for the parallel composition operator is almost the same, except that in parallel composition, if both operands are enabled then both are instructed to execute simultaneously.

4.3.2 Environmental Synchronization Synchronization operators have more complicated Execute() methods because of their potential for operator-specific transition-selection constraints. It is for this reason that we initially present each composition operator in isolation (i.e., we assume that models employ only one type of composition operator). The Execute() method for environmental synchronization is separated into three cases:

1. **Imposed constraint:** The invocation of Execute() includes as a parameter a constraint asserting that the selected transition(s) be triggered by a particular sync event. If the imposed synchronization-event constraint is one of the operator's own sync events, then the operator will synchronize its operands' executions: if both operands have transitions triggered by this event, then both are instructed to execute (lines 4-6). Otherwise, at most one operand may execute, and only if it has transitions enabled by the constraint's sync event (lines 7-14).
2. **Constraint Free:** The invocation of Execute() does not include an imposed synchronization-event constraint, which means the operator is free to impose its own constraint.
   (a) **Sync:** If both operands have transitions that are triggered by one of the operator's sync events, then the operator may choose to synchronize its operands' executions (lines 17-19), asserting a new event constraint (lines 17, 24-26).
   (b) **Nonsync:** The operator instructs one of its enabled operands to execute some transition that does not involve any of the operator's sync events (lines 27-36).

In all cases, all imposed and new constraints are propagated in the recursive calls to the operands' Execute() methods.

Note that if both the new constraint and the nonsync cases are both possible, then one is nondeterministically chosen. Thus, the composition hierarchy of a model *does not* impose a priority scheme among composition operators and their selection of enabled transitions to be executed.

4.3.3 Other Composition Operators. Our CGG also supports rendezvous, interrupt, sequence, and choice composition operators. The Java classes generated for these operators resemble the class generated for the environmental synchronization operator, in that they introduce member variables to keep track of operator-specific enabledness information, and their Execute() methods are structured into three cases: accommodating an imposed transition-selection constraint, imposing an operator-specific constraint, or imposing no constraint. The details of how all supported composition operators are implemented can be found in [35].

4.4 Heterogenous Compositional Hierarchies

In this section, we describe how the implementations of composition operators, as presented in the previous section, change when multiple types of operators are used in the same model. For example, most of the composition operators track distinct enabledness information: environmental synchronization keeps track of the synchronization events that trigger enabled transitions, and rendezvous composition keeps track of both the rendezvous events that are generated by enabled transitions and the rendezvous events that would trigger transitions.

In a heterogeneous composition hierarchy, each operator node must keep track of, and pass as IsEnabled() parameters, all information needed by any operator in the composition hierarchy. Thus, all operator classes have member variables for all types of enabledness information, and all of their IsEnabled() methods include parameters for these data. In fact, all of the operator classes' IsEnabled() methods are the same, except for how their respective *enabled* parameters are computed, which are operator-specific and are as described in the previous section.

The operators' Execute() methods must expand to accommodate any transition-selection constraint imposed by any type of ancestor operator. Among all of the composition operators supported by CGG, there are only four types of constraints:

1. A particular (interrupt) transition
2. Transitions triggered by syncEvents
3. Some (one) transition triggered by a rendezvous event
4. Some (one) transition that generates a rendezvous event

Of these, an Execute() method would receive at most one constraint specifying a particular transition and at most one constraint specifying a particular event (and would receive both only if the specified transition were triggered by the specified event). In the case of an imposed constraint, an operator asserts its semantics within the set of enabled transitions satisfying the constraints. If no constraint is imposed, then the operators' Execute() methods behave as described in the previous sub-section.

4.5 HTS - Execute Transitions

Each HTS object is responsible for making the final selection of transitions to execute and for realizing their executions. Each HTS also has an Execute() method (shown on the right in Figure 4) that is called when the HTS is instructed to execute as part of a step. In this method, one of the enabled transitions identified by IsEnabled() is chosen for execution. Invocation may include transition-selection constraints as parameters (line 1), in which case the transition chosen for execution must satisfy the constraints (lines 2-5). If there are no selection constraints, then the top-priority transition in enabledTrans is selected to execute (lines 6-7). In the end, the chosen

transition is "executed" via inline procedures that implement the **next_X** template parameters, which in turn update all of the snapshot elements (lines 10-17).

4.6 Optimizations

The CGG employs a number of simple optimizations to improve the performance of the generated Java code. Some optimizations are based on the modelling notation's semantics. For example, the Java classes for unused snapshot elements are not generated. As another example, the search for enabled transitions is done in order of the transitions' priority (based on the model's composition hierarchy, the HTS's state hierarchy, and the value of the priority template parameter). Thus, when an enabled transition is found, no transition of lower priority is checked.

Most optimizations are model independent. For example, some computations can be statically computed or optimized, such as the determination of which HTS states are entered and exited when a transition executes. As another example, if a composition operator is associative, then consecutive applications of that operator can be compressed into a single operator with multiple operands. A flattened composition hierarchy results in a more efficient execution step because there are fewer recursive calls and less caching of enabledness information.

5 Resolving Nondeterminism

The code described in the previous section simulates a model's nondeterminism. Such a semantics-preserving transformation is useful during modelling and analysis, but is not appropriate for a deployable implementation. Our CGG can either generate a nondeterministic program that completely simulates the input model or generate a deterministic program that satisfies the model's specification.

Our general strategy for resolving nondeterminism is to use priorities. If the priorities provided by the specifier – in the form of explicit priorities on transition labels, on synchronization events, the notation's priority scheme **pri** – are not sufficient to make the model deterministic, then we impose default priorities on the remaining nondeterministic choices. Because the specifier has implicitly indicated that all choices are equally valid, any default priority that we choose should be acceptable. In cases where a default priority would introduce an asymmetry that could lead to unfair executions, we rotate priority among the enabled entities:

- ***HTSs:*** simultaneously enabled transitions within an HTS, if they are not prioritized by explicit priorities on the transition labels or by the template parameter values, are "prioritized" according to the order in which they are declared.

- ***interleaving:*** the interleaving of two simultaneously enabled components is "prioritized" by alternating which component is executed when both are enabled.
- ***synchronized operations:*** simultaneously enabled synchronized transitions (enabled by multiple simultaneous sync or rend events), if not prioritized by an explicit ordering on the events, are "prioritized" by the order in which the events are declared.
- ***synchronized operations:*** if synchronized and non-sychronized transitions are simultaneously enabled, the synchronized transitions are given higher priority, so that components do not miss the opportunity to react to a synchronization event.
- ***rendezvous operations:*** if components can synchronize either by sending or receiving a rend event, the role (sender or receiver) that a component plays alternates.

The result of these decisions is a deterministic program that satisfies the model. A beneficial side effect is improved performance, because less enabledness information is communicated and there is no generation of random choices.

6 Evaluation

We report on the correctness, efficiency, and limitations of our work.

6.1 Correctness

To assess whether our generated code matches the semantics of its corresponding model, we compared the code's sequence of snapshots, taken at the end of execution steps, against the model's sequence of snapshots for the same sequence of input events. We designed a test suite of models that exercised different parts of the CGG. The test suite covered each implemented template-semantics parameter and each composition operator at least once, covered all pairs of composition operators, and tested more complicated hierarchies involving some of the more complicated operators. All testing was done on deterministic programs. Nondeterministic programs were manually inspected.

In order to test more complex composition hierarchies involving some of the more sophisticated operators, we also evaluated CGG on the Ground-Traffic Control System that was introduced in Section 2.5. This example also introduced a new template-parameter value (e.g., an enabling event persists until it triggers some transition). We generated the code for this model and tested it on input sequences that exercise a number of safety properties (e.g., an airplane can taxi across a runway only if the runway is not in use). We inspected the code's execution traces and verified that they conform to the model's traces.

6.2 Efficiency

We also evaluated the performance of our generated code, as an assessment of the performance penalty for supporting semantic configurability. We compared our generated code to that of three commercial tools: IBM's Rational Rose Realtime (Rose RT) [20], IBM Rational Rhapsody (Rhapsody) [39], and SmartState [1], each of which generates code for a single modelling notation. For each tool, we expressed its notation in template semantics and produced our own code generator for that notation. We then used the code generators to generate Java programs for four models:

- *PingPong* is an example model provided in the distribution of Rose RT. It consists of two concurrent machines that execute a simple request-reply protocol.
- An *Elevator* for a three floor building, whose components model the controller, service-request buttons, the engine, and the door and door timer.
- A *Heater*, whose components model the controller, the furnace, and sensors in the room.
- A hotel-room *Safebox*, whose components model a keypad for entering security codes, a display, and the status of the lock.

The *PingPong* model is a simple request-reply protocol that passes a token between two components a set number of times before terminating. The other three models are small but typical software controllers for embedded systems. Each of the embedded-system models is composed with an appropriate environment component that feeds input events to the system model. By forming a closed-world model of the system and its simulated environment, we are able to evaluate the performance of the generated code without having to interact with the program while it executes.

The first two studies, shown in Table 3, compare our generated code against that generated by Rose RT and Rhapsody. These two tools support the Unified Modelling Language (UML) and have similar semantics: communication between machines is via message passing, all generated messages are sent to a single global queue[3], and only the event at the head of this queue can trigger transitions. One difference between them is that in Rose RT a message event triggers only one (compound) transition, whereas in Rhapsody an event can initiate a sequence of transitions.

Using CGG, we generated code generators that simulate the semantics of the Rose RT and Rhapsody code generators. We ran the code generators on all four models, and then measured the execution times of the generated programs. The results reported in Table 3 are the average execution times over 10 runs, with each run consisting of 100,000 iterations between the system and environment components in *Heater*, *Elevator*, and *Safebox*, and 500,000 iterations between the Ping and Pong components in *PingPong*.

[3] Both Rose RT and Rhapsody allow multiple event queues and allow the specifier to indicate which machines share which event queues.

Table 3 UML Comparisons (seconds)

Model	Rose RT	Rhapsody	CGG-Det
PingPong-UML	0.5	1.3	1.2
Elevator-UML	3.5	18.8	53.3
Heater-UML	0.4	2.3	5.3
Safebox-UML	1.1	3.2	4.5

Table 4 Statecharts Comparisons (seconds)

Model	SmartState	CGG-Det
PingPong-SS	1.3	1.6
Elevator-SS	16.6	14.3
Heater-SS	3.9	2.6
Safebox-SS	6.8	6.2

All runs were performed on a 3.00Ghz Intel Pentium 4 CPU with 1GB of RAM, running Windows XP Professional Version 2002. We used the default code generation settings of both Rhapsody and Rose RT. Alternative settings (e.g., the time model setting of Rhapsody) were not applicable to our models, and so the possible code generation optimizations afforded by these settings were not applied in our evaluation. On average, deterministic CGG-generated programs (CGG-Det) took 8.8 times longer to run than Rose RT generated programs, and 1.9 times longer to run than Rhapsody generated programs. The deterministic CGG version of *PingPong* performed slightly better than the Rhapsody version. Similar comparisons with nondeterministic CGG-generated programs had similar results. This is not surprising given that all of the models in our evaluation suite are deterministic.

We also generated a statecharts-based code generator and compared its generated code to that generated by SmartState. SmartState semantics uses parallel composition and broadcasting of events. In addition, SmartState assumes open-world models, so we removed the environmental component from each input model and provided an application wrapper that generates environmental events for the model. The performance results are summarized in Table 4. On average, SmartState generated programs ran 1.2 times longer than deterministic CGG generated programs (CGG-Det). The SmartState generated *PingPong* program slightly outperformed the deterministic CGG generated program.

Overall, the cost of semantically configurable code generation appears to vary with the semantics chosen and the number of concurrent components. Our statecharts-based code generator performs slightly better than SmartState on the larger models, but not as well on the toy model *PingPong*. Our UML-based code generators are competitive with Rhapsody for models with fewer components (*PingPong*, *Heater*, *Safebox*), but less so on the larger model, *Elevator*. Rose RT significantly outperforms Rhapsody and CGG-Det on all models. We believe that with an investment in further

optimizions to CGG generated code, this performance gap can be reduced. For example, when the input model does not use any of the synchronization operators, each HTS could run on its own thread and the coordination of selecting which transitions to execute could be loosened.

In addition to the above case studies, we generated code for the Ground-Traffic Control System described in Section 2.5, composed with an environment component as in the embedded system case studies. The code for this case study could not be compared with that of commercial code generators, because it uses notation semantics and composition operators (namely environment synchronization and rendezvous) that are not supported by the other tools. The average execution time over 10 runs, with each run consisting of 100,000 iterations between the system and environment components was 15.1 seconds. The runs were performed on the same platform as the other case studies.

6.3 Limitations

The most significant limitation of our work is its inefficiency. In the current implementation, the generated code is single threaded, and each step of the program involves consulting all HTS and composition-operator objects to identify an admissible set of transitions to execute. That said, we were pleasantly surprised by how well our generated code performed compared to code generated by commercial tools. Moreover, there are several additional optimizations that could be explored. The most promising of these would be to detect when the input model does not use any of the composition operators involving synchronization (parallel, environmental synchronization, and rendezvous); and in these cases, replace the globally-coordinated step with a more distributed execution step. For example, any HTS that does not have an ancestral synchronization operator in the composition hierarchy could execute on its own thread.

A second limitation is with the extensibility of the CGG tool. Most obviously, the scope of the CGG is limited to what can be expressed using template semantics. Thus, adding new syntax and accompanying semantics (e.g., real-time clocks and timers) is outside the scope of the CGG. Adding new template-parameter values is the easiest change to make because the CGG has been structured to localize the code pertaining to parameter values. In contrast, implementing a new composition operator is not so easy. In addition to implementing a new class for the operator, the HTS and other operator classes would have to be changed to compute and communicate any new enabledness information needed by the new operator, and to realize and communicate any new transition-selection constraints asserted by the new operator.

7 Related Work

Most model-driven-engineering environments are centred on a single modelling notation that has a single semantics [8,20,38,39]. Configurability in such systems is geared more towards flexibility in the target language or platform [9,14] than in the modelling notation. There are a few exceptions. For example, Rational RoseRT and Rhapsody have options to choose whether event queues are associated with individual objects, groups of objects, or the whole model. Rhapsody allows both parallel and interleaving execution of concurrent regions and objects. BetterState supports multiple priority schemes on transitions and choices on the ordering of state entry/exit actions vs. transition actions. Such options allow the specifier some control over the modelling notation, but are not nearly as rich as our template-semantics parameters.

Our work can be viewed as an instance of generative programming, where template semantics is a domain-specific language for describing "features" of modelling notations, and CGG is a generator for a family of model-driven code generators. What distinguishes our work from typical generative programming is that the "features" are not functional requirements or components, but instead are semantic parameters of a general-purpose behavioural modelling notation.

Meta-modeling [32] notations are more expressive than template semantics in describing modelling notations. However, their focus is on the abstract syntax of a modelling notation and not on its semantics. Model transformation technologies, such as QVT [31] and graph grammars [25], facilitate the transformation of models whose meta-models are well-defined. However, it is not clear that these technologies will help meta-model designers to create code generators for their modelling notations. Instead, the state of the art in code generation seems to be to support configurable language features [13,15], and to embed the semantics of those features in the implementation of the code generator.

There are mechanisms other than template semantics, such as hypergraphs [34], inference graphs [12], graph grammars [3], structured operational semantics rules [6,12], higher-order logic [10], that support semantically configurable modelling notations and that have been used to create semantically configurable tools. More generally, compiler generators [23] are able to construct compilers directly from a language's semantics expressed using denotational semantics [2,33], operational semantics and rewrite rules [16, 17], natural semantics [11], and language algebras [24]. A key disadvantage of these approaches is that one has to write a complete semantics for the modelling notation, or at least must provide a complete definition of the semantic mapping [3]. The main premises behind our work on semantically configurable modelling environments are that (1) writing a notation's formal semantics is hard (sometimes worthy of a research publication), and (2) we can simplify the task of writing semantics definitions by taking advantage of the commonalities among notations' semantics. In the template-semantics

approach, specifying the semantics of a modelling notation is reduced to providing a collection of relatively small semantics-parameter values that instantiate a predefined semantic mapping. The trade-off is that our CGG creates code generators only for template-semantics expressible notations – although template semantics has been shown to be expressive enough to represent a wide variety of language semantics [29,30,37], and the CGG is structured to facilitate the adding of new template-parameter values.

8 Conclusion

In this work, we explore template semantics as a parse-able semantics description that enables semantically configurable code generation. Configurability is in the form of semantics parameters, so that the specifier is spared from having to provide a complete semantics definition.

We built a proof-of-concept code-generator generator that supports 7 different composition operators, 22 semantics parameters, and 57 parameter values that can be combined in multiple ways. Using this environment, we are able to create and generate code for models whose semantics vary from that of standard modelling notations, in ways that better fit the problem being modelled. We demonstrate the utility of our approach on an airport-runway model that could not so easily have been modelled and implemented using existing specification notations or code generators.

We view semantically configurable MDE as an appropriate compromise between a general-purpose, single-semantics notation that has significant tool support and a domain-specific language that has a small user base and few tools. Another potential use is to provide tool support for UML, which has a number of semantic variation points that match template semantics' variation points [37]. Our technology does not yet compete with commercial-grade code generators, but its future looks promising enough to continue investigating.

References

1. ApeSoft. Smartstate v4.1.0. http://www.smartstatestudio.com, 2008.
2. A. W. Appel. Semantics-directed code generation. In *Proc. ACM Sym. on Prin. Prog. Lang. (POPL'85)*, pages 315–324. ACM Press, 1985.
3. L. Baresi and M. Pezzè. Formal interpreters for diagram notations. *ACM Trans. on Soft. Eng. Meth.*, 14(1):42–84, 2005.
4. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Soft. Eng. Meth.*, 1(4):355–398, 1992.
5. C. Cleaveland. *Program Generators with XML and Java*. Prentice-Hall, 2001.
6. R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Sci. of Comp. Prog.*, 41(1):39–47, Jan 2002.
7. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.

8. D. Harel et al. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Soft. Eng.*, 16(4):403–414, April 1990.
9. A. D'Ambrogio. A model transformation framework for the automated building of performance models from uml models. In *Proc. Intl. Work. on Soft. and Perf. (WOSP'05)*, pages 75–86. ACM Press, 2005.
10. N. A. Day and J. J. Joyce. Symbolic functional evaluation. In *TPHOLs*, volume 1690 of *LNCS*, pages 341–358. Springer, 1999.
11. S. Diehl. Natural semantics-directed generation of compilers and abstract machines. *Form. Asps. of Comp.*, 12(2):71–99, October 2000.
12. 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.
13. G. S. S. et al. Clearwater: extensible, flexible, modular code generation. In *Proc. IEEE/ACM Intl. Conf. on Aut. Soft. Eng.(ASE'05)*, pages 144–153, New York, NY, USA, 2005. ACM Press.
14. J. Floch. Supporting evolution and maintenance by using a flexible automatic code generator. In *Proc. Intl. Conf. on Soft. Eng. (ICSE'95)*, pages 211–219. ACM Press, 1995.
15. J. Grundy et al. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In *Auto. Soft. Eng. (ASE)*, pages 25–36, 2006.
16. J. Hannan. Operational semantics-directed compilers and machine architectures. *ACM Trans. Prog.. Lang. Sys.*, 16(4):1215–1247, 1994.
17. J. Hannan and D. Miller. From operational semantics to abstract machines. *Math. Struct. Comp. Sci.*, 2(4):415–459, 1992.
18. D. Harel. On the formal semantics of statecharts. *Symp. on Logic in Comp. Sci.*, pages 54–64, 1987.
19. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Soft. Eng. and Meth. (TOSEM)*, 5(3):231–261, 1996.
20. IBM Rational. Rational Rose RealTime v7.0.0. http://www.ibm.com/rational, 2005.
21. ITU-T. Recommendation Z.100. Specification and Description Language (SDL). Technical Report Z-100, International Telecommunication Union - Standardization Sector, 1999.
22. N. Jones, C. Gomard, and P. Sestoft, Eds. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
23. N. Jones, Ed. *Semantics-Directed Compiler Generation*, volume LNCS 94. Springer-Verlag, 1980.
24. J. L. Knaack. *An algebraic approach to language translation*. PhD thesis, University of Iowa, 1995.
25. A. Königs and A. Schürr. Tool integration with triple graph grammars – a survey. *Elect. Notes in Theor. Comp. Sci.*, 148(1):113–150, 2006.
26. Y. Lu, J. M. Atlee, N. A. Day, and J. Niu. Mapping template semantics to SMV. In *Proc. of Auto. Soft. Eng. (ASE'04)*, pages 320–325, 2004.
27. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.
28. J. Niu. *Metro: A Semantics-Based Approach for Mapping Specification Notations to Analysis Tools*. PhD thesis, University of Waterloo, 2005.
29. J. Niu, J. M. Atlee, and N. A. Day. Template Semantics for Model-Based Notations. *IEEE Trans. on Soft. Eng.*, 29(10):866–882, October 2003.

30. J. Niu, J. M. Atlee, and N. A. Day. Understanding and comparing model-based specification notations. In *Proc. IEEE Intl. Req. Eng. Conf. (RE'03)*, pages 188–199. IEEE Computer Society Press, 2003.
31. Object Management Group. Revised submission for MOF 2.0 Query/View/Transformation RFP. http://www.omg.org/docs/ad/05-03-02.pdf.
32. Object Management Group. *Meta Object Facility Core Specification*, Formal/06-01-01, 2006.
33. L. Paulson. A semantics-directed compiler generator. In *Proc. ACM Sym. on Prin. of Prog. Lang. (POPL '82)*, pages 224–233. ACM Press, 1982.
34. M. Pezzè and M. Young. Constructing multi-formalism state-space analysis tools. In *IEEE Int. Conf. on Soft. Eng. (ICSE)*, pages 239–249. ACM Press, 1997.
35. A. Prout. *Parameterized Code Generation From Template Semantics*. Master's thesis, School of Computer Science, University of Waterloo, 2005.
36. A. Prout, J. Atlee, N. Day, and P. Shaker. Semantically configurable code generation. In *ACM/IEEE Int. Conf. on Mod. Driven Eng. Lang. and Sys.*, pages 705–720, 2008.
37. A. Taleghani and J. M. Atlee. Semantic variations among UML statemachines. In *ACM/IEEE Int. Conf. on Mod. Driven Eng. Lang. and Sys.*, pages 245–259, 2006.
38. Telelogic. Telelogic TAU SDL Suite. http://www.telelogic.com/corp/products/tau/sdl/index.cfm.
39. Telelogic. Rhapsody in J v7.1.1.0. http://modeling.telelogic.com/products/rhapsody/index.cfm, 2007.
40. T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *Proc. Intl. Symp. on Soft. Test. and Anal. (ISSTA'02)*, pages 169–179. ACM Press, 2002.
41. P. Zave and M. Jackson. A call abstraction for component coordination. In *International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction*, 2002.