Semantically Configurable Code Generation

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

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

Abstract. In model-driven engineering (MDE), software development is centred around a formal description (model) of the proposed software system, and other software artifacts are derived directly from the model. We are investigating *semantically configurable* MDE, in which specifiers are able to configure the semantics of their models. The goal of this work is to provide a modelling environment that offers 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 present a semantically configurable code-generator generator, which creates a Java-code generator for a modelling notation given the notation's semantics expressed as a set of parameter values. We are able to simulate multiple different model-based code generators, 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

A critical aspect of model-driven engineering is keeping the modelling tools – that is, the editing environments, model analyzers, and code generators – in sync with the semantics of their respective modelling notation as the notation evolves (e.g., statecharts, UML) or as new domain-specific notations are created. Maintaining these correspondences is time-consuming. Small changes to the semantics of a notation often result in wide-spread changes to the supporting tools.

To address this problem, we are investigating the feasibility of *semantically configurable* modelling tools as a way of avoiding the problem of maintaining separate tools for separate modelling notations. Semantically configurable tools enable specifiers to create or customize their own modelling notations, and yet still have access to tools for editing, manipulating, and analyzing models in those notations. Note that our work is distinct from complementary efforts on configurable tools [9,11] that support configurability with respect to the abstract syntax of a family of notations [23], but not with respect to semantics.

In previous work, we introduced *template semantics* [21] as a way to configure semantics definitions. 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

K. Czarnecki et al. (Eds.): MoDELS 2008, LNCS 5301, pp. 705–720, 2008.
© Springer-Verlag Berlin Heidelberg 2008

can be described succinctly 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, statecharts variants, Petri-Nets, SDL, and SCR. Because a notation's semantics can be expressed as a collection of parameters, it can be parsed, and we are starting to develop semantically parameterized tools [19].

In this paper, we present our first prototype of a semantically configurable code-generator generator (CGG) that creates Java code generators. The CGG takes a description of the 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

- **Representing composition operators in Java:** The Java scheduler imposes an interleaving semantics on concurrent threads, whereas many model-based concurrency operators are tightly synchronized (e.g., 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 modelling, such nondeterminism in source code is unnatural. The transformation of a model into source code may involve decisions to eliminate nondeterminism.
- **Optimizing the generated code:** A key concern of this work is how efficient CGG-generated code can be, given that our configurable CGG that is general enough to support a family of modelling notations.

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. We describe our code-generator generator and the architecture of the generated code in Sections 3 and 4. In Section 5, we discuss techniques for resolving nondeterminism in the model when generating deterministic code. We evaluate our work in Section 6 by comparing the performance of our CGG-generated code to the performances of code generated by three notation-specific code generators: Rational Rose RT [15], Rhapsody [28], and SmartState [1]. We conclude the paper with discussions on related work, limitations, and future work.

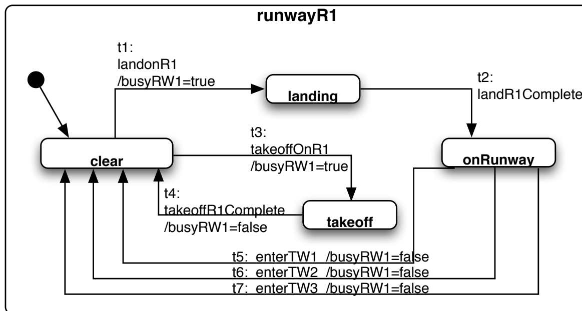


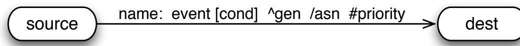
Fig. 1. Example HTS

2 Modelling Notation

In this section, we describe the syntax and configurable semantics of our modelling notation. We achieve configurability using template semantics [21].

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 [14]. 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).

2.2 Semantic Domain

The semantics of an HTS is defined in terms of sequences of snapshots, where a *snapshot* records information about the model's execution at a discrete point in the execution. The basic snapshot elements are

CS - the set of current states
IE - the set of events to be processed
AV - the current variable-value assignments
O - the set of 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
I_a - data about inputs to the HTS
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. The semantics of an HTS is the set of possible execution sequences.

2.3 Parameterized Semantics

Semantics are defined in terms of functions and relations over snapshot elements (the semantic domain). However, even though different modelling notations have

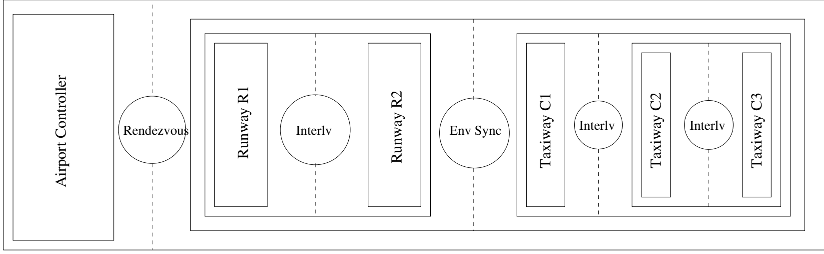


Fig. 2. Compositional Hierarchy for a Ground-Traffic Control System

many of the same language constructs and are defined over the same snapshot information, they vary in their use of that information.

In previous work, we developed a formalism called *template semantics* [21] in which semantics 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:

$$\textsc{enabled_trans}(ss, T) \equiv$$
$$\{\tau \in T \mid \mathbf{en_states}(ss,\tau) \wedge \mathbf{en_events}(ss,\tau) \wedge \mathbf{en_cond}(ss,\tau)\}$$

$$\textsc{execute}(ss, \tau, ss') \equiv$$
$$\text{let } \langle CS', IE', AV', O', CS'_a, IE'_a, AV'_a, I'_a \rangle \equiv ss' \text{ in}$$
$$\mathbf{next_CS}(ss,\tau,CS') \wedge \mathbf{next_CS}_a(ss,\tau,CS'_a) \wedge \mathbf{next_IE}(ss,\tau,IE') \wedge \mathbf{next_IE}_a(ss,\tau,IE'_a) \wedge$$
$$\mathbf{next_AV}(ss,\tau,AV') \wedge \mathbf{next_AV}_a(ss,\tau,AV'_a) \wedge \mathbf{next_O}(ss,\tau,O') \wedge \mathbf{next_I}_a(ss,\tau,I'_a)$$

Definition ENABLED_TRANS returns the subset of transitions whose source states, triggering events, 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. Definition EXECUTE uses template parameters **next_X**, one for each snapshot element X, as placeholders for how the execution of a transition τ affects the individual snapshot elements.

Our template semantics has 22 parameters that represent variations on how a model's snapshot can change during execution: when inputs are sensed from the environment, how inputs update the snapshot, how the snapshot determines the set of enabled transitions, priority among transitions, how a transition's actions update the snapshot, and how conflicting assignments to variables are resolved. We do not claim to have identified a complete set of template parameters, but we do believe that the set of parameters is relatively stable.

2.4 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.

Our prototype supports variations of seven binary composition operators, whose operands are either HTSs or collections of previously composed HTSs.

- ***interleaving:*** At most one enabled operand executes in a step
- ***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.
- ***choice:*** One operand is chosen to execute, and only the chosen operand executes.
- ***environmental synchronization:*** The operator specifies a set of *sync events*. When a sync event occurs, descendant HTSs that have a transition enabled by the event execute simultaneously. Otherwise, the operands' executions are interleaved.
- ***rendezvous:*** The operator specifies a set of *rend events*. The operands execute simultaneously only if (1) one operand has an enabled transition that generates a rend event, and (2) the other operand has a transition that is enabled by that 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 specified by Yavuc-Kahveci and Bultan [29]. 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 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 have developed a prototype code-generator generator (CGG) that is parameterized by our template parameters. The CGG produces notation-specific code generators that can transform models into representative Java programs.

The CGG source code is annotated with preprocessor directives that encapsulate the source code that is specific to each parameter value. The user provides a file that specifies the value of each template parameter. Thus, compiling the CGG source code, along with the semantics-parameter definition file, compiles only the parts of the CGG code that are associated with the specified parameter

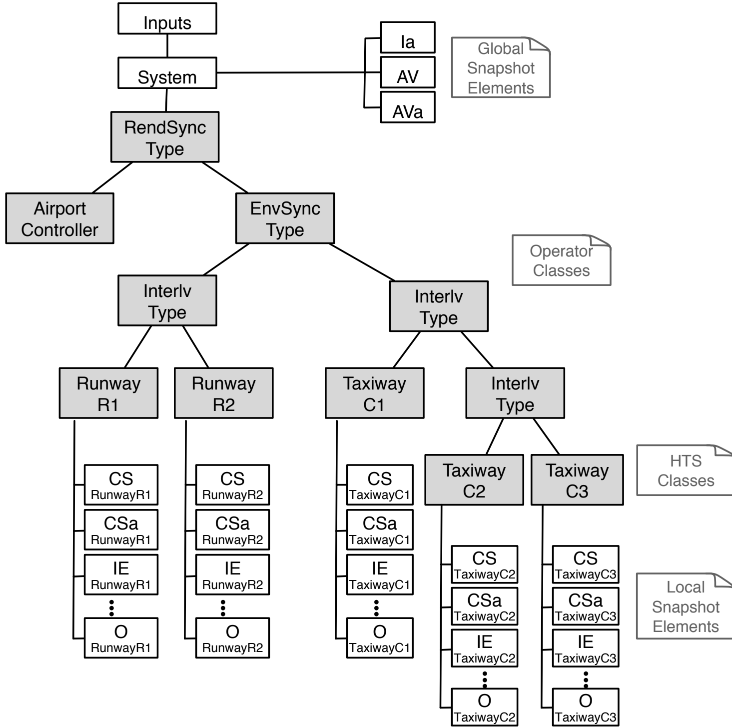


Fig. 3. Code structure for Ground Traffic Control System example. Shaded modules mimic the composition hierarchy of the model from Figure 2.

values, thereby producing a notation-specific code generator. 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 implement a complete set of parameter values, as this work is simply a proof of concept. In fact, we do not even claim to have identified a complete set of parameter values; 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

The generated code is a single-threaded program that controls the execution of its concurrent components via method calls. One might instead consider mapping HTSs to separate Java threads. The problem is that some composition operators require global knowledge to determine whether concurrent machines ought to

execute simultaneously: for example, the environment synchronization operator needs to know about all components that have transitions enabled by some sync 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.

The object model of the generated program for 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. 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. The first phase is an enabledness check: Triggered by the sensing of input events from the environment (module Inputs in Figure 3), the System module requests information about all enabled transitions. This request is recursively passed down the composition hierarchy, with each operator class requesting information from its operands' modules. At the leaf nodes of the hierarchy, each of the HTS modules identifies its enabled transitions, store 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 module receives all of the information.

In the second phase, execution decisions in the form of constraints flow from the System module down the composition hierarchy to the HTSs: (1) every operator receives execution constraints from its parent node, (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.

The rest of this section describes the generated Java classes in more detail. 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 how we produce deterministic code from nondeterministic models.

4.1 HTSs

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 to store information about enabled transitions.

Each HTS object is responsible for determining which of its HTS's transitions are enabled in the current snapshot and for executing the HTS's transitions. It has an IsEnabled() method that identifies the enabled transitions (shown on

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

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: in the case of Taxiway C1, if a rendezvous transition is enabled then it is stored in member variable rendTrans (lines 6-8), and if an enabled transition is triggered by a synchronization event then it is stored in syncTrans (lines 9-11). Ordinary enabled transitions are stored in member variable enabledTrans (lines 12-13). Abstract information about enabled transitions, sync events that enable transitions, and so on are passed back to the HTS's parent node via assignments to the IsEnabled() method's parameters (lines 1, 5, 7, 10).

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 execution constraints as parameters (line 1), in which case the transition chosen for execution must satisfy the constraints (lines 2-5). If there are no execution 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).

4.2 Composition Operators

The implementations of composition operators are model independent. A class is generated for each operator type used in the model, and an object is instantiated

```
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.

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.

Each operator class has an IsEnabled() method that determines which of the operator's operands are enabled. The method for an interleaving operator is shown on the left in Figure 5. The method calls its operands' IsEnabled() methods and stores the results in member variables (L/R)Enabled (lines 2-3). The method then computes the operator's enabledness, which is *true* if either of the operands is enabled (line 4), and returns the result via its parameter (line 1). The IsEnabled() methods of other operator types have a similar implementation, differing primarily in how they determine their own enabledness (line 4) and what information they return via their parameters (line 1). For example, the IsEnabled() method of an environmental synchronization operator returns not only a flag indicating whether the operands are enabled but also the set of sync events that enable its operands' transitions.

Each operator class also has an Execute() method that conveys execution constraints to the operator and its operands. The method for an interleaving operator is shown on the right in Figure 5. If both operands are enabled, then one of the components is nondeterministically chosen to execute (lines 2-4). Otherwise, the solely enabled operand is instructed to execute (lines 5-8). One of these three cases is guaranteed to hold, because an operator's Execute() method is invoked only if the operator is enabled.

Synchronization operators have more complicated Execute() methods than described above because of the potential for execution constraints. For example, environmental synchronization's Execute() method may have an input parameter that asserts that only transitions triggered by a particular sync event can execute in the current step. If no constraint is imposed by an operator higher in the composition hierarchy, then the environmental synchronization operator might assert its own execution constraint – that only transitions triggered by one of the operator's sync events can execute in the current step. Otherwise, the operator instructs one of its enabled operands to execute without imposing additional constraints. In all cases, the operator propagates constraints via recursive calls to the operands' Execute() methods, but uses information gathered by IsEnabled() to invoke only operands whose enabled transitions satisfy the constraints.

Other Composition Operators. Our CGG also supports rendezvous, interrupt, sequence, and choice composition operators. The Java classes generated

for these operators resemble that generated for the environmental synchronization operator, in that they introduce operator-specific member variables to keep track of operator-specific enabledness information, and their Execute() methods enforce execution constraints imposed by operators that are higher in the model's composition hierarchy. Note that in a model that uses multiple types of composition operators, it is not enough for each operator to keep track of just the enabledness information relevant to its own operation. Operator classes must have member variables for *all* types of enabledness information needed by *any* operator in the composition hierarchy, and all of their IsEnabled() methods must include parameters for these data. The details of how all supported composition operators are implemented can be found in [25].

4.3 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 template parameters). 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 to a single operator with multiple operands. A flatter 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 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 introduce default priorities. 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 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 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.

6 Evaluation

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

6.1 Utility

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 [19], a semantically configurable translator to SMV [20] that we have developed, to transform their models into representative SMV models, and then they 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. Many 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.

6.2 Correctness

To assess correctness, we designed a test suite of models that exercise different parts of the CGG. The test suite covers each implemented template-semantics parameter and each composition operator at least once, and covers all pairs of composition operators. For each test case, we manually inspected the sequence of snapshots produced by the generated program's execution and checked it against the sequence of snapshots comprising the model's "execution" of the same test.

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. This example also used some nonstandard template-parameter values (e.g., an enabling event persisted until

it triggered 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.

6.3 Efficiency

To determine the cost of semantically configurable code generation, we compared our generated code to that of three commercial tools, 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. The other models are small but typical software controllers for embedded systems: (1) An *Elevator* for a three floor building, whose components model the controller, service-request buttons, the engine, and the door and door timer. (2) A *Heater*, whose components model the controller, the furnace, and sensors in the room. (3) A hotel-room *Safebox*, whose components model a keypad for entering security codes, a display, and the status of the lock. 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 1, compare our generated code against that from two UML-based code generators: IBM's Rational Rose Realtime (Rose RT) [15] and Telelogic's Rhapsody (RH) [28]. The two tools support similar UML semantics: communication between machines is via message passing, all generated messages are sent to a single global queue[1], and only the event at the head of this queue can trigger transitions. One difference is that in Rose RT a message event triggers only one (compound) transition, whereas in RH an event can initiate a sequence of transitions.

Using CGG, we generated code generators that simulate the semantics of the Rose RT and RH code generators. We ran the code generators on all four models, and then measured the execution times of the generated programs. The results are reported in Table 1 are the average execution times over 10 runs, with each run consisting of 100000 iterations between the system and environment components in *Heater*, *Elevator*, and *Safebox*, and 500000 iterations between the Ping and Pong components in *PingPong*. All runs were performed on a 3.00Ghz Intel Pentium 4 CPU with 1GB of RAM, running Windows XP Professional Version 2002. 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 RH generated programs. The deterministic CGG version of *PingPong*

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

Table 1. UML Comparisons (seconds)

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

Table 2. Statecharts Comparisons (seconds)

Model	SS	CGG-Det
PingPongSS	1.3	1.6
ElevatorSS	16.6	14.3
HeaterSS	3.9	2.6
SafeboxSS	6.8	6.2

performed slightly better than the RH version. Similar comparisons with nondeterministic CGG generated programs 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 it to SmartState (SS) [1]. SS uses parallel composition and broadcasting of events. In addition, SS 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 2. On average, SS generated programs ran 1.2 times longer than deterministic CGG generated programs (CGG-Det). The SS 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 SS on the larger models, but not as well on the toy model *PingPong*. Our UML-based code generators are competitive with RH for models with fewer components (*PingPong*, *Heater*, *Safebox*), but less so on the larger model, *Elevator*. Rose RT significantly outperforms RH 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 enabledness checks and execution directives could be loosened.

7 Related Work

Most model-driven-engineering environments are centred on a single modelling notation that has a single semantics[6,28,15,27]. Configurability in such systems is geared more towards flexibility in the target language or platform [7,10] 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. 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. Admittedly, preprocessor directives are a rather primitive form of generative programming, and other generative technologies could be explored [3,4,5]. However, preprocessor technology is sufficiently powerful for our needs and is stable.

Our work is also loosely related to that of compiler generators, in which a compiler is constructed directly from a language's semantics [16]. Researchers have investigated how to generate compilers from denotational semantics [2,24], operational semantics and rewrite rules [12,13], natural semantics [8], and language algebras [17]. A key disadvantage of these approaches is that one has to write a complete semantics for the notation to be compiled (a feat sometimes worthy of a research publication). In contrast, in our approach, specifying the semantics of a modelling notation is reduced to providing a collection of relatively small semantics-parameter values. The trade-off is that our CGG creates code generators only for template-semantics expressible notations – although the CGG is structured to facilitate the adding of new template-parameter values.

Meta-modeling [23] 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 [22] and graph grammars [18], 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 [11,9], and to embed the semantics of those features in the implementation of the code generator.

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, such that the specifier is spared from having to provide a complete semantics definition. 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 [26]. 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 (2008), http://www.smartstatestudio.com
2. Appel, A.W.: Semantics-directed code generation. In: Proc. ACM Sym. on Prin. Prog. Lang (POPL 1985), pp. 315–324. ACM Press, New York (1985)
3. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Soft. Eng. Meth. 1(4), 355–398 (1992)
4. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall, Englewood Cliffs (2001)
5. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
6. Harel, D., et al.: STATEMATE: A working environment for the development of complex reactive systems. IEEE Trans. on Soft. Eng. 16(4), 403–414 (1990)
7. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from uml models. In: Proc. Intl. Work. on Soft. and Perf (WOSP 2005), pp. 75–86. ACM Press, New York (2005)
8. Diehl, S.: Natural semantics-directed generation of compilers and abstract machines. Form. Asps. of Comp. 12(2), 71–99 (2000)
9. G.S.S., et al.: Clearwater: extensible, flexible, modular code generation. In: Proc. IEEE/ACM Intl. Conf. on Aut. Soft. Eng(ASE 2005), pp. 144–153. ACM Press, New York (2005)
10. Floch, J.: Supporting evolution and maintenance by using a flexible automatic code generator. In: Proc. Intl. Conf. on Soft. Eng (ICSE 1995), pp. 211–219. ACM Press, New York (1995)
11. Grundy, J., et al.: Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In: Auto. Soft. Eng. (ASE), pp. 25–36 (2006)
12. Hannan, J.: Operational semantics-directed compilers and machine architectures. ACM Trans. Prog. Lang. Sys. 16(4), 1215–1247 (1994)
13. Hannan, J., Miller, D.: From operational semantics to abstract machines. Math. Struct. Comp. Sci. 2(4), 415–459 (1992)
14. Harel, D.: On the formal semantics of statecharts. Symp. on Logic in Comp. Sci., 54–64 (1987)
15. IBM Rational. Rational Rose RealTime v7.0.0 (2005), http://www.ibm.com/rational
16. Jones, N.: Semantics-Directed Compiler Generation. LNCS, vol. 94. Springer, Heidelberg (1980)
17. Knaack, J.L.: An algebraic approach to language translation. PhD thesis, University of Iowa (1995)
18. Königs, A., Schürr, A.: Tool integration with triple graph grammars – a survey. Elect. Notes in Theor. Comp. Sci. 148(1), 113–150 (2006)
19. Lu, Y., Atlee, J.M., Day, N.A., Niu, J.: Mapping template semantics to SMV. In: Proc. of Auto. Soft. Eng. (ASE 2004), pp. 320–325 (2004)
20. McMillan, K.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic Publishers, Dordrecht (1993)

21. Niu, J., Atlee, J.M., Day, N.A.: Template Semantics for Model-Based Notations. IEEE Trans. on Soft. Eng. 29(10), 866–882 (2003)
22. Object Management Group. Revised submission for MOF 2.0 Query/View/Transformation RFP, http://www.omg.org/docs/ad/05-03-02.pdf
23. Object Management Group. Meta Object Facility Core Specification, Formal/06-01-01 (2006)
24. Paulson, L.: A semantics-directed compiler generator. In: Proc. ACM Sym. on Prin. of Prog. Lang (POPL 1982), pp. 224–233. ACM Press, New York (1982)
25. Prout, A.: Parameterized Code Generation From Template Semantics. Master's thesis, School of Computer Science, University of Waterloo (2005)
26. Taleghani, A., Atlee, J.M.: Semantic variations among UML statemachines. In: ACM/IEEE Int. Conf. on Mod. Driven Eng. Lang. and Sys., pp. 245–259 (2006)
27. Telelogic. Telelogic TAU SDL Suite, http://www.telelogic.com/corp/products/tau/sdl/index.cfm
28. Telelogic. Rhapsody in J v7.1.1.0 (2007), http://modeling.telelogic.com/products/rhapsody/index.cfm
29. Yavuz-Kahveci, T., Bultan, T.: Specification, verification, and synthesis of concurrency control components. In: Proc. Intl. Symp. on Soft. Test. and Anal (ISSTA 2002), pp. 169–179. ACM Press, New York (2002)