Softw Syst Model (2012) 11:251–272
DOI 10.1007/s10270-010-0176-6

SPECIAL SECTION PAPER

Code generation for a family of executable modelling notations

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

Received: 30 June 2009 / Revised: 5 September 2010 / Accepted: 20 September 2010 / Published online: 27 October 2010

Abstract We are investigating *semantically configurable* model-driven engineering (MDE). The goal of this work is a modelling environment that supports flexible, configurable modelling notations, in which specifiers can configure the semantics of notations to suit their needs and yet still have access to the types of analysis tools and code generators normally associated with MDE. In this paper, we describe *semantically configurable code generation* for a family of behavioural modelling notations. The family includes variants of statecharts, process algebras, Petri Nets, and SDL88. The semantics of this family is defined using *template semantics*, which is a *parameterized* structured operational semantics in which parameters represent semantic variation points. A specific notation is derived by instantiating the family's template semantics with parameter values that specify semantic choices. We have developed a code-generator generator (CGG) that creates a suitable Java code generator for a subset of derivable modelling notations. Our prototype CGG supports 26 semantics parameters, 89 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.

Keywords Model-driven engineering · Code generation

Communicated by Prof. Krzysztof Czarnecki.

A. Prout · J. M. Atlee (✉) · N. A. Day · P. Shaker
David R. Cheriton School of Computer Science,
University of Waterloo, Waterloo, ON N2L 3G1, Canada
e-mail: jmatlee@uwaterloo.ca

A. Prout
e-mail: aprout@uwaterloo.ca

N. A. Day
e-mail: nday@uwaterloo.ca

P. Shaker
e-mail: p2shaker@uwaterloo.ca

1 Introduction

A significant obstacle to successful model-driven engineering (MDE) is the semantic mismatch between the problem or system being modelled and the chosen modelling notation. A primary reason for this mismatch is that many modellers restrict themselves to notations that have tool support (i.e., model editors, simulators, analyzers, and code generators). However, only a small fraction of notations—or, more specifically, a small fraction of the semantic variants of notations—are supported by substantial tool suites.

There is ample anecdotal evidence to suggest that modellers want to be able to use a wider set of notations and semantics. Consider the many variants of statecharts [62], some of which were invented precisely because neither the original variants [23,24] nor the tool-supported variants [12,25] suited the modellers' needs. As another prominent data point, consider the semantics of the United Modeling Language (UML) [50,59], which includes several semantic variation points because the Object Management Group (OMG) members who define and maintain the UML specification could not agree on a single semantics. As additional evidence, the authors have witnessed cases where notations are tweaked in ad hoc ways to ease the modelling of a particular problem. We have also witnessed graduate-level formal-methods students who, when given the flexibility to do so, combine the features and semantics of multiple modelling notations rather than express their modelling problems in some (single) standard notation. For example, they will

create models that have a statecharts execution semantics but that employ rendezvous for some critical synchronization, or they will create models with CCS semantics augmented with global shared variables.

In this paper, we focus on semantic variability within the family of behavioural modelling notations. Most obviously, this family includes statecharts and its variants, but more generally it encompasses any notation whose semantics can be expressed operationally as a set of execution traces. Example notations include variants of process algebras, variants of dataflow languages, Petri Nets, SDL, and so on. At a very high level, these notations have a common semantics: they all have notions of specified transitions, enabled transitions, executing transitions, effects of transitions. But there are subtle variations in the semantics of these shared concepts:

- the choice of event(s) to trigger the next execution step
- the duration of processed and unprocessed events
- the choice of variable values when evaluating expressions
- the choice of default priorities among transitions
- the effects of environmental inputs (i.e., input events, sensor values) on the execution state
- the effects of an executing transition on the execution state

We are interested in developing modelling tools and environments that support these types of semantic variations.

The intended use case is the organization that works on multiple projects whose modelling needs are different. We have found that, in our own experience, different semantic choices are appropriate for different modelling problems. The fact that UML StateMachines have semantic variation points confirms the need to support variability. If an organization finds itself using the same variant of modelling notation and semantics on a number of its projects, then it should invest time and effort in creating development tools that are optimized for that language [3,36,46,58]. But if an organization uses multiple notation variants, then it would be better if its infrastructure team could maintain a *product line* of modelling tools whose semantics variations can be configured and extended.

With this use case in mind, we propose *semantically configurable* modelling tools and environments that support a *family of related notations*. We use *template semantics* [47,48] to define a family of modelling notations. A template semantics is effectively a *parameterized* structured operational semantics [53]. The *templates* are parameterized semantics of concepts that are common among notations in the family (e.g., set of enabled transitions). *Parameters* within a template represent semantic variation points (e.g., how the set of enabled transitions is determined), and *parameter values* define semantic variants. For example, notations differ with respect to which events can trigger the next execution step (e.g., the single event at the head of an event queue, or any event generated in the last execution step, or any event that has not yet been processed). Given a template-semantics definition for a family of notations, the semantics for a distinct notation within this family can be succinctly specified by instantiating the templates' parameters with specific parameter values.

In previous work [48,49], we defined the template semantics for a family of behavioural modelling notations. The family comprises notations whose semantics can be expressed operationally as sequences of execution steps. The family includes process algebras (e.g., CCS [44], CSP [28], basic LOTOS [32]), statecharts variants (e.g., statecharts [23,24], STATEMATE [12], RSML [38], UML StateMachines [50]), and even more sophisticated notations like SCR [27], SDL88 [33], and BoxTalk [66]. We have started to develop semantically configurable tools that support this family of notations. Metro [47] is a suite of semantically parameterized analysis tools (e.g., model checkers) that can be configured with arbitrary parameter values. Express [39] is a semantically configurable translator that converts models into the input language of the Symbolic Model Verifier (SMV) [43]; Express is configurable via a fixed menu of possible parameter values.

In this paper, we provide an extended description of a prototype of a semantically configurable code-generator generator (CGG), which was first presented in [55]. The CGG takes as input a description of a modelling notation's semantics expressed as a set of template parameter values, selected from fixed menus of semantics choices; it produces a code generator for models expressed in that notation. Our current prototype CGG supports 26 semantics parameters, 89 parameter values, and 7 composition operators (e.g., interleaving, parallel composition, rendezvous, choice, sequential composition)—including new template parameters and parameter values to support communication queues between model components.

Given the number of semantics parameters and combinations of choices, we would not expect the average modeller to configure her modelling environment. For one thing, not all combinations of parameter values result in a consistent semantics definition (this issue is discussed in Sect. 7.2). The individual who is making semantics decisions must understand the consequences of the decisions and their interdependencies. This level of understanding is comparable to that needed to define the semantics of a new domain-specific language (DSL)—which is the expected input to most modelling-tool frameworks [3,36,46,58]. What makes our approach unique is that we have simplified the configuration task to one of selecting from menus of semantic options, rather than writing the semantics from scratch. A consequence of this decision is that the set of code generators that can be produced by the CGG is fixed, but we have structured

the CGG so that it is easily extensible to support new semantic options.

Problems that we addressed in the course of this work include

- *Configurable CGG* A primary contribution of this work is the design of a CGG as a parameterized software product line that encapsulates semantic variation points. This work is unique in that (1) the type of variability focuses on execution semantics, (2) the semantics variability is fine grained, and (3) a wide variety of semantics choices are supported. Moreover, the design of the CGG is structured to ease the task of incorporating new semantic options. In contrast, other approaches to configurable model-based code generation are parameterized with respect to modelling-language constructs [58]; object types [61]; the target language or platform [13,18]; optimizations in the generated code [40]; or a small set of coarse-grained semantic variations, such as parallel versus interleaving semantics [30], synchronous versus asynchronous message passing [57], or default priority schemes [64].
- *Configurable execution semantics* A secondary contribution is the design of the run-time architecture of the generated Java programs. The generated programs have a common skeletal algorithm that simulates the execution of an input model in terms of abstract execution steps. This simulation algorithm is parameterized by template-semantics parameters and is specialized by composition operators (similar to the execution step of SMV models generated by Express [39]). This run-time architecture facilitates semantic configurability.
- *Diverse composition operators in Java* A side effect of the above contribution is a means for implementing a variety of composition operators in Java—and for accommodating models that have heterogeneous composition operations. 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. Support for these operations requires explicit synchronization of components.
- *Resolving nondeterminism* There are several natural sources of nondeterminism in models (e.g., selecting one of many enabled transitions to execute). Although it may be appropriate to leave such nondeterminism unresolved during the modelling phase, nondeterminism in source code is unnatural. Our code generators employ a number of strategies for eliminating nondeterminism.
- *Configurability evaluation* Using von der Beeck's comparison of statecharts variants as a coverage metric, we assess the degree to which the CGG family of notations covers von der Beeck's set of semantic variabilities.
- *Performance evaluation* A long-term concern of this work is how efficient CGG-generated code is, given the competing concern that a configurable CGG 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 [29], IBM Rational Rhapsody [30], and SmartState [1]—each of which has been optimized for a particular modelling notation.

The rest of this paper is organized as follows. In Sect. 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 implemented in our prototype. In Sects. 3 and 4, we describe our CGG and the architecture of the generated code, respectively. We discuss techniques for resolving nondeterminism in Sect. 5. We present the results of our variability assessment and performance evaluations in Sect. 6, and discuss some of the advantages, limitations, and trade-offs of semantically configurable tools in Sect. 7. We conclude the paper with discussions on related and future work.

2 Semantically configurable notations

In this section, we describe the syntax and semantics of our semantically configurable modelling language. The intended scope of configurability is the family of executable behavioural modelling notations—that is, those notations whose semantics can be expressed operationally as sequences of execution steps. We achieve semantic configurability by using template semantics [48,49] to define the family of notations. The semantics are configurable via a set of user-provided parameter values. Thus, modellers create models using the syntax of our notation, and they configure the semantics of their models by selecting template-semantics parameter values. As will be seen, the CGG implements a fixed set of parameter values, so it supports only a subset of the intended family of notations.

2.1 Syntax

We base the syntax of our modelling notation on a form of extended finite-state machine that we call *hierarchical transition systems (HTS)*, which are adapted from statecharts [23]. An example HTS is shown in Fig. 1. It includes control states and state hierarchy, state transitions, events, and typed variables. An HTS designates a default initial state for each hierarchical state and initial values of all the variables (not shown).

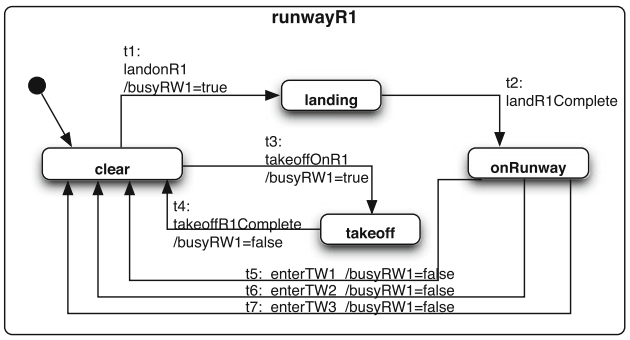
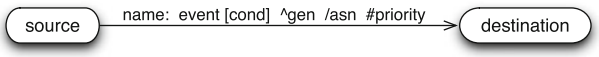


Fig. 1 Example HTS

Transitions have the following form:



Each transition has a *source* state, is triggered by zero or more *event*s, may have a guard *cond*ition (a predicate on variable values), and may have an explicit *priority*. If a transition executes, it leads to a *destination* state, may generate events (specified by zero or more ^*gen* actions), and may assign new values to variables (specified by zero or more */asn* actions). For example, the HTS in Fig. 1 models the states of an airport runway. The transition *t1* from state *clear* to state *landing* is triggered by a request to land on the runway (event *landonR1*) and has an action of assigning variable *busyRW1* to true. To ease discussion of HTS models, we annotate each transition with a *name* (e.g., *t1*) that we use to reference the transition.

HTSs do not model concurrent behaviour. In UML terminology, an HTS corresponds to a simple-composite state with only one region. Concurrency is achieved by composing multiple HTSs.

2.2 Semantic domain

A notation's semantics is expressed as a transition relation over execution states, which is a compact representation of the possible executions of a model.

More formally, the semantic domain of HTS execution is *sequences of snapshots*, where a *snapshot* records information about a model's execution at a discrete point in the execution. Snapshot information is stored in distinct elements:

CS the *configuration* of current states
IE the events to be processed
AV the *variable valuation*
I_a data about inputs to the HTS
O the generated events (to be communicated to other HTSs)
Q the contents of communication queues between HTSs

In addition, the snapshot includes auxiliary elements that, for different notations, store different types of information about an 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 previous values
Q_a data about queues, such as the chosen input queue

The expression $ss.CS$ refers to element CS in snapshot ss.

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

2.3 Template semantics

We use template semantics [48], a parameterized structured operational semantics [53], to specify how an execution step (in a model) affects the execution's snapshot. A template-semantics definition is expressed as a set of functions and relations over a model's elements (e.g., input events, transitions) and its snapshot elements. Parameterized definitions, called *templates*, define the most generic aspects of a model's execution. Our definition of an execution step of an HTS comprises five templates:

1. sensing inputs from the environment
2. determining the set of enabled transitions
3. executing a transition
4. executing a micro-step (which uses the previous two templates)
5. executing a macro-step

Each template definition is heavily parameterized by semantic variation points. Consider template ENABLED_TRANS, which defines the set of enabled transitions:

$$\begin{aligned}&\text{ENABLED\_TRANS}(ss, T)\\ &\equiv \{\tau \in T \mid \textbf{en\_state}(ss, \tau) \wedge \textbf{en\_event}(ss, \tau)\\ &\qquad \wedge \textbf{en\_cond}(ss, \tau)\}\end{aligned}$$

Predicates **en_state, en_event,** and **en_cond** are template parameters that specify how the state-, event-, and variable-related snapshot elements, respectively, are evaluated to determine whether a transition τ is enabled in snapshot ss. Parameter values are discussed in the next subsection.

As another example, the template definition APPLY_TRANS applies the effects of an executing transition τ to a snapshot

Construct	Start of Macro-step	Micro-step
states	$\mathbf{reset\_CS}(ss, I, CS')$ $\mathbf{reset\_CS}_a(ss, I, CS'_a)$ $\mathbf{en\_state}(ss, \tau)$	$\mathbf{next\_CS}(ss, \tau, CS')$ $\mathbf{next\_CS}_a(ss, \tau, CS'_a)$
events	$\mathbf{reset\_IE}(ss, I, IE')$ $\mathbf{reset\_IE}_a(ss, I, IE'_a)$ $\mathbf{reset\_I}_a(ss, I, I'_a)$ $\mathbf{en\_event}(ss, \tau)$	$\mathbf{next\_IE}(ss, \tau, IE')$ $\mathbf{next\_IE}_a(ss, \tau, IE'_a)$ $\mathbf{next\_I}_a(ss, \tau, I'_a)$
variables	$\mathbf{reset\_AV}(ss, I, AV')$ $\mathbf{reset\_AV}_a(ss, I, AV'_a)$ $\mathbf{en\_cond}(ss, \tau)$	$\mathbf{next\_AV}(ss, \tau, AV')$ $\mathbf{next\_AV}_a(ss, \tau, AV'_a)$
outputs	$\mathbf{reset\_O}(ss, I, O')$	$\mathbf{next\_O}(ss, \tau, O')$
communication queues	$\mathbf{reset\_Q}(ss, I, Q')$ $\mathbf{reset\_Q}_a(ss, I, Q'_a)$	$\mathbf{next\_Q}(ss, \tau, Q')$ $\mathbf{next\_Q}_a(ss, \tau, Q'_a)$
additional parameters	$\mathbf{macro\_semantics}$ $\mathbf{pri}(T) : 2^T$ $\mathbf{resolve}$	

Fig. 2 The list of template parameters that specialize the template-semantics definitions. Argument ss refers to the current snapshot of a model's execution, I refers to system inputs, τ refers to an executing transition, and primed arguments (e.g., CS') refer to updated snapshot elements (e.g., updated due to sensed inputs or the execution of a transition). Argument T refers to the set of transitions specified in the model

ss, to derive the resultant next snapshot ss'. In the definition, unprimed elements refer to snapshot values before the transition τ executes and primed elements refer to snapshot values after the transition executes.

$$
\begin{aligned}
&\text{APPLY\_TRANS}(ss, \tau, ss') \\
&\quad \equiv \text{let } \langle CS', IE', AV', O', CS'_a, IE'_a, AV'_a, I'_a \rangle \\
&\quad \equiv ss' \text{ in } \mathbf{next\_CS}(ss, \tau, CS') \wedge \mathbf{next\_CS}_a(ss, \tau, CS'_a) \\
&\quad \wedge \mathbf{next\_IE}(ss, \tau, IE') \wedge \mathbf{next\_IE}_a(ss, \tau, IE'_a) \\
&\quad \wedge \mathbf{next\_AV}(ss, \tau, AV') \wedge \mathbf{next\_AV}_a(ss, \tau, AV'_a) \\
&\quad \wedge \mathbf{next\_Q}(ss, \tau, Q') \wedge \mathbf{next\_Q}_a(ss, \tau, Q'_a) \\
&\quad \wedge \mathbf{next\_O}(ss, \tau, O') \wedge \mathbf{next\_I}_a(ss, \tau, I'_a)
\end{aligned}
$$

The template uses parameters **next_X**, one for each snapshot element $ss.X$, 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 (specified by **next_CS**); it might also update variable values (specified by **next_AV**), update the set of enabling events (specified by **next_IE**), and so on.

In this manner, the templates are effectively logic expressions over the template parameters, with the details of a notation's semantics being defined by the choice of template parameter values.

2.4 Template parameters

Our template definitions have a total of 26 parameters that represent variations on how a model's snapshot can change during execution. The parameters are listed in Fig. 2, organized by language construct. For example, the seven event-related parameters work together to determine which events can enable transitions and which events remain to-be-processed.

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

- **reset_IE** Empty the set of events (in snapshot element $ss.IE$) that were generated in the previous macro-step.
- **reset_I$_a$** Add inputs I to the events already in snapshot element I_a.
- **reset_CS** Make no change to the set of current states CS.

Another ten parameters are of the form **next_X**(ss, τ, X'), each specifying how a snapshot element, $ss.X$, is updated due to the effects of an executing transition τ. Example values for some of these parameters include

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

There are three ENABLED_TRANS parameters that specify how the snapshot elements are evaluated to identify enabled transitions. Specifically, the parameters **en_cond**, **en_event**, and **en_state** determine if a transition is enabled based on its guard condition, triggering events, and source state, respectively. Template parameter **pri** specifies a default priority scheme on transitions (e.g., transitions whose source states are higher in the model's state hierarchy might have priority over transitions whose source states are lower in the state hierarchy). Parameter **macro_semantics** specifies when new inputs are sensed from the environment (e.g., after every micro step, or when no more transitions are enabled). Parameter **resolve** specifies how to resolve concurrent assignments to shared variables.

To define the semantics of a particular modelling notation, the user specifies a parameter value for each of these 26 template parameters. Example parameter values for just the event-related template parameters are listed in Table 1. However, not every combination of parameter values results in a meaningful semantics definition. Parameters that are associated with the same language construct tend to have inter-dependent values. To give an obvious example, if it is important to process events in a specific order, then the snapshot element that maintains events ($ss.IE$) might be structured as a queue, in which case the operations over that snapshot element (**reset_IE, next_IE, en_event**) should be expressed in terms of enqueue, dequeue and

Table 1 Sample event-related template parameter values

Parameter	Parameter value	Informal definition
reset_IE(ss, I, IE')	$IE' = \emptyset$	Make IE empty
	$IE' = ss.IE$	Make no change to IE
	$IE' = I$	Assign IE to be the inputs I from the environment
	$IE' = ss.IE \cup I$	Add inputs I to the events already in set IE
	$IE' = ss.IE \frown I$	Append inputs I to end of the queue IE of events
next_IE(ss, τ, IE')	$IE' = gen(\tau)$	Assign 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_IE$_a$ (ss, I, IE'_a)	$IE'_a = \emptyset$	Make IEa empty
	$IE'_a = \{head(ss.I_a)\}$	Assign IEa to be the head element in event queue Ia
	$\exists q \in ss.Q \cdot [IE'_a = \{head(q)\}]$	Assign IEa to be the head element of an arbitrary input queue in Q's set of queues
next_IE$_a$ (ss, τ, IE'_a)	$IE'_a = \emptyset$	Make IEa 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
	$IE'_a = ss.IE_a \frown gen(\tau)$	Append the events generated by τ to the end of queue IEa
reset_I$_a$ (ss, I, I'_a)	$I'_a = \emptyset$	Make Ia empty
	$I'_a = I$	Assign 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
next_I$_a$ (ss, τ, I'_a)	$I'_a = \emptyset$	Make Ia 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
reset_Q(ss, I, Q')	$\forall q \in ss.Q \cdot [if\ ss'.IE_a = head(q)$ $then\ q' = tail(q) \frown I$ $else\ q' = q \frown I\ ]$	Remove selected event IE_a′ from the appropriate q and append input events I to the end of all queues inQ
	$\forall q \in ss.Q \cdot [q' = q \frown directed(q, I)\ ]$	Append to each q in Q the subset of inputs that are directed to q
next_Q(ss, I, Q')	$\forall q \in ss.Q \cdot [if\ head(q) \in trig(\tau)$ $then\ q' = tail(q) \frown directed(q, gen(\tau))$ $else\ q' = q \frown directed(q, gen(\tau))\ ]$	Remove τ's triggering event from the appropriate queue, and append to each q in Q the subset of generated events directed to q
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 event at the head of queue IE

view-head-of-queue operations. To give a more subtle example, the states, events, or variables used to determine whether a transition is enabled may be different from the snapshot's current states, events, and variable values: for example, the set of enabling states may be a subset (or empty set) of the current states, as a means of constraining the set of enabled transitions; a single event may be chosen from the pool of events at the start of a macro-step and be the only enabling event throughout the macro-step; transition guards may be evaluated with respect to variable values that held at the start of the macro-step rather than values assigned during the macro-step. To realize these semantic choices, auxiliary snapshot elements are needed to record enabling information (distinct from the snapshot elements that record the model's executable state); the template parameter values need to keep both types of snapshot elements up-to-date and in sync as the

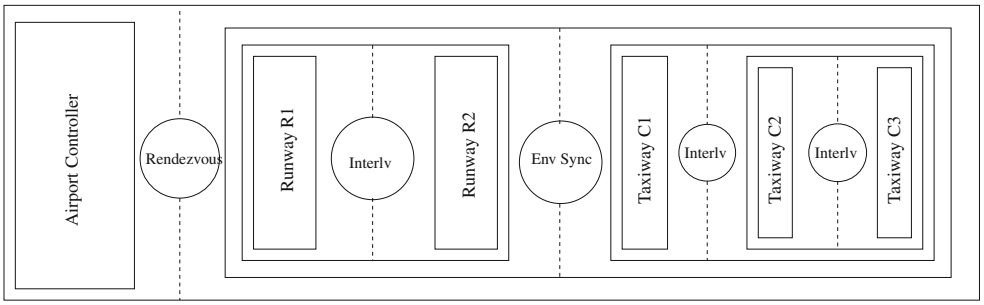


Fig. 3 Compositional hierarchy for a ground-traffic control system

execution proceeds; and the template parameters (**en_state, en_event, en_cond**) need to be specified over the enabling snapshot values.

Template parameter values must be specified or selected by someone who understands the semantics choices and dependencies—such as a local MDE infrastructure expert—to ensure that the parameter values are mutually supportive and result in a consistent and desired modelling-notation semantics. This limitation is discussed more thoroughly in Sect. 7.2.

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.

The operations are defined as additional templates, and they are parameterized with the same template parameters as described above. More formally, each operation takes two operands, each of which is either an HTS or a collection of previously composed HTSs, and specifies how the operands execute together. The result of a nested composition of HTSs is a binary tree, whose internal nodes are all composition operators and whose leaf nodes are all HTSs. We use the term *composition hierarchy* to refer to a model's composition structure (see the shaded modules in Fig. 4).

Our prototype CGG supports variations of seven composition operators:

- *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.
- *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.
- *interrupt* Control passes between the operands via a set of interrupt transitions.
- *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 that rendezvous 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.

Figure 3 shows the composition hierarchy for a Ground-Traffic Control System [65] 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 (Fig. 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 all of the roadways to ensure that they have a shared understanding of the status of the roadways.

3 Configurable CGG

We use template semantics as the basis for structuring a prototype CGG as a software product line for a family of

code generators. Each code generator in the family transforms models, written in a specific behavioural modelling notation, into representative Java programs. The core of the CGG product line comprises code that is common to all of code generators in the family. This core includes the parsing of an input model into an internal representation of the model's syntactic structure (e.g., state hierarchies of HTSs, transitions in HTSs, compositions of components); the definition and initialization of the model's execution state (i.e., a class definition for each snapshot element); and the generation of a skeletal algorithm that simulates an input model. The skeletal part of the simulation algorithm reflects the parameterized execution semantics that is common to all notations in our modelling-language family. Specifically, it simulates abstract execution steps of an input model. Like the template-semantics definitions described in the previous section, the algorithm is parameterized with semantic choices. The simulation algorithm is described in more detail in Sect. 4.

The features of the CGG product line are the semantic choices of the notation family. Thus, they correspond to the template parameter values discussed in the previous section. They are implemented as subroutines that flesh out the skeletal simulation algorithm—in an analogous way that primitive operations complete the implementation of a template method in the Template Method design pattern [19].

To produce a code generator for a particular behavioural modelling notation, the user instantiates the CGG product line by specifying parameter values for each of the 26 template-semantics parameters. Our CGG prototype supports a fixed collection of parameter values, so this task reduces to selecting from among menus of supported values. The output of the CGG is the core code that is shared among all code generators and a set of relevant features (subroutines) that implement the selected semantic choices.

Our prototype CGG is implemented using preprocessor directives and conditional compilation, as a primitive form of generative programming. The user provides a file that lists a preprocessor #define declaration for each template parameter, specifying the value of that parameter. The CGG source code is annotated with preprocessor directives that indicate the parts of the source that are specific to each supported parameter value—namely, the routine definition that corresponds to the parameter value. 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. Generative-programming technologies other than conditional compilation could have been explored [4,8,11,34]. However, preprocessor directives are sufficiently powerful and their technology is stable.

Our prototype CGG supports 89 parameter values, roughly 2–11 values per parameter. We have not attempted to identify or implement a complete 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.

What is important is that we have structured the CGG to ease the task of extending it to support new semantic options for the given semantic variation points. Each new semantic option entails (1) specifying a new parameter value (name or expression), (2) implementing the execution semantics of the new option as a subroutine, and (3) encapsulating the new subroutine implementation within a conditional preprocessor directive (whose condition is the newly defined parameter value).

4 Generated Java code

Our CGG generates code generators that take as input a model and produce as output a Java program that simulates the input model. The resultant Java program is effectively an interpreter that maintains an internal representation of the input model and its current execution state, and that derives the next execution state of the model by simulating an execution step. The runtime structure of the Java program resembles the composition hierachy of the input model. Consider the object model of the program generated from our model of the Ground-Traffic Control System (from Fig. 3), shown in Fig. 4. 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, Q, Q_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 Fig. 4), 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.

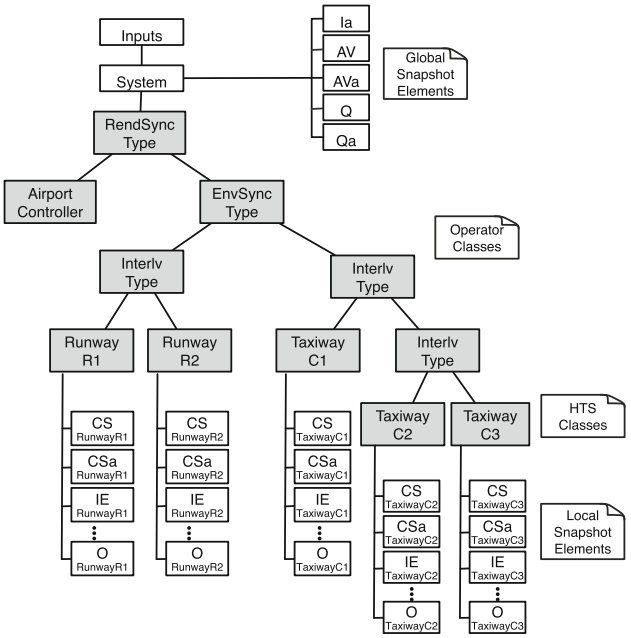


Fig. 4 Code structure for the ground-traffic control system example. *Shaded objects* mimic the composition hierarchy of the model from Fig. 3

In the second phase, execution decisions in the form of constraints flow from the System object down the composition hierarchy to the HTSs: every 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 input model, which is useful for simulation and reasoning about all possible executions. In Sect. 5, we discuss ways of resolving nondeterminism, to produce deterministic code from nondeterministic models. We conclude this section with a discussion of how we optimize the generated code.

4.1 HTS: enabled transitions

A separate class is generated for each HTS in the input model. Figure 5 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 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 (shown on the left in Fig. 5) that identifies the enabled transitions. Much of this task is done by methods that implement the semantic parameters **en_state**, **en_event**, and **en_cond** (line 4). 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).

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 Sect. 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 [54].

A Java class is generated for each operator type used in the input 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.

Fig. 5 Pseudocode for taxiway HTS. Configurable code is shown in SMALL CAPS

```
VAR Set enabledTrans
VAR Map syncTrans, rendTrans
VAR Transition exec
VAR EventSnap &Ia = globalIa
VAR VarSnap &AV = globalAV
VAR VarSnap &AVa = globalAVa
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_EVENT(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.find(rendEv)
4: else if syncEv is not null then
5:    exec = syncTrans.find(syncEv)
6: else
7:    exec = priority(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: Q.NEXT(exec)
18: Qa.NEXT(exec)
19: O.NEXT(exec)
```

Fig. 6 Pseudocode for interleaving composition

```
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
```

4.2.1 Interleaving and parallel composition

The IsEnabled() method for the interleaving operator is shown on the left in Fig. 6. 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 HTSs have more intricate implementations. Figure 7 presents the pseudocode for the Java class that implements environmental-synchronization composition. Associated with each instance of this operator is a set of synchronization events (variable *syncEvents*). All enabled transitions in component HTSs that are triggered by the same synchronization event execute simultaneously. Member variables *(L/R)Enabled* record whether the left or right operands, respectively, have enabled transitions. Member variables *(L/R)SynEv* record the synchronization event(s) that trigger the currently enabled transitions in the left and right operands, respectively.

The IsEnabled() method collects information about its operands' enabledness (lines 2–3), and then computes its own enabledness. The 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* synchronization event (line 4). The operator passes back to its parent a flag indicating whether it is enabled and the set of synchronization events that enable its components' transitions.[1]

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

[1] Note that *any* synchronization event that enables any of the operands' transitions is passed to the parent node (lines 1, 7), not just those identified in line 5, because other operators in the composition hierarchy may be interested in these other events.

```
VAR set syncEvents
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)≠ ∅
6: enabled = un-sync ∨ sync
7: syncEv = LSyncEv ∪ RSyncEv

1: Execute(event constraint)
2: {Case 1: enforce synchronization imposed by ancestor}
3: if constraint 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 ≠ ∅ ∧ 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. 7 Pseudocode for environmental synchronization

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 Fig. 6. If both of an 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 (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 typically have more complicated Execute() methods because they sometimes impose constraints on which enabled transitions should execute, as a means of synchronizing components. 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 synchronization event. If the imposed constraint involves one of the operator's own *syncEvents*, 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 as a parameter a constraint asserting that the selected transition(s) be triggered by a particular synchronization event, which means that the operator is free to impose its own constraint.

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

a. *Sync* If both operands have transitions that are triggered by one of the operator's *syncEvents*, then the operator may choose to synchronize its operands' executions (lines 17–19), asserting a new event constraint (lines 17, 23–26).
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 the *Sync* 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 [54].

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 the rendezvous events that are generated by enabled transitions as well as the rendezvous events that would enable 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. In fact, it is only the code that enforces the constraints passed by parameter that needs to be changed to accommodate heterogeneous composition operators; the rest of an operator's Execute() method remains unchanged. 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.

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 has an Execute() method (shown on the right in Fig. 5) 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 and assigned to variable *exec*. Constraints on which transition is selected are given as parameters (line 1) and are enforced by the method (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 10-19).

4.6 Optimizations

The CGG employs a number of simple optimizations to improve the performance of the generated Java code. Some optimizations are model independent and are incorporated into every generated code generator. For example, if a notation's semantics does not make use of all the snapshot elements, then the Java classes for unused snapshot elements are not generated.

Other optimizations are based on the structure of the input model. For example, the IsEnabled() method that is generated for each HTS component is more efficient than presented

above: 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. As another example, some computations performed by HTS or composition-operator modules can be statically computed or optimized, such as the determination of which HTS states are entered and exited when a transition executes. As a third 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.

To generate a deterministic program, we resolve the model's natural nondeterminism using priorities (e.g., prioritizing among multiple enabled transitions within an HTS). If the priorities provided by the specifier—in the form of explicit priorities on transition labels, on synchronization events, or 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 (e.g., starvation of a frequently enabled HTS or enabled transition), 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 simultaneously occurring synchronization or rendezvous 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-synchronized 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 a component can synchronize either by sending or receiving a rendezvous 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 evaluated two aspects of our approach: the degree of semantic variability that is supported by CGG and the efficiency of the generated code.

6.1 Semantic variability

In this section, we examine the degree of semantic variability in CGG by comparing its semantic choices with the list of modelling features and semantic variations given in von der Beeck's "A Comparison of Statecharts Variants" [62].

Table 2 lists the semantic variations: the leftmost column lists variation points as they are numbered in the von der Beeck paper,[3] and the second column lists choices for each variation point. The variations include how long an event can persist and continue to enable transitions, how simultaneously enabled transitions are prioritized, whether multiple transitions can execute in response to the same input event, whether a number of advanced modelling features are supported (e.g., negated events, in(state) references, history, timing), and the semantic consequences of those features (e.g., global consistency, compositionality). For example, if negated trigger events are supported, it raises a question of macro-step consistency: if a transition that is triggered by a negated event $\sim e$ is followed in the same macro-step by another transition that produces that event e as an action, then is the second transition disallowed because its actions would contradict the first transition's enabling conditions (*global consistency*) or is it allowed because each transition is enabled in its respective execution state (*local consistency*)?

[3] The one variation that is missing from the table is (8): operational versus denotational semantics, which is not really a semantics variation.

Table 2 Coverage of semantic variations in CGG (grey rows highlight non-coverage, and light-grey rows indicate semantics can be added)

Notation Feature	Semantic Variations	Supportable Variation	Implemented in CGG
(1) Synchrony hypothesis	Yes	value of **macro_semantics**	YES
	No	another value of **macro_semantics**	YES
(2) Causal transitions	self-triggering transitions		excluded from family
	causal transitions	templates for MICRO-STEP, MACRO-STEP	YES
(3) Negated trigger event		value of **en_events**	specified in [49]
(4) Negated trigger event that occurs in same macro-step	local consistency	value of **en_events**	specified in [49]
	global consistency	value of **en_events**	specified in [49]
(5) Interlevel transitions		definition of transition	YES
(6) State references	in(state) conditions		
(7) Compositional semantics	Self-start, self-terminate transitions		
(9) Instantaneous states	Yes	value of **en_states**	YES
	No	value of **en_states**	YES
(10) Durability of Events	single micro-step	value of **next_IE**	YES
	single macro-step	value of **next_IE**	YES
	until processed	values of **reset_IE, next_IE**	YES
	durability defined in trigger expression		
(11–14) Multiple transitions per macro-step	one transition per HTS	values of **next_CS**$_a$, **en_states**	YES
	each state entered at most once	values of **next_CS**$_a$, **en_states**	specified in [48]
	no limits; infinite sequence possible	values of **next_CS**$_a$, **en_states**	YES
(15) Determinism	deterministic models only	value of **pri**, parallel composition only	YES
	models may be nondeterministic	templates are relations	YES
(16) Priorities on transition execution	none	value of **pri**	YES
	explicit priority	value of **pri**	YES
	source state with highest rank	value of **pri**	YES
	source state with lowest rank	value of **pri**	YES
	transition with highest scope	value of **pri**	specified in [48]
	transition with lowest scope	value of **pri**	specified in [48]
(17) Preemptive interrupt	only interrupting transition executes	value of **pri**	YES
	interrupted xor interrupting transition	value of **pri**	YES
	interrupted and interrupting transitions		
(18) Internal vs. external events	distinguished	values of **next_IE, next_I**$_a$, **en_events**	YES
	combined	values of **next_IE, next_I**$_a$, **en_events**	YES
(19) Timing	timeout since last event e timed transition		
	alarm clocks	extension to templates, new parameters	specified in [48]
History pseudostate	shallow history	value of **next_CS**, **next_CS**$_a$	specified in [48]
	deep history	value of **next_CS**, **next_CS**$_a$	specified in [48]

The semantic choices that have been implemented in CGG—which is over half of the choices listed in the table—are those listed in white rows in Table 2; for redundancy, these rows have value "YES" in the last column. In each of these rows, the values in the third column indicates how the semantics choice is supported in CGG in terms of the relevant template definitions and parameters. With respect to the unsupported choices, we distinguish between those that belong to our intended family of notations and those that do not. The former choices, while not implemented in CGG, have been expressed as template parameter values in previous papers on template semantics. Thus, they could be added to CGG as new menu choices for existing template parameters, without changing other parts of CGG's implementation. These implementable semantic choices are highlighted in lighter-grey rows in Table 2; the values in last column of these rows cite references where the interested reader can find more information. The semantic choices that appear in darker-grey rows have not been included in the family of notations to be supported; the last column of these rows is empty. One of these choices (self-triggering transitions) is explicitly excluded from the family.

In summary, 21 of the 37 semantic variations listed in von der Beeck are currently implemented in CGG, and another 9 are considered implementable.

6.2 Efficiency

By employing an architecture that supports semantic configurability, we incur a penalty in the performance of our generated code. In this section, we report on that penalty by comparing the performance of our generated code to that of three commercial tools: IBM's Rational Rose Realtime (Rose RT) [29], IBM Rational Rhapsody (Rhapsody) [30], and SmartState [1]. We re-expressed the semantics of each of the commercial tools' modelling language in terms of CGG parameter values 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.
- A room *Heater*, whose components model the controller, the furnace, and sensors in a room whose temperature is being controlled.
- A hotel-room *Safebox*, whose components model a keypad for entering security codes, a display, and the status of the lock.
- An *Elevator* for a three-story building, whose components model the controller, service-request buttons, the

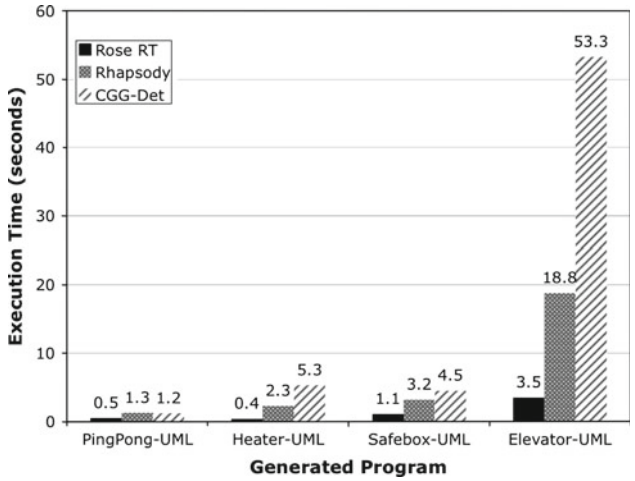


Fig. 8 Comparison of UML tools

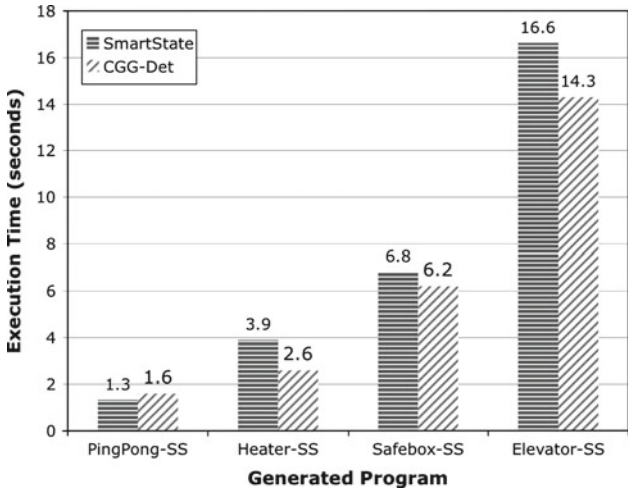


Fig. 9 Comparison of statecharts tools

engine, and the door and door timer. This model is the largest of the four, comprising 11 components.

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. 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 Fig. 8, compare our generated code against that generated by Rose RT and Rhapsody. These two tools support the UML and have similar semantics: communication between components is via message passing, all generated messages are sent to a single global queue[4] 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; this difference is not manifested in the models in our study.

Using CGG, we generated code generators that simulate the semantics of Rose RT and Rhapsody. We ran all four code generators on all four models, and then measured the execution times of the generated programs. The results reported in Fig. 8 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*. All runs were performed on a 3.00 GHz 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 Fig. 9. 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*,

[4] Both Rose RT and Rhapsody also allow multiple event queues and allow the specifier to indicate which components share which event queues. CGG does not currently support this option, but it could via new parameter values for queue-related template parameters [59].

Safebox), but less so on the larger model, *Elevator*. Rose RT significantly outperforms Rhapsody and CGG-Det on all models. We theorize that much of the performance gap can be attributed to the hierarchical architecture of our generated programs; this issue is discussed further in Sect. 7.1.

In addition to the above case studies, we generated code for the Ground-Traffic Control System described in Sect. 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.

7 Discussion

Our approach to semantically configurable code generation is distinguished in that (1) a notation's semantics is *user configured* from a fixed set of semantics choices, rather than *user defined*, (2) the semantic choices are fine grained, and (3) the approach accommodates models that have heterogeneous composition operators. In this section, we discuss some of the advantages, disadvantages, and trade-offs of our approach.

7.1 Performance

The most significant limitation of our work is its inefficiency. We were pleasantly surprised by how well our generated code performed against comparable programs generated by Rhapsody and SmartState. But our generated programs run almost an order of magnitude slower than comparable programs generated by Rose RT.

We hypothesize that the primary cause of our performance overhead is our interpreted execution step, which is described in Sects. 4.1–4.4. Recall that the execution step collects enabledness information about each of the components and filters this information through each of the program's composition operators; likewise, the execution step collects execution decisions imposed by each of the composition operators and applies them to the affected components. The execution step is interpreted, so that models (and thus generated programs) can employ heterogeneous combinations of composition operators. In contrast, the commercial tools with which we compare our work do not support heterogeneous composition operators in the same model; thus, they can optimize their generated programs for a single type of composition.

If our hypothesis is correct, then it may be that the longer execution times of our generated programs are due to our approach's flexibility with respect to composition operators, and are not due to the configurability in execution semantics. If this is the case, it may be possible to support semantic configurability with respect to execution semantics (i.e., the template parameters) and incur a smaller performance penalty by restricting the set of possible composition operators in the notation family.

There may also be other opportunities for improving performance. We have not seriously explored efficient data structures that could help to optimize the execution of our generated code [63].

7.2 Usability

What distinguishes our approach from others [3,9,14,16,20,52,58] is that the user does not need to *construct* a semantics definition for the modelling language. Instead, the user specifies parameter values for a given parameterized semantics definition—a semantics task that we believe, but have not validated, is simpler and less error prone than writing a semantics definition from scratch. Moreover, we have simplified this task further by asking users to select from menus of predefined parameter values. The result is a single tool that a user can use to experiment with several fine-grained semantics choices.

Although the task of defining a semantics is simplified, the expertise needed is not significantly reduced. The user who selects template parameter values needs to understand the consequences of those decisions: how the parameter values interact with the template definitions (and ultimately with the snapshot elements), and how the parameter values interact with each other. Most of the dependencies among template parameter values are obvious. For example, the set of state-based template parameters are interdependent, as are the set of event-based template parameters; and there are no dependencies between state-based and event-based parameter values. The more subtle interdependencies are between the choice of composition operator and template parameter values. For example, if the composition operator has an interleaving semantics, and events have a duration of "one micro-step", then a transition may miss the chance to react to an event because it might not be chosen to execute in the micro-step in which the event exists and enables transitions.

At present, it is the user's responsibility to choose template parameter values that result in a consistent and coherent semantics. This expertise is comparable to that needed in approaches in which the user writes the full semantics for their customized modelling notation. In the future, we plan to provide more guidance to users. In particular, we want to restrict a user's choice of parameter values based on the semantics decisions that he or she has already made.

To provide this kind of support, we are currently analyzing the dependencies among the template parameter values and composition operators in our family of notations.

Our template-semantics approach can also be helpful in communicating a model's intended semantics. A consequence of notation variants is that a single model can have multiple interpretations. This is currently the case with statecharts models, and the model reader needs to be instructed as to how to interpret a given model. If the intended semantics is that of a popular or standard notation, then it usually suffices to state the name of the notation in which the model is written. If the semantics deviates from widely recognized notations or variants, then another means of communicating semantics needs to be devised. One of the strengths of template semantics is that it highlights the semantics variabilities in a family of notations, and separates these out from the family's common semantics. This separation makes it easier to compare notations' semantics by comparing their template parameter values [49]. In a similar way, template parameter values could be a concise and focused way of communicating the semantics of a model. This use of template semantics still needs to be explored.

7.3 Extensibility

Our approach to semantic configurability has the same advantages and disadvantages of other software product lines: one can quickly and easily create a code generator for a particular notation that is within the family of supported notations, but configurability is limited to the set of combinations of supported template parameter values. Notable modelling features that are not supported by our CGG include transitions triggered by the *absence* of an event occurrence, pseudostates (shallow history, deep history, join, fork, junction, choice), state activities or operations, object creation or termination, change events, completion events, and time events. In previous work, we have been able to incorporate the semantics of many of these features into the template-semantics definition of our family of behavioural notations [48,49,59]. Thus, CGG could be extended to include, at least, negated event triggers and pseudostates. In general, the variability supported in CGG trails the variability in the template-semantics definition of our notation family.

The CGG is designed to ease extensibility with respect to execution semantics. In particular, the CGG is structured to localize into distinct subroutines the code that implements the various semantics choices. Thus, extending our CGG to support a new template parameter value entails (1) adding a new subroutine that implements it and (2) adding a new menu choice to the set of supported semantics options.

If CGG were to support dataflow languages, such as SCR [27] or Stateflow [41], it would need to be extended to include a new composition operator that allows for ordered execution of a model's components. Such an extension is not as easy as adding support for a new template parameter value. In addition to implementing a new class for the composition operator itself, the HTS and other operator classes would have to be extended 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. We have limited experience in adding a new composition operator to CGG: an operator that prioritizes executions in one component over those in another. This operator was added (designed, implemented, and tested) by a new team member, who was unfamiliar with the code, in the course of two days.

Adding new syntax and accompanying semantics to our family of modelling notations is currently outside the scope of our approach. That said, we have some experience in extending a template-semantics definition to incorporate new modelling features and syntax: a timing feature that supports real-time clocks and timers. This particular change to our template-semantics definition was relatively easy because it was *additive*. For example, the template ENABLED_TRANS is extended with an additional conjunct that checks the enabledness of new **clock** snapshot elements, and the template APPLY_TRANS is extended with new template parameters that update the clock snapshot elements. It is quite possible that implementing the feature in CGG would similarly be additive. Our experience with this template-semantics extension is discussed in [48].

7.4 Correctness

A threat to the validity of our results is the correctness of our implementation of CGG. The CGG is a proof-of-concept prototype, so we have applied only basic testing strategies in its verification.

Specifically, we developed a test suite that covers each of the 89 template parameter values and covers all pairs of composition operators. In addition, we developed ad-hoc tests that exercise more complicated composition hierarchies. Initially, all tests were examined manually. To assess whether a generated program matches the semantics of its corresponding model, with respect to a particular sequence of input events, we compared the program's and model's respective "outputs": that is, their sequences of snapshots. Each program outputs the contents of its shapshot elements at the end of each execution step, and this output was compared against the expected output as determined by the model and its semantics definition. If the generated program was deemed correct, then its verified output was subsequently used in automated regression testing. Nondeterministic programs, however, were always assessed manually.

A more convincing method of testing CGG would be to employ coverage and adequacy metrics that are appropriate

for testing a software product line [10,42] and that provide a more thorough coverage of feature combinations.

7.5 Generality of the approach

While this work focuses on behavioural modelling notations, our approach to semantically configurable code generation could be transferred to other families of modelling notations (e.g., sequence diagrams). The general steps of this approach are

1. Perform a commonality analysis on the family of notations, to identify the commonalities and variabilities in their semantics.
2. Devise or identify a normal-form syntax that is general enough to express models in any of the notations in the family. (We used hierarchical state machines.)
3. Express the semantics of the family in terms of the identified commonalities and variabilities. The key here is to identify the semantic variation points and the set of possibilities at those points. (We used a parameterized structured operational semantics that we call template semantics, and menus of semantic parameter values for each template parameter.)
4. Encode as data structures in the target programming language both the model (in our case, the state hierarchy, composition hierarchy, state transitions, etc.) and the elements that make up the semantic domain (in our case, the configuration of states, pool of events, variable valuations, etc.)
5. Encode in the target programming language a skeletal algorithm that simulates the model taking an execution step.
   - The algorithm should simulate the common semantics expressed in step 3, and operate on the data structures defined in step 4.
   - The algorithm should be parameterized (e.g., using subroutines) with respect to the variation points expressed in step 3, to support semantic variability.
6. For each subroutine in the simulation algorithm (representing a semantics variation point, as described in step 5), encode for each possible variant an appropriate implementation of the subroutine. Exactly one implementation for each subroutine name is included in any actual code generator, so there is no need to worry about name clashes.

A CGG, then, is implemented as a software product line; and individual code generators instantiate the product line with specific subroutines for each semantic variation point.

8 Related work

Most model-driven-engineering environments are centred on a single modelling notation that has a single semantics [5,7,12,26,29,56,31,30,60]. Configurability in such systems is geared more towards flexibility in the target language or platform [13,18] or optimizations in the generated code than in the semantics of the modelling notation. There are a few exceptions. For example, Rational Rose RT 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 versus transition actions. ObjecTime supports both synchronous and asynchronous message passing. Such options allow the specifier some control over the modelling notation, but the scope of configurability is much smaller and coarser grained than that of our template-semantic approach, which enables multiple options for event handling, transition priority, and composition operators.

There has been increasing interest in flexible and extensible modelling notations and their support tools. At the most basic level are modelling tools that allow users to configure their modelling notations with new language features (e.g., state variables, tabular expressions) or multiple target languages. These tools support a collection of optional language features or target platforms that are predefined in the tool and are selected by the user when appropriate for a specific modelling problem [20,58]. The effort required to extend the set of predefined features seems to be roughly comparable to the effort we have experienced in extending a template-semantics definition with new language features, such as history and deep history pseudostates (as in statecharts) and clocks [48]. We have not implemented these advanced features in our CGG, nor we have incorporated other types of variability (e.g., target platforms, optimization settings) because CGG is meant to be a proof-of-concept prototype; but we do not see any obstacles to adding these features and variabilities.

Starting in the late 1990s, there was interest in more general modelling environments in which the modelling notation could be completely defined by the user. Modelling-tool frameworks take as input the definition of a modelling notation, including its semantics, and generate an MDE tool, such as a model checker or simulator, that is appropriate for the notation's semantics. A number of different methods for expressing a notation's semantics were explored, such as hypergraphs [52], inference graphs [16], graph-grammar mappings to Petri Nets [3], attribute grammars [20], structured-operational-semantics rules [9,16], and higher-order logic [14]. More generally, compiler generators

[35] are able to construct compilers directly from a language's semantics expressed using denotational semantics [2,51], operational semantics and rewrite rules [21,22], natural semantics [15], and language algebras [37]. More recent MDE frameworks, such as MetaEdit+ [36] and Kermeta [46], allow users to define their own domain-specific modelling notation and corresponding code generator; the user is responsible for expressing a model's executable semantics via the tool's own action language [46] or rule language [36], which is designed to be easier to use by developers.

A key disadvantage of all these approaches is that the user 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 defining a semantics 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 or selecting a collection of relatively small semantics-parameter values that instantiate a predefined semantics definition. The trade-off is that our CGG creates code generators only for the family of notations that is scoped by our template-semantics definition and the supported parameter values. That said, template semantics has been shown to be expressive enough to represent a wide variety of language semantics, including several variants of statecharts, process algebras, Petri Nets, and SDL88 [48,49,59]. Moreover, the CGG is structured to facilitate the adding of new template parameter values.

The decomposition of our template-semantics definition is fine grained: each template parameter represents a distinct query of the model and snapshot (e.g., identify the set of enabling states) or a distinct variation point in how one aspect of an execution step (e.g., sensing of inputs from the environment, executing a set of transitions) affects one of the snapshot elements [i.e., the configuration of states (CS), the variable evaluation (AV), etc.]. This decomposition enables us to construct highly configurable modelling tools, including code generators. The downside is that the user must be aware of the interdependencies among template parameters and must provide or select parameter values that, together, result in a comprehensible semantics. An alternative decomposition is a semantics definition that is organized into a hierarchy of higher-level concepts (e.g., event policy, concurrency policy, synchronization policy, data-store policy) and that better corresponds to a user's view of semantic variations in languages [6,17,45]. These approaches promote modularity and orthogonality in semantic definitions, with the intent to ease the task of constructing or understanding a notation's semantics. However, because each semantic variation point affects a broader scope of the semantic domain, configurable tools that are based on these types of semantic decomposition [6] have more complex implementations of the semantic choices. We are currently exploring how to combine the approaches: if the higher-level semantic concepts can be mapped to lower-level queries and changes to snapshot elements, then it may be possible to develop semantically configurable tools where the user's view is in terms of highly abstract semantic choices, but the implementation is in terms of lower-level semantic choices.

Lastly, our CGG can be viewed as an instance of generative programming, where generative software development [4,8,11,34] aims at developing families of related systems by creating generators for DSLs. In generative programming, a DSL describes the features of a system family and is used to specify individual members of the family. A generator transforms a DSL specification into an implementation. In our case, template semantics is a domain-specific language for describing the "semantic features" of behavioural modelling notations, and CGG is a generator for a corresponding family of code generators. What distinguishes our work from typical generative programming is that our "features" are not functional requirements or components, but instead are semantic parameters of a general-purpose behavioural modelling notation. Thus, our contribution is not so much a generator of product instances, but rather a flexible model-driven-engineering environment, where the modeller is able to experiment with multiple modelling notations and is able to generate corresponding code generators for those notations.

9 Conclusion

In this work, we explore semantically configurable code generation for a family of behavioural modelling notations. Configurability is in the form of semantics parameters, so that the user is spared from having to provide a complete semantics definition. Moreover, the configuration task is reduced to selecting from menus of semantics parameter values, rather than defining parameter values.

We built a proof-of-concept CGG that supports 7 different composition operators, 26 semantics parameters, and 89 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 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. The result is an environment in which modelling-notation designers can experiment with alternative semantics. Another potential use is to provide tool support for UML, which has a number of semantic variation points that match template semantics' variation points [59]. Our technology does not yet compete with commercial-grade code generators, but its future looks promising.

References

1. ApeSoft. Smartstate v4.1.0. http://www.smartstatestudio.com (2008)
2. Appel, A.W.: Semantics-directed code generation. In: Proceedings of ACM Symposium on Principles of Programming Language (POPL'85), pp. 315–324. ACM Press, New York (1985)
3. Baresi, L., Pezzè, M.: Formal interpreters for diagram notations. ACM Trans. Softw. Eng. Methodol. **14**(1), 42–84 (2005)
4. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. **1**(4), 355–398 (1992)
5. Berry, G., Gonthier, G.: The esterel synchronous programming language: design, semantics, implementation. Sci. Comp. Prog. **19**(2), 87–152 (1992)
6. Bjrklund, D., Lilius, J., Porres, I.: A unified approach to code generation from behavioral diagrams. In: Proceedings of Forum on Specification and Design Languages (FDL), pp. 21–34 (2003)
7. Burmester, S., Giese, H., Schfer, W.: Code generation for hard real-time systems from real-time statecharts. Technical Report tr-ri-03-244, University of Paderborn (2003)
8. Cleaveland, C.: Program Generators with XML and Java. Prentice-Hall, UK (2001)
9. Cleaveland, R., Sims, S.: Generic tools for verifying concurrent systems. Sci. Comp. Prog. **41**(1), 39–47 (2002)
10. Cohen, M.B., Dwyer, M.B., Shi, J.: Coverage and adequacy in software product line testing. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis (ROSATEA), pp. 53–63 (2006)
11. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York/USA (2000)
12. Harel, D. et al.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Trans. Softw. Eng. **16**(4), 403–414 (1990)
13. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from UML models. In: Proceedings of International Workshop on Software and Performance (WOSP'05), pp. 75–86. ACM Press, New York (2005)
14. Day, N.A., Joyce, J.J.: Symbolic functional evaluation. In: TPHOLs, Volume 1690 of LNCS, pp. 341–358. Springer, Berlin (1999)
15. Diehl, S.: Natural semantics-directed generation of compilers and abstract machines. Form. Asps. Comp. **12**(2), 71–99 (2000)
16. Dillon, L., Stirewalt, R.: Inference graphs: a computational structure supporting generation of customizable and correct analysis components. IEEE Trans. Softw. Eng. **29**(2), 133–150 (2003)
17. Esmaeilsabzali, S., Day, N., Atlee, J.M., Niu, J.: Deconstructing the semantics of big-step modelling languages. Requir. Eng. **15**(2), 235–256 (2010)
18. Floch, J.: Supporting evolution and maintenance by using a flexible automatic code generator. In: Proceedings of International Conference on Software Engineering (ICSE'95), pp. 211–219. ACM Press, New York (1995)
19. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, USA (1995)
20. Gao, J., Heimdahl, M.P.E., Wyk, E.V.: Flexible and extensible notations for modeling languages. In: Proceedings of Fundamental Approaches to Software Engineering (FASE), pp. 102–116 (2007)
21. Hannan, J.: Operational semantics-directed compilers and machine architectures. ACM Trans. Program. Lang. Syst. **16**(4), 1215–1247 (1994)
22. Hannan, J., Miller, D.: From operational semantics to abstract machines. Math. Struct. Comput. Sci. **2**(4), 415–459 (1992)
23. Harel, D.: On the formal semantics of statecharts. In: Symposium on Logic in Computer Science, pp. 54–64 (1987)
24. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)
25. Harel, D., Naamad, A.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4), 293–333 (1996)
26. Heimdahl, M.P.E., Keenan, D.J.: Generating code from hierarchical state-based requirements. In: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97), pp. 210–220 (1997)
27. Heitmeyer, C.L., Jeffords, R.D., Labaw, B.G.: Automated consistency checking of requirements specifications. ACM Trans. Softw. Eng. Methodol. **5**(3), 231–261 (1996)
28. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, UK (1985)
29. IBM: Rational Rose RealTime v7.0.0. http://www.ibm.com/rational (2005)
30. IBM: Rational Rhapsody in J v7.4. http://www.ibm.com/software/awdtools/rhapsody/ (2007)
31. IBM: Rational SDL Suite. http://www.ibm.com/software/awdtools/sdlsuite/ (2010). Accessed Sept 2010
32. ISO8807: LOTOS: a formal description technique based on the temporal ordering of observational behaviour. Technical Report, ISO (1988)
33. ITU-T: Recommendation Z.100. Specification and description language (SDL). Technical Report Z-100, International Telecommunication Union-Standardization Sector (1999)
34. Jones, N., Gomard, C., Sestoft, P. (eds.): Partial Evaluation and Automatic Program Generation. Prentice-Hall, UK (1993)
35. Jones, N. (ed.): Semantics-Directed Compiler Generation, volume LNCS 94. Springer, Berlin (1980)
36. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley, NY (2008)
37. Knaack, J.L.: An algebraic approach to language translation. PhD Thesis, University of Iowa (1995)
38. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements specification for process-control systems. IEEE Trans. Softw. Eng. **20**(9), 684–707 (1994)
39. Lu, Y., Atlee, J.M., Day, N.A., Niu, J.: Mapping template semantics to SMV. In: Proceedings of Automotive Software Engineering (ASE'04), pp. 320–325 (2004)
40. MathWorks. Simulink. http://www.mathworks.com/products/simulink (2010). Accessed Sept 2010
41. Mathworks. Stateflow 7. http://www.mathworks.com/products/stateflow/ (2010). Accessed Sept 2010
42. McGregor, J.D.: Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon, Software Engineering Institute (2001)
43. McMillan, K.: Symbolic Model Checking: An Approach to the State Explosion Problem. Kluwer Academic, Dordrecht (1993)
44. Milner, R.: Communication and Concurrency. Prentice-Hall, New York (1989)

45. Mosses, P.: Action Semantics. Cambridge University Press, London (1992)
46. Muller, P.-A., Fleurey, F., Jzquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Proceedings of International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713, pp. 264–278 (2005)
47. Niu, J.: Metro: a semantics-based approach for mapping specification notations to analysis tools. PhD Thesis, University of Waterloo (2005)
48. Niu, J., Atlee, J.M., Day, N.A.: Template semantics for model-based notations. IEEE Trans. Softw. Eng. **29**(10), 866–882 (2003)
49. Niu, J., Atlee, J.M., Day, N.A.: Understanding and comparing model-based specification notations. In: Proceedings of IEEE International Requirements Engineering Conference (RE'03), pp. 188–199. IEEE Computer Society Press, USA (2003)
50. Object Management Group.: Unified modelling language: superstructure. version 2.0, formal/05-07-04. http://www.omg.org (2005)
51. Paulson, L.: A semantics-directed compiler generator. In: Proceedings of ACM Symposium on Principles of Programming Language (POPL '82), pp. 224–233. ACM Press, New York (1982)
52. Pezzè, M., Young, M.: Constructing multi-formalism state-space analysis tools. In: IEEE International Conference on Software Engineering (ICSE), pp. 239–249. ACM Press, New York (1997)
53. Plotkin, G.: A structural approach to operational semantics. Computer Science Department, Aarhus University, Aarhus (1981)
54. Prout, A.: Parameterized code generation from template semantics. Master's Thesis, School of Computer Science, University of Waterloo (2005)
55. Prout, A., Atlee, J., Day, N., Shaker, P.: Semantically configurable code generation. In: ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems, pp. 705–720 (2008)
56. Selic, B., Gullekson, G., Ward, P.: Real-Time Object-Oriented Modeling. Wiley, New York (1994)
57. Smith, K.: Real-time object-oriented modeling: ObjecTime CASE tool simplifies real-time software development. Dr. Dobbs J Softw Tools **22**(12), 64–74 (1997)
58. Swint, G.S., et al.: Clearwater: extensible, flexible, modular code generation. In: Proceedings of IEEE/ACM International Conference on Automotive Software Engineerig (ASE'05), pp. 144–153. ACM Press, New York (2005)
59. Taleghani, A. Atlee, J.M.: Semantic variations among UML statemachines. In: ACM/IEEE International Conference on Model-Driven Engineering Languages and Systems, pp. 245–259 (2006)
60. Tiella, R., Villafiorita, A., Tomasi, S.: FSMC+, a tool for the generation of Java code from statecharts. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, pp. 93–102 (2007)
61. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: UML 2004—The Unified Modeling Language, volume LNCS 3273, pp. 290–304, October 2004 (2004)
62. von der Beeck, M.: A comparison of statecharts variants. In: ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 128–148 (1994)
63. Wasowski, A.: On efficient program synthesis from statecharts. In: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pp. 163–170 (2003)
64. WindRiver.: Betterstate. http://www.windriver.com/portal/server.pt (2005)
65. Yavuz-Kahveci, T., Bultan, T.: Specification, verification, and synthesis of concurrency control components. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA'02), pp. 169–179. ACM Press, New York (2002)
66. Zave, P., Jackson, M.: A call abstraction for component coordination. In: International Colloquium on Automata, Languages, and Programming: Workshop on Formal Methods and Component Interaction (2002)

Author Biographies

Adam Prout (BMath. and MMath., Computer Science, University of Waterloo) is employed by Microsoft as a software engineer. His graduate studies were focused on software modeling.

Joanne M. Atlee (Ph.D. and M.S., Computer Science, University of Maryland; B.S., Computer Science and Physics, College of William and Mary; P.Eng.) is an Associate Professor in the David R. Cheriton School of Computer Science at the University of Waterloo. Her research interests include software modelling, automated analysis of software models, modular software development, feature interactions, and software-engineering education. Atlee serves on the editorial boards for Software and Systems Modeling and Requirements Engineering. She is an at-large member of the ACM SIGSOFT Executive Committee and is a member of the International Federation for Information Processing (IFIP) Working Group 2.9 on Software Requirements Engineering. She was Program Co-Chair for the 31st International Conference on Software Engineering (ICSE'09) and was Program Chair for the 13th IEEE Requirements Engineering Conference (RE'05).

Nancy A. Day received the M.Sc. and Ph.D. degrees in computer science from the University of British Columbia in 1993 and 1998 respectively, and the B.Sc. degree in computer science from the University of Western Ontario in 1991. From 1998 to 2000, she was a postdoctoral research associate at the Oregon Graduate Institute. She is currently an associate professor in the Cheriton School of Computer Science at the University of Waterloo and is a member of the Waterloo Formal Methods (WatForm) research group. Her research interests include formal methods, requirements engineering, and system safety. She is a member of the ACM and IEEE Computer Society.

Pourya Shaker is a Ph.D. student in the Waterloo Formal Methods group of the School of Computer Science at the University of Waterloo. He received B.Eng. and M.Eng. degrees in Computer Engineering at Memorial University. His research interests are in the area of model-driven software development, specifically aspect-oriented and feature-oriented modelling languages.