Parameterized Code Generation from Template Semantics

by

Adam Prout

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We have developed a tool that can create a Java code generator for a behavioural modelling notation given only a description of the notation's semantics as a set of parameters. This description is based on template semantics, which has previously been used to describe a wide variety of notations. As a result, we have a technique for generating Java code for models written in any notation describable in template semantics. Since template semantics allows for models containing nondeterminism, we introduce mechanisms for eliminating this nondeterminism when generating code. We describe Java implementations of several template-semantics composition operators that have no natural Java representations and give some techniques for optimizing the generated code without sacrificing correctness. The efficiency of our generated code is comparable to that of commercial notation-specific code generators.

Acknowledgments

Without the contributions, guidance, and technical insights of my supervisor, professor Joanne M. Atlee, and thesis reader, professor Nancy A. Day, this thesis would not have been possible. Their support and hardwork right from the inception of my thesis work through to its completion is greatly appreciated. I learned much about research and writing from my time spent with them. I would also like to thank professor Andrew J. Malton, who took the time to read my thesis and provide insightful feedback.

Thanks to my fellow graduate students in the WatForm research group, and more specifically those students working on the Metro project. Their expertise helped work out some of the details of my research.

My parents, Doug and Cathy, provided support both financially and nutritionally throughout my years at Waterloo. Their encouragement kept me focused on completing my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis investigates parameterized code generation from models written in model-based notations. We have developed a semantics-based code-generator generator (CGG) that can create a Java code generator for a modelling notation given a template-semantics description of the notation's semantics. This chapter introduces some of the motivations behind this approach and describes the structure of the thesis.

1.1 Motivation

Model-driven development's (MDDs) methodology is to express all important aspects of a system's business and application logic using appropriate domain-specific models[26]. This provides separation between platform-independent models (PIMs) and their associated platform-specific models (PSMs). A PIM is a model normally written in a domain-specific language (DSL) for specifying the system, and a PSM is generally a source-code artifact. This separation has many advantages. First, PIMs express the system's specification and design in a much more intuitive fashion. Second, a PIM can be automatically translated to a PSM. This translation helps in the generation of correct source code and allows the system to be ported to other languages and platforms with ease. Third, formal-verification techniques,

such as model checking[7], are better suited to PIMs due to their higher level of abstraction. The key to all these benefits is the existence of translators or code generators to transform a PIM into an equivalent PSM for a given platform. Without such translators, the separation of PIMs and PSMs becomes cumbersome. To allow PIMs to be written in DSLs, we need a technique to build code generators for a given DSL easily. Otherwise, MDD technology will be limited to a few standard modelling notations whose semantics may, or may not, be appropriate for modelling a given system.

In this thesis, the PIMs we are specifically interested in are those written in **model-based requirements notations**. Model-based notations are used to specify a system's reactive behaviour. Examples include such notations as statecharts[12], STATEMATE[13], RSML[20] and SDL88[1]. These notations allow the user to build a model of a system, often in the form of a state machine, that describes the allowable execution steps of the system. They support concurrency and synchronization through the use of composition operators that describe how multiple state machines take steps concurrently.

Our goal is to allow a PIM to be written in any model-based notation describable in template semantics, and to be able to generate a representative Java program for that PIM. Template semantics[24, 23] structures the operational semantics of a family of notations as a set of predefined templates that are instantiated with user-provided parameter values. Thus, template semantics allows a user to specify the semantics of a modelling notation by providing values for a set of template parameters. The template captures the common aspects of family members and the parameter values capture the unique aspects. Template semantics has been used to describe a wide variety of notations including SDL88[1], SCR[15], LOTOS[19], CSP[16], and several statecharts[12] variants. Given a model and a template-semantics description of its notation, we will demonstrate how to generate Java code whose behaviour matches the specified semantics of the model.

We take a generative approach[8] to Java code generation based on product-family development. Generative programming's goal is to create generators that take as input a feature specification in a domain-specific language and that output

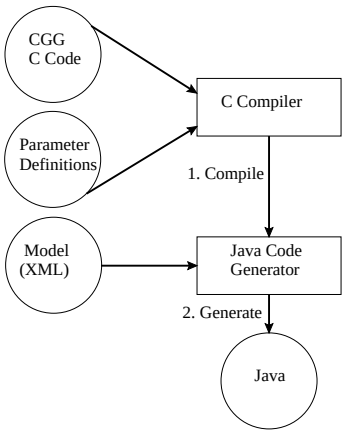


Figure 1.1: Code Generation Process

the family member described in the specification. We want to apply the generative approach to building Java code generators for model-based notations from their template-semantics descriptions. The product-family, in this case, is the family of Java code generators for model-based notations. The domain-specific language for describing family members is template semantics. A template-semantics description of a notation essentially indicates which features a code generator for that notation must posses. This approach allows us to quickly produce a Java code generator for a given PIM by giving our generative tool a description of the semantics (features) of the PIM's notation.

The generative approach to creating Java code from a model has two stages (Figure 1.1). The first stage is the generative stage described above. In this stage a notation-specific code generator is compiled for the model's notation. The user specifies the semantics (features) of his modelling notation by providing a file with several C preprocessor #define directives, one for each template-semantics parameter. The code-generator generator (CGG) is a C program that has been annotated with preprocessor directives that indicate the parts of the source program that are specific to each supported template-parameter value. This code structure enables conditional compilation to compile only the parts of the code that are required to generate Java for a notation's specified features. Compiling the CGG source code, along with the template-parameter definition file, produces a code generator for the

user-specified modelling notation.

The second stage is to execute the produced code generator on a model written in the user's modelling notation and have it generate Java. That is, this step transforms a PIM into a representative PSM for the Java platform. Our code generators expect an XML representation of the model as input. This XML format is used by all existing tools which make use of template semantics. The generator's translation algorithm produces an abstract-syntax tree in C representing the structure of the model. This syntax tree is traversed to fill several intermediate data structures that are used to increase the efficiency of code generation. These data structures store the results of frequently used computations on the syntax tree (i.e., a list of all nodes at the leaves of the tree, a list of all ancestors of a given node, etc). The tree is traversed in pre-order a final time to produce Java code corresponding to each node in the tree.

This two stage approach separates the logic for processing the template parameters from the logic for generating Java for a model. Thus, a user can build a code generator for a given notation once, and then use that generator over and over without having to reprocess the parameter values each time. We project that this will be the typical usage of the tool.

1.2 Main Contributions

This thesis demonstrates how to generate Java code from models written in a template-semantics-supported modelling notation. The main contributions of this thesis are

- **Supporting composition operators with no natural Java representation:** Java's only built-in support for concurrency interleaves the execution of multiple threads. We show how to encode in Java all of template semantics' composition operators, including some that require concurrent machines to synchronize at certain points during their execution by taking steps in parallel.

- **Techniques for eliminating nondeterminism:** Template semantics allows for notations that can specify nondeterministic models. For example, a notation that allows a model to have more than one enabled transition at some point during the model's execution. Leaving this nondeterminism unresolved in the modelling phase is natural, but it needs to be eliminated when translating to source code. Java's model of computation is inherently deterministic, and it is costly to simulate nondeterminism (e.g., using random number generators).
- **Techniques for optimizing the generated Java:** We show how to generate efficient Java code without sacrificing correctness.

1.3 Evaluation

We evaluate our code-generator generator (CGG) with respect to the following characteristics of the generated Java code

- **Correctness:** The generated Java code should match the model from which it was created. That is, if the code and model are in equivalent states at a given observable point in a deterministic execution, then their respective next observable points in the execution must be equivalent states. The definition of an observable point in an execution is part of template semantics and is described in Section 2.1. We don't present a formal proof of this correctness criteria. This criteria is what we are aiming to validate when testing and inspecting the generated Java code.
- **Efficiency:** We compare the efficiency of our generated Java with the Java created by commercial notation-specific code generators. This comparison allows us to assess the cost of supporting configurable semantics (the cost of generality) as compared to notation-specific code generators that may include notation-specific optimizations not possible in our code generators.

We evaluate correctness by testing and inspecting the Java generated from a test suite of models containing different combinations of composition operators. We also test the code created by code generators built using different combinations of

template-parameter values. To demonstrate the functionality of the CGG on a real world system, we show the results of a case study based on an aircraft ground-traffic control system. The model built for this study uses several of the more complex composition operators. We evaluate efficiency by comparing the execution speed of the generated Java of our CGG-created code generators to the execution speed of the generated Java of four commercial tools that support specific notations.

1.4 Goals

We also have the following goals, which are difficult to evaluate objectively:

- **Extendibility of the CGG:** The CGG source should be structured to allow the easy addition of new template-parameter values. By easily, we mean that the required changes to support a new value for a given parameter should be localized, such as being limited to a single procedure in the CGG source code. The CGG should also allow new composition operators to be implemented.
- **Readability of the generated Java:** The structure of the generated Java code should closely match the modular structure of the model. We want the human modeller to be able to locate the modules of the generated code that correspond to modules of the model and to be able to modify the generated code (i.e., by adding extra code that executes on a transition, or extra code to sense the environment for input).

1.5 Thesis Organization

Chapter 2 introduces template semantics and some basics on Java threading that are required to understand the explanations of our Java translation. Chapter 3 describes the translation from models written in a template-semantics-supported notation to Java code. The focus of this chapter is on how a step in the model's execution (i.e., the firing of one or more transitions) is realized in Java. Chapter 4

looks at how to refine the algorithms given in Chapter 3, which mimic a model's nondeterminism, to resolve the nondeterminism and generate deterministic code (i.e., no random choices). We then carry out some validation in Chapter 5 by introducing the ground-traffic control case study and by using some benchmark models to compare the efficiency of our generated code to the code that is generated by commercial notation-specific tools.

Chapter 2

Background

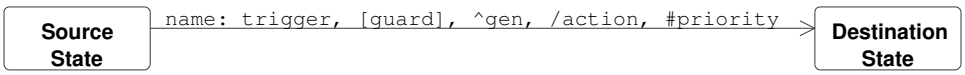
In this section, we describe the details of template semantics, including its model of computation, template parameters, and composition operators. We also give a brief overview of the concurrency primitives built into Java.

2.1 Template Semantics

Template semantics allows one to precisely describe the semantics of a model-based notation. Template semantics provides a set of pre-defined templates that captures the common aspects of the family of model-based notations. A user specifies a particular notation by filling in the template parameters with appropriate values. The basic unit of syntax for a template-semantics-supported notation is a nonconcurrent extended state machine called a **hierarchical transition system** (HTS). Concurrency is introduced through **composition operators**, which define how groups of HTSs execute together (in parallel, interleaved etc.). Template semantics has previously been used to describe the semantics of notations such as SCR[15], LOTOS[19], CSP[16], and several statecharts[12] variants. An informal description of template semantics follows; for a more complete description, see[24, 23].

2.1.1 Syntax of an HTS

An HTS contains states, transitions, events, and typed variables but no concurrency. A transition in this system has the form:



Where `trigger` is a list of possible triggering events, `guard` is a boolean guard condition over the HTS's variables, `gen` is an event to generate, `action` is an action to execute on the HTS's variables (a transition can have multiple `^gen` and `/action`'s), and `priority` is the priority value attached to this transition. `Trigger`, `guard` and, `priority` are used to determine if the transition is enabled, and `gen` and `action` are side effects that occur when the transition executes. HTSs support a state hierarchy through super states and default sub-states. A super state is a state containing one or more sub-states. A default sub-state is the sub-state that is entered by default whenever its super-state is entered. An HTS also has a set of initial states, and optionally, a set of final states. Figure 5.3 in Chapter 5 shows an example HTS for an airport ground-traffic controller.

2.1.2 Semantics of an HTS

An HTS's execution is formalized as a sequence of snapshots. A snapshot is an 10-tuple $(CS, IE, O, AV, Q, CSa, IEa, AVa, Qa, Ia)$ that records the current states (CS), internal events (IE), output events (O), variable values (AV), and queue contents (Q) of the HTS. CSa, IEa, AVa, Qa, and Ia are auxiliary elements used to store additional information about states, internal events, outputs, variables, queues, and external events, respectively. Internal events are those generated by executing transitions while external events are those sensed from the environment. The Q and Qa snapshot elements are new to this thesis and are not in the original template semantics definition. An execution is represented by a sequence of snapshots $s_0, s_1, s_2...$ starting at an initial snapshot reflecting initial states and variable values. An allowable step in the execution moves the HTS between consecutive snapshots (from s_i to s_{i+1}). There are two different types of steps. A **micro-step**

is a single transition's execution in an HTS. A **macro-step** is one or more micro-steps executed in succession in response to events sensed from the environment. The length of a macro-step is specified by a template parameter. The only time an HTS senses the environment for input is at the start of a macro-step. Thus, the macro-step parameter determines how often the model will sense the environment.

2.1.3 Template Parameters

The template parameters can be broken down into four categories: enabled-trans, apply, reset, and miscellaneous.

1. **enabled-trans:** These parameters determine how the snapshot elements are used to check if a transition is enabled. The enabling parameters are en_cond(ss,τ), en_events(ss,τ), and en_states(ss,τ), where ss is the current snapshot and τ is the transition whose enabledness we are testing. These parameters are relations that determine if a transition is enabled based on its guard condition, triggering events, and source state respectively. Example values for each parameter include

   - *en_cond*: Check if the variable values in the AV snapshot element satisfy τ's guard condition.
   - *en_events*: Check if τ's triggering events are in the internal events (IE) snapshot element.
   - *en_states*: Check if τ's source state is in the current states (CS) snapshot element.

2. **apply:** These parameters are relations that determine how the snapshot elements should be updated as a result of a transition's execution. Ten next_XX(ss, τ, XX') parameters (one for each snapshot element) encode the behaviour of apply, where XX is the snapshot element (CS,IE,...) that is updated, ss is the current snapshot, and τ is the transition that was chosen to execute. Example values for some of the parameters include

- *next_CS*: Remove τ's source state from CS and add its destination state.
- *next_IE*: Overwrite IE with τ's generated events removing all previously generated events.
- *next_AV*: Update the variable values in AV with the assignments in τ's actions.

3. **reset:** These parameters are functions that determine how to reset the snapshot elements at the beginning of a macro-step. Reset cleans up left-over information from the previous macro-step's execution and incorporates events sensed from the environment into the snapshot. Reset is encoded by ten reset_XX(ss,I) parameters (one for each snapshot element), where XX is the snapshot element being reset, ss is the current snapshot, and I are events sensed from the environment. Example values for some of the parameters include

   - *reset_CS*: Make no change to CS.
   - *reset_IE*: Remove all events accumulated in IE in the previous macro-step by setting IE to empty.
   - *reset_Ia*: Add to Ia all external events sensed from the environment.

4. **miscellaneous:** The pri, macro_semantics, and resolve parameters are used to specify the priority scheme, macro-step semantics type, and how to resolve shared-variable conflicts, respectively.

   - The pri parameter specifies a priority scheme over all transitions in the model. Example priority schemes include explicit priority (the user explicitly provides a priority for each transition) and priority schemes based on the depth of a transition's source state in the state hierarchy.
   - The macro_semantics parameter can specify either simple or stable semantics. Under simple semantics, a macro-step is at most a single micro-step in length. Thus, the environment is sensed and the snapshot is reset

(using the reset parameters) after every transition. Under stable semantics, a macro-step is a series of micro-steps ending in a stable snapshot (one in which no transition is enabled).

- The resolve parameter is used to resolve conflicts that occur when shared variables are assigned different values simultaneously. This parameter indicates how to choose which single value a shared variable will take on. For example, the resolve parameter could specify that the model nondeterministically choose one of the values a shared variable was assigned to be the value for the variable.

By providing values for these parameters, a user can precisely define a notation's semantics. The user can choose amongst 3-5 implemented values for each parameter. The only exception is the resolve parameter, which currently has a single implemented value: nondeterministic resolution. Appendix A gives a complete listing of the implemented parameter values.

2.1.4 Template Semantics for SmartState

In this section we provide a complete template semantics description of SmartState[4], a commercial statecharts variant. This example demonstrates the type of information that a user needs to provide to describe a notation in template semantics. We developed this template semantics description to be used in Section 5.3, to generate a code generator for SmartState. The other notations referred to in this thesis will not be accompanied by a complete template-semantics description.

SmartState supports both internal and external events. External events are active only in the first micro-step of a macro-step, but internal events persist until they trigger a transition or until the end of the macro-step in which they were generated. SmartState only allows internal events to be generated by the first transition of a macro-step. All transitions must have at least one triggering event. Below is the template-semantics description for SmartState along with an explanation of each parameter value.

- *reset_Ia(ss,I) : I.gen*
  At the start of a macro-step, Ia is set to the external events sensed from the environment (I.gen). This will overwrite any previously sensed external events.
- *reset_IE(ss,τ) : $\emptyset$*
  At the start of a macro-step, all previously generated internal events are discarded. This ensures that internal events can trigger transitions only in the same macro-step in which they are generated.
- *next_Ia(ss,τ,Ia') : Ia'=$\emptyset$*
  When a transition executes, all previously sensed external events are thrown out. This allows only one external-event-triggered transition to execute per macro-step.
- *next_IE(ss,τ,IE'):(ss.Ia $\neq \emptyset \Rightarrow$ IE'=gen(τ)) $\wedge$ (ss.Ia $= \emptyset \Rightarrow$ IE'=ss.IE$\backslash$trig(τ))*
  The first transition to fire at the start of a macro-step fires in response to environmental input. This parameter value states that if τ is the first transition to fire in the macro-step (i.e., Ia contains the external event that triggers τ, so ss.Ia is not empty), then IE is set to the internal events generated by τ. If τ is not the first transition to execute in the macro-step (which means that Ia must be empty), then τ's triggering event is removed from IE and any events τ generates are ignored. Thus, this parameter value asserts that only transitions that are triggered on external events can generate internal events, and that each of these internal events will trigger only one subsequent transition in the macro-step.
- *en_events(ss,τ) : trig(τ) $\subseteq$ ss.IE $\cup$ ss.Ia*
  In SmartState, every transition must have a triggering event. This event can be internal or external. To be enabled, transition τ's triggering events must be in either IE or Ia. IE contains internal events generated in this macro-step and Ia contains external events sensed at the start of this macro-step.
- *reset_CS(ss,I) : ss.CS*
  At the start of a macro-step, the current states (CS) are not changed.

- *next_CS(ss,τ,CS') : CS'=entered(dest(τ))*
  When transition τ executes, all states currently in CS are over-written with the destination state of τ (dest(τ)) and all of dest(τ)'s default sub-states and ancestor super-states (preserving the state hierarchy).
- *en_states(ss,τ) : src(τ) $\subseteq$ ss.CS*
  To be enabled, transition τ's source state (scr(τ)) must be in CS.
- *reset_AV(ss,I) : ss.AV*
  At the start of a macro-step the variable values (AV) are not changed.
- *next_AV(ss,τ,AV') : AV'=assign(ss.AV, eval(ss.AV, incr(asn(τ))))*
  When transition τ executes, variable values in AV are updated incrementally according to the assignments in the actions of transition τ (asn(τ)). Incremental update means that if τ's actions include multiple variable assignments, each is evaluated and executed in order. Thus, an earlier assignment can effect the value of later assignments (e.g., at the end of x:=1; y:=x;, both x and y have a value of one) and if there are multiple assignments to the same variable the later assignments overwrite the earlier assignments (e.g., at the end of x:=1; y:=2; x:=y;, x and y have the value of 2).
- *en_cond(ss,τ) : ss.AV $\models$ cond(τ)*
  For transition τ to be enabled, the current variable values in the model must satisfy the guard condition of τ.
- *macro semantics : stable*
  The model senses the environment at the start of a macro-step and then does not sense again until no more transitions are enabled (the model is stable).

2.1.5 Composition Operators

Composition operators define how groups of HTSs should execute concurrently. In template semantics, all composition operators are binary. An operator's operands are either HTSs or other composed HTSs. Each operator defines how its operands take a collective micro-step. The seven composition operators that we refer to in this thesis are:

- Interleaving: If both operands are enabled, then one or the other can execute in a micro-step, but never both. If only one operand is enabled, then it executes.

- Parallel: If both operands are enabled, they both execute in parallel in the same micro-step. If only one operand is enabled, then it executes.

- Environmental Synchronization: Each environmental synchronization operator has a designated set of synchronization events associated with it. Both of the operator's operands execute in parallel if they are both enabled on the same synchronization event; otherwise, the operands interleave the execution of transitions that are *not* triggered by synchronization events.

- Rendezvous Synchronization: Each Rendezvous synchronization operator has a designated set of synchronization events associated with it. If one operand generates a synchronization event and the other operand is triggered by this same event, then both operands execute in parallel. Otherwise, the operands interleave the execution of transitions that are *not* triggered by synchronization events.

- Interrupt: Only one of the two operands has control and can execute. Control is transferred from one operand to the other via a set of interrupt transitions whose source is in one operand and whose destination is in the other. This operator is template semantics' means of modelling transitions between concurrent regions, as seen in many statecharts[12] variants.

- Sequence: One operand executes until finished, and then the other operand executes until finished.

- Choice: When the model starts, one of the two operands is nondeterministically chosen to have control. This operand has control and is the only operand that can execute transitions throughout the lifetime of the model.

2.2 Java

Java[10] is a general-purpose object-oriented programming language designed by Sun Microsystems. Java contains native support for concurrency and synchronization using threads, monitors, and locks. Java threads are fundamentally interleaved. The scheduler chooses a non-blocked (not blocked in a monitor and not blocked at a lock) thread to execute. This thread executes in isolation until blocked or until the scheduler orders it to stop executing.

A monitor is a synchronization construct containing mutex methods and condition queues. In Java, each monitor has a single condition queue and a single mutex lock. The mutex lock ensures that only a single thread can access the monitor at a time and the condition queue is used to store threads (in a queue) that are waiting for some condition to become true before accessing the monitor.

For a thread to access a monitor by calling one the monitor's mutex methods, it must first implicitly acquire the monitor's mutex lock. The mutex lock is implicitly released when the thread exits the monitor (when it finishes executing the mutex method). This implicit locking ensures that only a single thread, the thread that holds the mutex lock, can ever be inside the monitor at any point in time.

The monitor's condition queue is used to synchronize threads from within the monitor. Two possible operations are defined on a condition queue: wait() and signal(). A thread that calls wait() inside one of the monitor's mutex methods blocks and enqueues itself onto the back of the monitor's condition queue. A thread enqueued on a condition queue sleeps and will not execute until it is removed from the condition queue. Calling wait() also causes the thread to release the monitor's mutex lock, allowing another thread to enter the monitor. The thread waiting at the front of the condition queue wakes up when another thread calls signal() from within the monitor. A signalled thread needs to re-acquire the mutex lock before it continues execution at the same point where it previously called wait().

Thus, a Java monitor forces threads to be in one of three possible states: blocked waiting to acquire the mutex lock (because some other thread is executing inside the monitor and is holding the lock), blocked on a condition queue waiting for another

thread to signal it, or executing within the monitor holding the mutex lock.

Chapter 3

Parameterized Code Generation

In this chapter, we describe the architecture of the Java application that is generated by our CGG-created code generators. The process of generating Java has two major parts. The first is generation of Java for the composition operators used in the model. The second is generation of Java for each HTS in the model, as directed by the template parameters. The modeller can choose values for each template parameter from a list of implemented values (the list of parameter values supported by the CGG is shown in Appendix A).

3.1 Scheduling and Execution

A number of high-level decisions need to be made regarding how elements of a model will be represented in Java. Template semantics allows for notations containing hierarchies of concurrent entities (HTSs) and synchronization constructs (composition operators). Java has built-in support for concurrency using threads and built-in support for synchronization using monitors.

We initially investigated a multi-threaded approach that mapped each HTS to its own thread in Java and each composition operator to a monitor, as seen in Figure 3.1. The monitors formed a binary tree structure in Java that mirrored the tree of composition operators in the model. A monitor synchronized its child

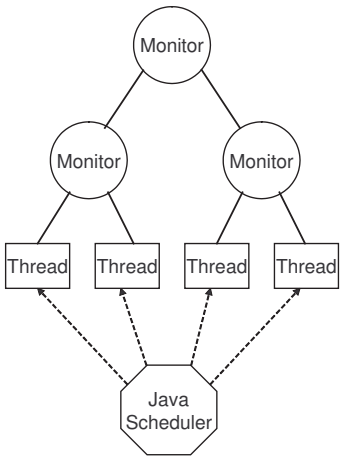


Figure 3.1: Multi-threaded Mapping

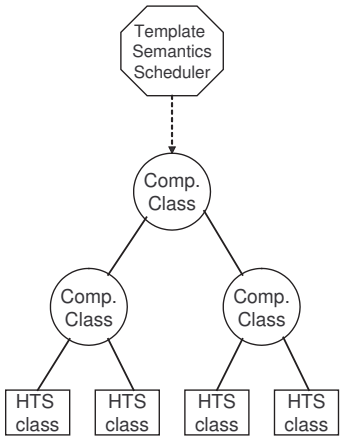


Figure 3.2: Single-threaded Mapping

threads (using its condition queue) by following the semantics of the composition operator it represented. This multi-threaded design has all the usual benefits of threading. It allows for better resource sharing and prevents blocking operations (i.e., blocking for input on a socket or standard input), from blocking the execution of the entire system. The main disadvantage of threading is the greatly increased complexity required to manage the synchronization among threads. Under the threads and monitor mapping, we would end up synchronizing a lot of threads (i.e., forcing many threads to block on condition queues) during every micro-step in the model. A micro-step is generally too small of a unit of work to make up for the cost of this synchronization. The source of this synchronization problem is the set of composition operators that require concurrent machines to take steps in parallel: parallel, environmental synchronization, and rendezvous operators. Java has no notion of executing multiple instructions at the same time. Parallel execution is simulated by forcing the system to block all threads on condition queues except those that are to execute the parallel transitions. The parallel transitions then execute in isolation, in some interleaved order, and update the system with the combined result of their execution when finished. When the parallel execution is finished, all previously blocked threads (i.e., those threads not involved in the parallel execution) are signalled and removed from condition queues allowing them

to potentially execute in the next micro-step. As a result, the multi-threaded design breaks one of the most fundamental rules of using concurrency. Namely, that the work each thread is assigned should be largely independent of the work assigned to other threads, and that synchronizing of threads should be kept to a minimum.

In multi-threaded programs, the default situation is one in which all threads execute concurrently and independently and only synchronize to collaborate with each other once in a while. But the nature of template-semantics' micro-steps is that the composition operators decide which HTSs (and sometimes which enabled transitions) are allowed to execute in a particular micro-step. To simulate these semantics, the default situation in our multi-threaded generated code is that a program's threads *do not* execute; rather in every micro-step, a chosen few threads will be selected for execution. And the only way to realize this centralized decision is to force all threads to block in every step, and then release the chosen few. The result is little real concurrency per step. This problem lead us to abandon the multi-threaded approach.

The key to an improved mapping is to notice that the Java scheduler chooses a thread for execution based on some internal scheduling algorithm that is completely independent of whether the HTS executing in that thread is enabled and can execute. As was stated above, the potential for parallel transition execution forces us to synchronize threads on every step of the system. Thus, we need our own template-semantics-aware scheduler that wakes up only the HTSs that are actually enabled. If multiple HTSs are to execute together, then our scheduler will schedule them to execute, ignoring all other HTSs. This is the opposite of what occurs if we use the Java scheduler in the multi-threaded mapping. In that mapping, all HTSs would be running and we would have to stop the threads that are not part of the parallel execution.

Under the single-threaded mapping, HTSs and composition operators both map to Java classes, as shown in Figure 3.2. They are organized into a composition tree that mirrors the model's compositional hierarchy. Our own scheduler determines which transitions are enabled and schedules HTSs for execution based on this knowledge. There is only a single Java thread in the system. The complexity of dealing

with synchronizing threads has been removed. It is this mapping that the rest of this chapter describes in detail.

Figure 3.2 seems to indicate that an explicit scheduler component will control the execution of the model. While this is true in an abstract sense, the scheduler itself is actually built into each of the composition-operator and HTS classes. Each micro-step in the model has two stages to its execution in the Java code. The first stage is an enabledness check over the entire composition tree. Information about enabledness flows up the composition tree from the HTSs at the leaves to the root component. The type of information that flows up the tree varies based on the types of operators present in the tree. At the very least, this information will include a boolean enabled flag indicating the existence of an enabled transition in an HTS, and will sometimes include the events which trigger these transitions. This information flow occurs through parameters modified by each operator's enabled_check() method, as shown in Figure 3.3. The root calls this method on both its children, who subsequently call it on their children, performing a post-order traversal of the composition tree. When the method call returns to the root component, the root will have global knowledge about which HTSs are prepared to execute and the types of transitions they are prepared to execute. Moreover, each component or HTS will have cached its enabledness information locally, in case it is scheduled for execution in the future.

The second stage of a micro-step is the execution stage, as shown in Figure 3.4. Once the root has collected its children's enabledness information, it will make an execution decision. An execution decision involves directing one or both child components to execute and providing them constraints on the types of transitions they are allowed to execute. Example execution decisions include instructing components to synchronize on a sync event, instructing a particular enabled interrupt transition to execute, or allowing a component to make its own unconstrained execution decision. As execution decisions flow down through the composition tree, each node makes local execution decisions that satisfy the constraints imposed by its ancestors' decisions. Execution decisions flow down the tree in this manner until they reach the HTSs at the leaves. It is at this point that the chosen transitions

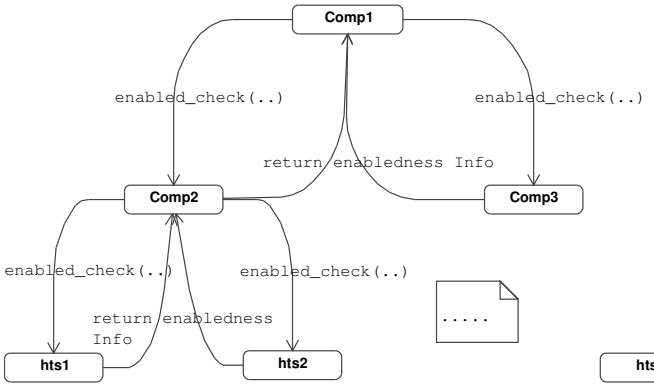


Figure 3.3: Enabledness Checking

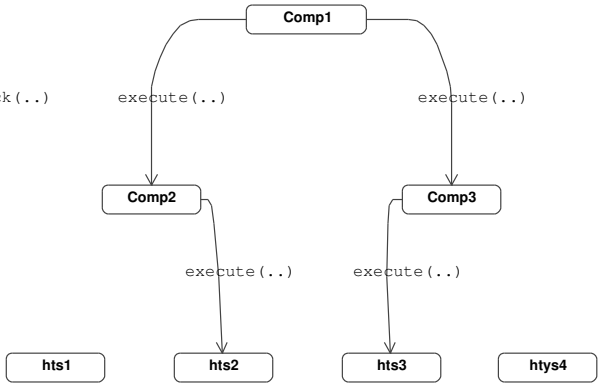


Figure 3.4: Execution

are executed. The two stages of a micro-step's execution in Java are similar to the two stage approach used by Express[21], a tool that generates SMV[22] code given a model and template semantics description of the model's notation.

Recall that template semantics supports the notion of HTSs taking steps in parallel. Java has no notion of executing two instructions at the same time; everything is fundamentally interleaved. In the following sections, whenever we refer to HTSs taking steps in parallel (i.e., both HTSs execute a transition in the same micro-step) we mean that the HTSs will simulate parallel execution. This simulation is carried out by having the HTSs execute in isolation in some interleaved order and then updating their snapshots when the step is finished with the combined effects of executing all of the parallel transitions. The only complication is updating the values of variables that are shared by multiple HTSs. The details of how shared variables are handled are deferred until Section 3.4.2.

3.2 Architecture of the Generated Code

The object diagram in Figure 3.5 shows the architecture of the Java code that would be generated for the example system shown in Figure 3.6. The backbone of the generated code consists of a binary tree of objects representing composition operators (root and C1) with HTS objects (M1, M2, M3) at the leaves. Each template-

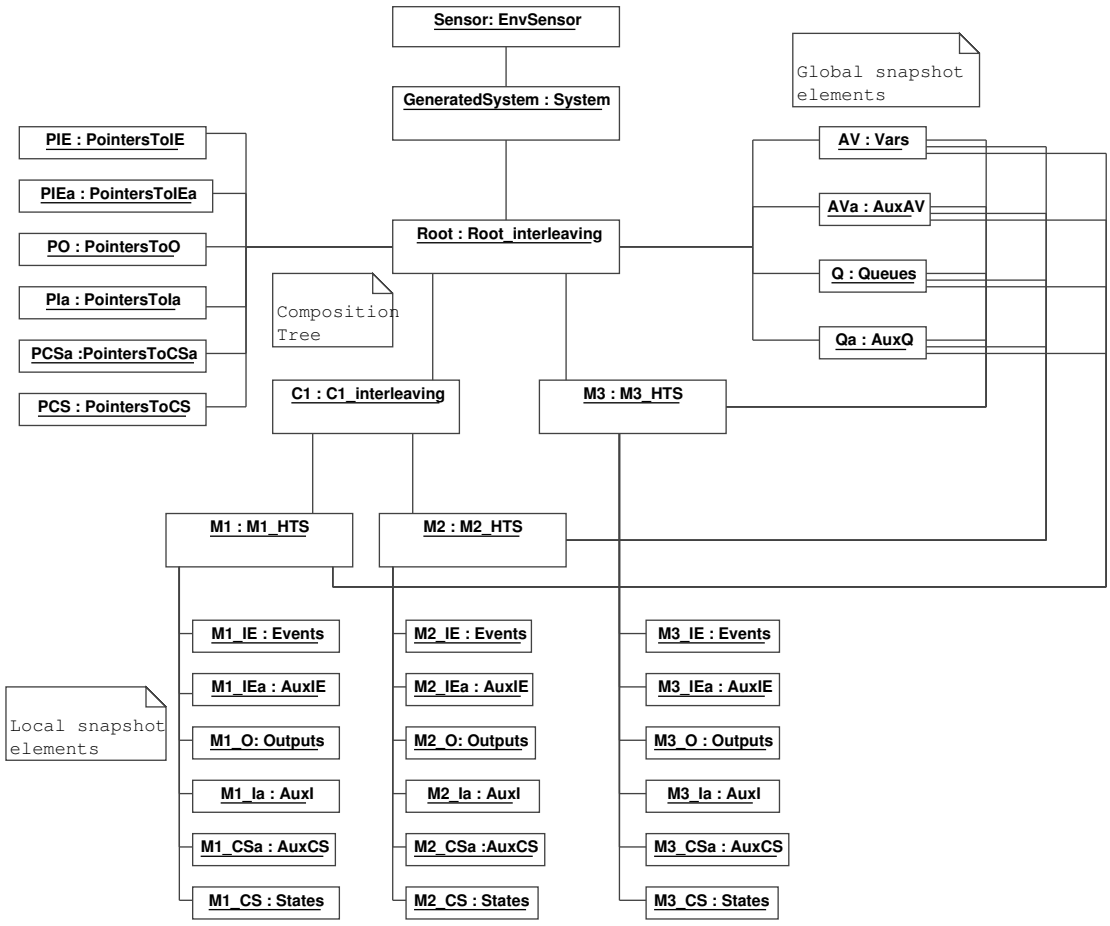


Figure 3.5: Generated Java Object Diagram

semantics snapshot element also maps to a class. Global snapshot elements such as AV, AVa, Q, and, Qa are represented by classes Vars, AuxAV, Queues, and, AuxQ, respectively; they each have one instance (AV, AVa, Q, Qa) that is shared by the entire system. This is in contrast to local snapshot elements IE, IEa, O, Ia, CS, and CSa, which each have an instance that is stored locally at each HTS object. These snapshot-element classes implement a specific data type (set, queue, set of queues, etc.) whose operations match the operations specified in the respective reset_XX and next_XX parameters. There are also repository classes (PointersToXX) for each of the local snapshot-element classes. These classes are needed by the root of the composition tree to reset the snapshot elements at the start of a macro-step. The individual snapshot-element objects the root needs to reset are stored at the leaves of the tree in the HTS objects. The root uses the PointersToXX classes to quickly access these local snapshot-element objects. These classes store reference pointers to all local snapshot-element objects of a given type (these relationships are not shown in Figure 3.5). For example, the PointersToIEa class stores a reference to each local AuxIE object (M1_IEa, M2_IEa, M3_IEa) in the system. These references facilitate the efficient execution of the reset_XX parameters by the root, as will be described in Section 3.7.

The System class is responsible for constructing the composition tree on start up and for establishing references to the shared global-snapshot elements. The System class also contains the advance() method, which executes a single micro-step. The generated code includes a default main() method that calls advance() in a for-ever loop.

The EnvSensor class is used to sense the environment for input. It updates the event-related snapshot elements (Q, Qa, IE, IEa or Ia) with environmental input, as directed by template-parameter values. Environmental input is sensed on standard input by default. The default generated code prompts for input on standard input at the beginning of every macro-step. The prompt provides a list of all environmental events in the model and the user chooses which of these events will be generated. We also wanted to ensure it would be possible to add code for sensing the environment from other sources (i.e., a GUI, or on sockets). For

this reason the EnvSensor class is generated separately so that all environmental sensing can occur in a single module. To add code for sensing the environment, all the user has to do is provide an implementation of the senseEnv() method in the EnvSensor class, which returns a set of events that have been sensed. This method will be called at the appropriate time by the generated code as per the notation's template-semantics description. Alternatively, the System class includes a method generateEvent(string name), which generates an event that will be sensed from the environment at the start of the next macro-step. This allows events to be generated from the environment asynchronously with the execution of the generated code. The generated events are buffered during a macro-step and will be sensed at the start of the next macro-step.

Output events are stored in the O snapshot element, which is represented by the Outputs object stored in each HTS object. Output events can be accessed by manipulating the PointersToO object, which stores references to each local Outputs object, or by calling the checkOutputEvent(string event) method of the System class. The checkOutputEvent() method will check if the event is in any of the Outputs objects for the user.

3.3 Scheduling and Execution Example

To make things more concrete, consider Figure 3.6, which depicts the composition tree of an example model. This model has a **homogeneous** composition tree in which all composition operators in the model are of the same type. The model consists of three interleaved HTSs (M1, M2, M3). Because the machines are interleaved, only a single transition per micro-step will ever be executed, even if multiple machines have transitions that are ready to execute in that step. The rest of this section describes the scheduling and execution of transitions in this example model.

As was stated above, the first phase in executing a micro-step is to do an enabledness check over the model's composition tree. The parameters of the enabled_check() method encode the enabledness information that will flow up the

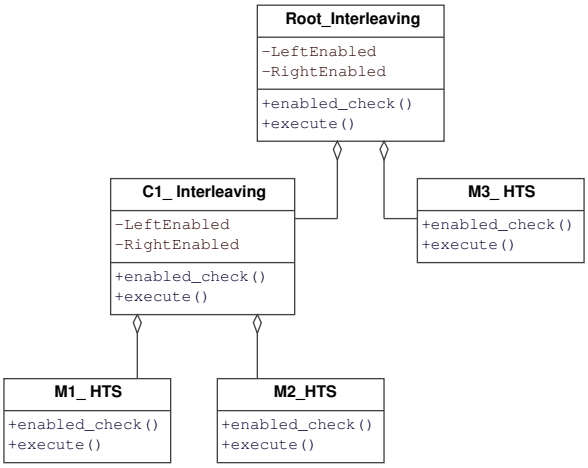


Figure 3.6: Interleaving Example

tree. In the case of our example model, which is composed entirely of interleaving composition, the only parameter is an enabled flag. This flag indicates whether the operator's subtree has an enabled transition. Each operator stores both of its children's enabled flags in member variables for use during the execution stage. This is the purpose of the member variables (left/right)Enabled in the composition-operator classes (Root_Interleaving and C1_Interleaving) shown in Figure 3.6.

The pseudocode in Figure 3.7 shows the enabled_check() method used by interleaving operators. In Phase 1, the component recursively calls enabled_check() on its left and right children and stores its children's enabled flags in member variables (leftEnabled, rightEnabled). In Phase 2, the operator passes its own enabledness information on to its parent by setting the enabled parameter (line 6). In the case of interleaving, the enabled parameter is set to the disjunction of the children's enabled flags: if either child has an enabled transition then the operator is itself enabled. For example, assume HTSs M1 and M2 have enabled transitions but HTS M3 is stable. During enabledness checking, component C1 would have set both leftEnabled and rightEnabled to true, as both of its children are enabled HTSs. C1 would then tell the root that it is enabled, causing the root to set its leftEnabled member variable to true. Since the root's right component is the stable HTS M3, the root's rightEnabled variable would be set to false.

```
{Member Variables}
VAR bool leftEnabled, rightEnabled
1: enabled_check(bool enabled)
2: {Phase 1: find the enabledness information for our left and right children}
3: left.enabled_check(leftEnabled)
4: right.enabled_check(rightEnabled)
5: {Phase 2: pass pertinent enabledness information to our parent}
6: enabled = leftEnabled ∨ rightEnabled

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

Figure 3.7: Interleaving Composition Scheduling

An HTS reacts to enabled_check() calls in an entirely different fashion. It does an enabledness check over its transitions and sets the enabled flag if it finds an enabled transition. This is described in detail in Section 3.4.1.

The pseudocode at the bottom of Figure 3.7 shows the execute() method used by interleaving operators. If both child components are enabled, the operator randomly chooses one to execute (lines 2-4). Otherwise, on lines 5-9, if only one child component is enabled, the operator instructs it to execute. For example, assume again that HTSs M1 and M2 have enabled transitions but that machine M3 is stable. At the end of the enabledness phase, the root's leftEnabled variable would be set to true and its rightEnabled variable would be set to false. Thus, during the execution phase, the root's execute() method would call its left component C1's execute() method (line 6). C1 has both its leftEnabled and rightEnabled variables set to true, so its execute() method would randomly choose one of its two children (M1 or M2) to execute (lines 3 and 4).

In contrast to the operator's execute() method, the HTS execute() method actually executes an enabled transition, according to the apply (i.e., next_XX)

template parameters. This is described in detail in Section 3.4.1.

In this simple example, it would appear as if the general architecture of enabledness information flowing up the tree and execution decisions flowing down is overly complex and inefficient. When we look at models where the composition tree is not homogeneous, it will become clear why this methodology is required. We will also introduce some optimizations to the general architecture in Section 3.8 that help improve efficiency if a model uses homogeneous hierarchies of certain composition operators.

3.4 Mapping Homogeneous Compositional Hierarchies to Java

We have shown above how a simple composition operator (interleaving) maps to Java when the composition tree contains only operators of a single type (i.e., the composition tree is homogeneous). We now look at how the rest of the composition operators map to Java under the assumption of a homogeneous hierarchy. This assumption is made purely to simplify the explanations of scheduling and execution. The code generator itself makes no distinction between models with heterogeneous vs. homogeneous composition hierarchies. The changes to the algorithms for homogeneous hierarchies, so that they support heterogeneous hierarchies, will be described in Section 3.5.

In the following discussion, our pseudocode aims to match exactly the semantics of the template-semantics' composition operators. This is significant, because these operators often involve nondeterministic choices. Because Java's model of computation is inherently deterministic, we simulate these choices using a random-number generator, in an attempt to simulate the operator's nondeterminism. This type of pseudo-nondeterministic source code is not useful for a deployable implementation, but it can be useful when simulating the model, in that simulation results may help the modeller decide how to resolve the nondeterminism. Our CGG tool has an option to create code generators that generate pseudo-nondeterministic code (using

random-number generators) or to create code generators that generate deterministic code. The following sections describe the pseudo-nondeterministic Java code. Chapter 4 describes the mechanisms we employ to remove this nondeterminism from the generated Java.

The first part of this section describes the basic HTS enabled_check() and execute() pseudocode for a model containing no composition operators (i.e., the model is a single HTS). Then, for each composition operator, we describe the operator's own enabled_check() and execute() methods as well as describe the changes to the basic HTS enabled_check() and execute() pseudocode that are needed to support the new operator.

3.4.1 Single HTS

The simplest homogeneous composition tree is the one containing no composition operators. The tree consists of a single HTS. In these models, enabled_check() finds and stores all enabled transitions, and execute() selects one of them to execute. We will show how to modify the basic HTS enabled_check() and execute() methods to accommodate the composition operators as we introduce each composition operator class in the following sections.

Each HTS in a model is translated into its own Java class. Each HTS class can be broken down into four major segments:

- constant enumerated-type declarations
- template-semantics-snapshot and member-variable declarations
- enabled_check() method declaration
- execute() method declaration

The constant enumerated-type declarations capture static structural information about the HTS, such as the sets of states and transitions. Java 1.4 has no native support for enumerated types, but they can be simulated by a series of Integer constants. The HTS class has one such unique constant for every transition and state in the HTS.

VAR Vars AV
VAR AuxAV AVa
VAR Queues Q
VAR AuxQ Qa
VAR Events IE
VAR AuxIE IEa
VAR AuxI Ia
VAR States CS
VAR AuxCS CSa

Figure 3.8: Snapshot Element Declarations

The template-semantics-snapshot elements are declared in each HTS class as shown in Figure 3.8. Member variables are declared only for those snapshot elements whose associated template-semantics parameter value is not set to n/a. The snapshot elements for AV, AVa, Q, and Qa are shared among the HTSs in the model; as such, they are translated into global objects which are shared amongst all HTS objects in the generated code. Each HTS stores a reference to each global element in a member variable named after the element. The rest of the snapshot elements are local to each HTS and are translated into member variables storing local objects. In the remainder of this section we describe how these variables are manipulated by various HTS methods. You can ignore the class types of the snapshot elements. They implement an appropriate data type (set, queue, set of queues, etc.) as specified by the next_XX and reset_XX parameters for each element XX.

In a composition hierarchy containing no composition operators, an HTS's enabled_check() method (Figure 3.9) simply finds, and stores in the member-variable enabled_transitions, the set of highest-priority enabled transitions. A transition's enabledness depends on the three enabling template parameters (en_state, en_cond, and en_events), which check if the transition's source state, guard condition, and, triggering events, respectively, hold in the current snapshot (line 6). Finding the set of highest-priority enabled transitions is done by checking each transition, in descending order of priority (line 4), and if a transition is found to be enabled (lines 6-10), then the checking loop will exit as soon as a transition with lower priority is encountered (line 12). This is a simple optimization that eliminates the

```
   {Member Variables}
   VAR set enabled_transitions
 1: enabled_check(bool enabled)
 2: max_pri_seen=-1
 3: {find the the set of priority-enabled transitions}
 4: for each transition τ in descending order of priority do
 5:    if pri(τ)>=max_pri_seen then
 6:       if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
 7:          max_pri_seen = pri(τ)
 8:          enabled=true
 9:          enabled_transitions.add(τ)
10:       end if
11:    else
12:       break {lower priority transition encountered, stop the checking loop}
13:    end if
14: end for
```

Figure 3.9: Basic HTS Enabledness Check

need to check all transitions in the HTS. When a high-priority enabled transition is found, the enabled parameter is set to true (line 8), to notify the HTS's parent of the enabled transition, and the transition is stored in the member-variable enabled_transitions (line 9) for use during the execute() method.

In a composition hierarchy containing a single HTS, the execute() method (Figure 3.10) picks one of the highest-priority enabled transitions stored in the enabled_transitions member variable (which was stored during enabled_check()) (line 3). The chosen transition is executed according to the Java implementations of the ten next_XX template parameters (lines 5-13). The XX_NEXT() procedures in the pseudocode represent inline code substitutions of the Java implementations of the next_XX template parameters (i.e., the generated code does not contain these procedures, instead next_XX's Java implementation is substituted wherever XX_NEXT is used in the pseudocode). The Java implementation of each next_XX parameter value manipulates the data structure used to implement the parameter's corresponding snapshot element. Each parameter value has a straightforward Java implementation. For example, the IE member variable could be implemented as a set of events, and the Java implementation of next_IE could remove the transition's triggering events from IE and add the transition's generated events to IE. In

```
   {Member Variables}
   VAR list enabled_transitions
 1: execute()
 2: { Phase 1: Select an enabled transition to execute}
 3: exe_trans = pick a random transition from enabled_transitions
 4: { Phase 2: Execute the selected transition}
 5: CS_NEXT(exe_trans)
 6: CSa_NEXT(exe_trans)
 7: AV_NEXT(exe_trans)
 8: AVa_NEXT(exe_trans)
 9: Q_NEXT(exe_trans)
10: Qa_NEXT(exe_trans)
11: IE_NEXT(exe_trans)
12: IEa_NEXT(exe_trans)
13: Ia_NEXT(exe_trans)
14: O_NEXT(exe_trans)
```

Figure 3.10: Basic HTS Execution

another semantics, the implementation could clear the IE set.

3.4.2 Parallel

A homogeneous hierarchy of parallel operators has the simplest mapping to Java. Parallel composition is much like interleaving. It differs only in the case where both child components are enabled. Parallel composition allows both child components to execute transitions in the same micro-step, whereas interleaving chooses one of the two child components to execute. Thus, the algorithms for enabledness and execution are very similar to those seen in Section 3.3. In fact, the pseudocode for parallel composition's enabled_check() is identical to that for interleaving composition. Parallel composition's execute() method (Figure 3.11) allows both left and right child components to execute if they are both enabled (lines 2-6).

This pseudocode shows how we simulate parallel execution in Java (lines 3-6). Parallel execution is simulated by executing the two children one after the other, the left child first, followed by the right child. Since the only information shared amongst HTSs are variable values, this ordering will be equivalent to executing the children in parallel, as long as we ensure that the variable assignments made by

```
 1: execute()
 2: if leftEnabled ∧ rightEnabled then
 3:    begin parallel simulation
 4:       left.execute()
 5:       right.execute()
 6:    end parallel simulation (resolve shared vars)
 7: else if leftEnabled then
 8:    left.execute()
 9: else if rightEnabled then
10:    right.execute()
11: end if
```

Figure 3.11: Parallel Composition Execution

the left child are not visible to the right child until after the right child finishes its execution. This is done by caching, for each shared variable, the old variable value and each parallel transition's new value (if it assigns the variable a new value). At the end of the micro-step, all HTSs must agree on one of the cached values to be the new value for each shared variable. The template-semantics parameter resolve indicates how to choose which value a shared variable will take on after parallel assignments are made to it (line 6). This is an expensive process and can be disabled if a model's parallel transitions do not reference shared variables (this can be automatically detected).

No changes to the basic HTS enabled_check() and execute() algorithms given in Section 3.4.1 are needed to support parallel operators. The operators in the following sections will require changing these algorithms.

3.4.3 Interrupt

Interrupt composition allows only one of its child components to execute, until an interrupt transition transfers control to the other child, after which only the other child can execute until an interrupt transition transfers control back to the first child. Thus, interrupt only checks one of its children for enabledness. Each interrupt operator in the model has a distinguished set of interrupt transitions associated with it, called ourInterrupts, which it uses to transfer control between its child components.

An interrupt-composition operator must do extra work when one of these interrupt transitions executes. As a result, the operator needs to know when an interrupt transition is enabled. The operator also needs to know the priority value of its children's highest-priority enabled transition. The operator uses this priority value to determine whether an interrupt transition has highest priority and should execute instead. These two extra pieces of enabledness information flow up the composition tree in two new parameters (interrupts and max_pri) to the enabled_check() method.

Interrupt is the only composition operator that uses the model's priority scheme. In any micro-step, an interrupt operator will execute the highest-priority enabled transition in its subtree (interrupt or noninterrupt). The other operators use the priority scheme only at the HTS level, for finding the highest-priority enabled transitions. At the level of composition, the other operators use nondeterminism to choose which transition(s) or child to execute if more than one is enabled. The reason for this distinction is that interrupt composition doesn't represent a traditional form of concurrency. It instead combines two components by explicitly defining how control will pass between them via interrupt transitions. Thus, the component rooted at an interrupt operator can be thought of as one heavy-weight HTS having a single priority scheme.

Although interrupt transitions are associated with an interrupt operator, via the set ourInterrupts, the snapshot elements needed to check an interrupt transition's enabledness or to execute an interrupt transition are stored in its source states' HTS. Thus, the enabledness checking and execution of an interrupt transition is performed by its source states' HTS. That is, we determine which HTS is associated with each interrupt-transition's source state, and we add the interrupt transition to the HTS's check for enabled transitions; and the HTS is also responsible for executing the interrupt transition if it is, later, ordered to do so by its parent. Even interrupt transitions whose source states are composition operators are handled in this fashion. If an interrupt transition's source state is a composition operator, then the enabledness checking for the transition is performed by all HTSs that descend from the source composition operator, and the interrupt transition's execution is

```
  {Member Variables}
  VAR set enabled_transitions
 1: enabled_check(set interrupts,int max_pri,bool enabled)
 2: max_pri_seen=-1
 3: {find the the set of priority-enabled transitions}
 4: for each transition τ in descending order of priority do
 5:    if pri(τ)>=max_pri_seen then
 6:       if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
 7:          max_pri_seen = pri(τ)
 8:          if τ is an interrupt transition then
 9:             interrupts.add(τ)
10:          else
11:             enabled=true
12:             max_pri=pri(τ)
13:             enabled_transitions.add(τ)
14:          end if
15:       end if
16:    else
17:       break {lower priority transition encountered, stop the checking loop}
18:    end if
19: end for
```

Figure 3.12: HTS Enabledness Check Under Interrupt

performed by one of the descendant HTSs.

HTS enabled_check() Supporting Interrupt

An HTS that has an ancestor interrupt operator responds differently to an enabled_check() call compared to an HTS whose ancestors are all interleaving or parallel operators. The enabled_check() method with support for interrupt operators (Figure 3.12) still sets the enabled parameter, but now it also needs to set the two new parameters (interrupts, and max_pri). Any enabled interrupt transition that the HTS finds is added to the interrupts parameter (line 9). The max_pri parameter is set to the highest-priority enabled noninterrupt transition (line 12). The enabled parameter is set to true only if a noninterrupt transition is enabled (line 11), and the enabled_transitions member variable only stores enabled noninterrupt transitions (line 13). This is because interrupt transitions are scheduled for execution by interrupt operators. Thus, the only transitions an HTS can au-

```
   {Member Variables}
   VAR bool leftEnabled, rightEnabled
   VAR set leftInterrupts, rightInterrupts
   VAR integer leftMaxPri, rightMaxPri
   VAR {left,right} hasControl
 1: enabled_check(set interrupts,int max_pri,bool enabled)
 2: if hasControl==left then
 3:    {Phase 1: find the enabledness information for the active child}
 4:    left.enabled_check(leftInterrupts,leftMaxPri,leftEnabled)
 5:    {Phase 2: pass pertinent enabledness information to our parent}
 6:    interr_enabled = leftInterrupts ∩ ourInterrupts≠ Ø
 7:    noninterr_enabled = leftEnabled
 8:    enabled = interr_enabled ∨ noninterr_enabled
 9:    maxInterrPri = maxPri(leftInterrupts ∩ ourInterrupts)
10:    max_pri=max(leftMaxPri, maxInterrPri )
11:    interrupts = leftInterrupts
12: else
13:    {Symmetric case when the right component is active}
14: end if
15:
```

Figure 3.13: Interrupt Composition Enabledness Check

tonomously choose to execute, when allowed to choose the executing transition, are noninterrupt transitions.

Interrupt's enabled_check()

As with the other composition operators, an interrupt operator needs to store in member variables the enabledness information of its left and right children. Thus, an interrupt-operator class has add four new member variables: leftInterrupts, rightInterrupts, leftMaxPri, and rightMaxPri. The algorithm for interrupt's enabled_check() is shown in Figure 3.13. The variable hasControl stores which child component is active and can execute transitions. The pseudocode for the left child being active is shown on lines 2-11. The pseudocode for the right child being active is symmetric. In Phase 1, the interrupt operator asks the active child for its enabledness information by recursively calling its enabled_check() method. Just as with interleaving and parallel, Phase 2 communicates the operator's own enabledness information to its parent by setting the parameters. On line 11, the interrupts

parameter is set to contain all of the enabled interrupt transitions detected by the active child, including those associated with other operators. The operator's own enabledness is communicated to its parent via the enabled parameter (line 8). An interrupt operator is enabled when one of its own interrupt transitions is enabled (interrupt case) or when the active child is enabled (noninterrupt case). The noninterrupt case (line 7) occurs when the active child's enabled flag is set in Phase 1. The interrupt case (line 6) occurs when an interrupt transition in the operator's set ourInterrupts is enabled in the active child. The enabled parameter is set to the disjunction of these two cases. An interrupt operator must set the max_pri parameter to the priority value of the highest-priority enabled transition in its composition subtree. The highest-priority transition is either one of the operator's interrupt transitions or it is the highest-priority enabled transition in the active child. Line 9 finds the highest-priority enabled interrupt transition using the function maxPri(); maxPri() takes a set of transitions and returns the priority value of the highest-priority transition in this set. Line 10, then, determines the priority value of the subtree's highest-priority enabled transition by comparing the operator's highest-priority interrupt transition with the active child's highest-priority enabled transition, and passes the result to the operator's parent (by setting the max_pri parameter). When enabled_check() finishes execution, the interrupt operator has enough information to make an execution decision.

Interrupt's execute()

An interrupt operator's execute() method, assuming that a model only uses interrupt composition, is shown in Figure 3.14. The execute() method contains a parameter, interr_trans, that represents a scheduling constraint imposed on this operator by an ancestor operator. The constraint is an order to execute the interrupt transition indicated by the interr_trans parameter. Thus, there are three cases to an interrupt-composition operator's execute() method:

- **Case 1:** Execute an interrupt transition imposed by an ancestor interrupt operator, as dictated by the interr_trans parameter
- **Case 2:** Execute one of the operator's own enabled interrupt transitions

```
{Member Variables}
VAR bool leftEnabled, rightEnabled
VAR set leftInterrupts, rightInterrupts
VAR integer leftMaxPri, rightMaxPri
VAR {left,right} hasControl
 1: execute(transition interr_trans)
 2: {Case 1: execute an interrupt ordered by an ancestor }
 3: if interr_trans is not null then
 4:    if hasControl==left then
 5:       left.execute(interr_trans)
 6:    else
 7:       right.execute(interr_trans)
 8:    end if
 9: else
10:    {Case 2, 3: make a local execution decision}
11:    if hasControl==left then
12:       if  maxPri(leftInterrupts ∩ ourInterrupts) == leftMaxPri then
13:          choice = randomly choose interr or non-interr
14:       else if  maxPri(leftInterrupts ∩ ourInterrupts) > leftMaxPri then
15:          choice = interr
16:       else
17:          choice = non-interr
18:       end if
19:       if choice==interr then
20:          trans = choose the highest priority transition in leftInterrupts ∩ ourInterrupts
21:          left.execute(trans)
22:          hasControl = right
23:          clear left components CS snapshot
24:       else
25:          left.execute(null)
26:       end if
27:    else
28:       {Symmetric case for the right side having control}
29:    end if
30: end if
```

Figure 3.14: Interrupt Composition Execution

- **Case 3:** Execute an enabled transition in the active child (which is not one of the operator's own interrupt transitions).

The first case occurs if the interr_trans parameter is set, indicating that an ancestor operator has decided to schedule the specified interrupt transition for execution. The operator follows its ancestor's directive and tells the active child to execute the transition in interr_trans by passing it as a parameter to the active child's execute() method (lines 5 and 7). Cases 2 and 3 occur if interr_trans is null, indicating that the operator can make its own execution decisions. The choice of whether to execute case 2 or case 3 is based on which enabled transition in the operator's subtree has the highest priority. If one of the operator's enabled interrupt transitions has highest priority, then the operator executes case 2, otherwise it executes case 3 (lines 12-18). This decision, based on transition priority, is why the priority value of the highest-priority enabled transition is passed up the tree during enabled_check(). If the highest-priority interrupt transition has equal priority to the highest-priority transition in the active child, then the operator randomly chooses which type of transition to execute (lines 12, 13). Executing an interrupt transition in case 2 involves switching hasControl to the other child component (line 22) and clearing the currently executing component's current states (CS) snapshot element (line 23). The operator must also set the interrupt parameter when calling execute() on the active child, to force the child to execute the chosen interrupt transition (line 21). Case 3 is carried out by calling execute() with a null parameter (line 25), imposing no constraints on the active child's execution. Case 3 allows the active child to make its own execution decision.

HTS execute() Supporting Interrupt

To support interrupt operators, the HTS execute() method is modified to process the interr_trans parameter. This parameter indicates that an ancestor interrupt operator has decided to schedule the interrupt transition in interr_trans for execution. An outline of the changes to the HTS execute() method that are required to process this parameter are shown in Figure 3.15. An HTS reacts to the interr_trans constraint by executing the interrupt transition in interr_trans (line 4). If interr_trans

```
   {Member Variables}
   VAR list enabled_transitions
1: execute(transition interr_trans)
2: { Phase 1: Select an enabled transition to execute}
3: if interr_trans is not null then
4:    exe_trans = interr_trans
5: else
6:    exe_trans = pick a random transition from enabled_transitions { a high-priority enabled
      transition}
7: end if
8: { Phase 2: Execute the selected transition}
9: ...
```

Figure 3.15: HTS Execution Under Interrupt

is null, the HTS executes a randomly selected enabled high-priority noninterrupt transition (line 6), just as it did in the basic HTS execute() pseudocode.

Interrupt is the first composition operator to use the notion of scheduling constraints in the execute() method. Interrupt's scheduling constraint is the most straightforward: it is an order to execute a specific transition. When a parent imposes this constraint on a child, the child component has no room to make any decisions. Some of the other operators impose more general constraints that allow the child operator some flexibility in making decisions about which transitions to execute. However, the general framework for execute() is still the same: the operator responds to all ancestor's constraints and imposes those plus its own constraints on its descendants.

3.4.4 Sequence and Choice

The code generated for both sequence and choice operators is very similar to that generated for interrupt. However, these operators do not have interrupt transitions, so there is no need for the extra enabledness information and parameters that interrupt composition required.

In choice composition, a random decision is made at the start of execution about which child component will be active, and this decision persists throughout

the execution. Thus, the enabled_check() and execute() methods only refer to the chosen component.

In sequence composition, the first child component executes to completion and then the second child component executes to completion. This is equivalent to the operator executing a single interrupt transition from the final state of the first child component to the initial state of the second child component. Thus, a finished flag must be added to a sequence operator's enabledness information. In an HTS, this flag is set to true if the HTS reaches a final state. In a sequence-composition operator, this flag is set to true if both of its child components have set their finished flags. A sequence operator's enabled_check() and execute() methods are interested only in the active component (just as in interrupt composition) and they monitor the active child's finished flag. When the initially-active-child component's finished flag is set, the execute() method switches control to the other child component. When the second child's finished flag is set, the sequence operator itself indicates that it has finished execution. Both of these operator's execute() and enabled_check() methods are based on straight-forward modifications to interrupt's execute() and enabled_check() methods. We don't show the pseudocode for choice or sequence operators.

3.4.5 Environmental Synchronization

Environmental synchronization (sync) introduces the idea of sync events. Each environmental sync class has a set of designated sync events, called syncEvents. If both the left and right child components have enabled transitions that are triggered by the same event in syncEvents, then these transitions can execute in parallel. Otherwise, the operator interleaves transitions that are not triggered by a sync event. To implement these semantics, the enabled_check() method needs a new parameter, env_events, which passes up the composition tree the set of sync events that trigger enabled transitions.

HTS enabled_check() Supporting Environmental Synchronization

```
   {Member Variables}
   VAR set enabled_transitions
   VAR map env_trans
 1: enabled_check(set env_events,bool enabled)
 2: max_pri_seen=-1
 3: {find the the set of priority-enabled transitions}
 4: for each transition τ in descending order of priority do
 5:    if pri(τ)>=max_pri_seen then
 6:       if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
 7:          max_pri_seen = pri(τ)
 8:          if τ is triggered by an env sync event e then
 9:             env_events.add(e)
10:             env_trans.put(e,τ) {put τ in the map under key e}
11:          else
12:             enabled=true
13:             max_pri=pri(τ)
14:             enabled_transitions.add(τ)
15:          end if
16:       end if
17:    else
18:       break {lower priority transition encountered, stop the checking loop}
19:    end if
20: end for
```

Figure 3.16: HTS Enabledness Check Under Environmental Sync

The HTS enabled_check() method needed to support environmental-synchronization operators is shown in Figure 3.16. The method has a new parameter, env_events, which passes up the composition tree the sync events which trigger enabled transitions. The enabled_check() method searches for and stores enabled transitions that are triggered on sync events (lines 8-10) in a new member variable, env_trans. When an enabled transition is triggered by a sync event, that sync event is added to the env_event parameter (line 9) and the enabled transition is placed in the env_trans map, keyed on the sync event which triggered it (line 10). The HTS stores these sync transitions, so that during the execution phase, if an ancestor constrains the HTS to execute an enabled transition triggered on a particular sync event, the HTS can quickly retrieve a list of such transitions by indexing into the env_trans map with the specified sync event. The enabled_transitions member variable now contains only enabled transitions that are not triggered on a sync event (line 14).

```
{Member Variables}
VAR bool leftEnabled, rightEnabled
VAR set leftEnv_Events, rightEnv_Events
1: enabled_check(set env_Events,bool enabled)
2: {Phase 1: find the enabledness information for our left and right children}
3: left.enabled_check(leftEnv_Events,leftEnabled)
4: right.enabled_check(rightEnv_Events,rightEnabled)
5: {Phase 2: pass pertinent enabledness information to our parent}
6: unsync = leftEnabled ∨ rightEnabled
7: sync = (leftEnv_Events ∩ rightEnv_Events ∩ syncEvents)≠ Ø
8: enabled = un-sync ∨ sync
9: env_Events = ( leftEnv_Events ∩ rightEnv_Events ∩ syncEvents) ∪
                leftEnv_Events/syncEvents ∪ rightEnv_Events/syncEvents
```

Figure 3.17: Environmental Sync Composition Enabledness Check

Environmental Synchronization's enabled_check()

In the same manner as the other operators, an environmental sync operator stores its children's enabledness information in member variables. We add two member variables (left/right)Env_Events to record the sync events that trigger enabled transitions in the left and right child, respectively.

The enabled_check() algorithm for an environmental-synchronization operator, assuming that a model only uses environmental-synchronization composition, is shown in Figure 3.17. Similar to the enabled_check() methods for the other operators, Phase 1 of an environmental-synchronization operator's enabled_check() method recursively calls its children's enabled_check() methods. Phase 2 differs in that an environmental-sync operator is enabled if (1) either of its children has an enabled transition not triggered by one of the operator's sync events (the unsync case), or if (2) both children have one or more transitions that are triggered by the same event in syncEvents (the sync case). The unsync case is encoded on line 6 as the disjunction of the children's enabled flags. The sync case is encoded on line 7; it checks if there exists a sync event in the set syncEvents that the right and left children are both enabled on. The operator's enabledness is the disjunction of these two cases. The env_Events parameter is set so that ancestor operators are aware of which sync events trigger transitions in the environmental-sync operator's

subtree (line 9). Env_Events is set to all sync events passed up from the operator's children that the operator is willing to execute transitions triggered on (if ordered to by an ancestor operator during execute()). These events include any event not in syncEvents that was passed up from the operator's child components (the operator can interleave transitions triggered on these events) or any event in syncEvents that enables transitions in both children (the operator can execute a synchronization on these events). Events in syncEvents that enabled transitions in only one child are not added to env_Events. This is because the operator does not allow only one child to execute transitions triggered by one of its sync events (it must synchronize both children on a sync event).

Environmental Synchronization's execute()

The execute() method for environmental sync is shown in Figure 3.18. It is made up of three cases:

1. Execute a synchronization imposed by an ancestor component (ancestral constraint case)
2. Execute a synchronization on one of the operator's sync events (local sync case)
3. Let a child component execute transition(s) not triggered by one of the operator's sync events (unsync case)

Consider the model show in Figure 3.19 whose composition tree contains two environmental sync operators env1 and env2, where env1 is the parent of env2. Env1 synchronizes on events a1 and a2, and env2 synchronizes on events a1 and a3. This model can exhibit each of the three cases above within env2's execute() method. The first case occurs when env1 chooses one of its sync events and calls env2's execute() method setting the env_event parameter to the chosen event. This parameter constrains env2 to consider only transitions triggered by the event in env_event. There are four sub-cases of case 1 (lines 3-17) that show the different ways that env2 can react to this constraint. In the first sub-case (lines 4-8), the imposed synchronization happens to involve one of the operator's own sync events, and the

```
{Member Variables}
VAR bool leftEnabled, rightEnabled
VAR set leftEnv_Events, rightEnv_Events
 1: execute(event env_event)
 2: {Case 1: execute a synchronization imposed by an ancestor }
 3: if env_event is not null then
 4:   if env_event∈leftEnv_Events ∧ env_event∈rightEnv_Events ∧ env_event∈syncEvents then
 5:     begin parallel execution simulation
 6:       left.execute(env_event)
 7:       right.execute(env_event)
 8:     end parallel execution simulation
 9:   else if env_event∈leftEnv_Events ∧ env_event∈rightEnv_Events ∧ env_event∉syncEvents
      then
10:     compToExecute = choose one of the left or right components to execute
11:     compToExecute.execute(env_event)
12:   else if env_event ∈leftEnv_Events ∧ env_event∉syncEvents then
13:     left.execute(env_event)
14:   else if env_event ∈rightEnv_Events ∧ env_event∉syncEvents then
15:     right.execute(env_event)
16:   end if
17: else
18:   {Case 2, 3: make a local execution decisions}
19:   sync_set = leftEnv_Events ∩ rightEnv_Events ∩ syncEvents
20:   if  sync_set ≠ Ø ∧ (leftEnabled ∨ rightEnabled)  then
21:     choice = randomly choose one of sync or unsync
22:   else if sync_set ≠ Ø then
23:     choice = sync
24:   else if leftEnabled ∨ rightEnabled then
25:     choice = unsync
26:   end if
27:   if choice==sync then
28:     events = choose an event in sync_set
29:     begin parallel execution simulation
30:       left.execute(event)
31:       right.execute(event)
32:     end parallel execution simulation
33:   else if choice==unsync then
34:     if leftEnabled ∧ rightEnabled then
35:       compToExecute = randomly choose left or right child
36:       compToExecute.execute(null)
37:     else if leftEnabled then
38:       left.execute(null)
39:     else if rightEnabled then
40:       right.execute(null)
41:     end if
42:   end if
43: end if
```

Figure 3.18: Environmental Sync Composition Execution

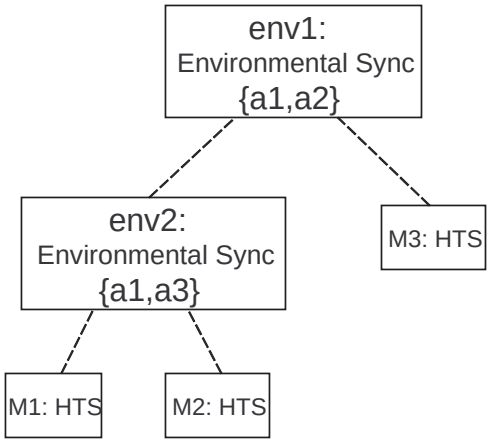


Figure 3.19: Environmental Synchronization Example

operator's left and right components are both ready to react to this event; thus, in this subcase, the operator can satisfy the ancestral constraint by instructing both of its child components to execute in parallel transitions triggered on env_event (lines 6 and 7). This sub-case would occur if env1 ordered a synchronization on a1, which is also a sync event for env2. Env2 reacts to this sync command by constraining both its children to execute in parallel on a1 (instructing its children to also synchronize on a1). The manner in which sync operators pass up enabledness information ensures that it will never be the case that env1 orders a sync on a1 when only one of env2's children are prepared to execute on a1, (i.e., during enabled_check(), env2 will not pass a1 up to env1, unless both env2's children have enabled transitions triggered on a1). In the remaining three sub-cases of case 1, the event in env_event is not a designated sync event for this operator. These sub-cases would occur if env1 ordered a synchronization on a2, which is not in env2's sync set ({a1,a3}). As such, these three sub-cases are unsync cases relative to env2, so only one of its child components can execute and react to a2.

Case 2 (lines 27-32) is the local sync case and would occur if env1 decided not to synchronize on one of its sync events and allowed env2 to execute unconstrained. If both env2's child components have enabled transitions that are triggered by one of env2's sync events, then env2 can choose to synchronize on this event and instruct

```
   {Member Variables}
   VAR list enabled_transitions
   VAR map env_trans
1: execute(event env_event)
2: { Phase 1: Select an enabled transition to execute}
3: if env_event is not null then
4:    exe_trans = pick a random transition from env_trans.get(env_event)
5: else
6:    exe_trans = pick a random transition from enabled_transitions
7: end if
8: { Phase 2: Execute the selected transition}
9: ...
```

Figure 3.20: HTS Execution Under Environmental Synchronization

both of its child components to execute transitions triggered by this event. Env2 instructs its children to synchronize by passing the chosen sync event as a parameter to each child's execute() method on lines 30 and 31.

Case 3 (lines 34-41) is the un-sync case, in which env2 is not constrained by env1 and it decides not to synchronize and allows one of its two child components to execute unconstrained (the children's execution is interleaved). Unconstrained execution is encoded by passing null as the parameter to the chosen child's execute() method (lines 36,38,40). If both sync and un-sync executions are enabled, the operator randomly chooses which case is executed (Line 19,20).

HTS execute() Supporting Environmental Synchronization

To support environmental-synchronization operators, the HTS execute() method is modified to process the new parameter, env_event, representing the constraint to execute a transition triggered on env_event (Figure 3.20). If the env_event parameter is set, then the HTS will retrieve from the env_trans map the list of enabled transitions that are triggered on env_event and randomly pick one of them to execute (line 4). If the env_event parameter is not set, then the HTS will execute an enabled transition not triggered on a sync event (a transition stored in enabled_transitions) (line 6).

3.4.6 Rendezvous Synchronization

Rendezvous composition looks very similar to environmental synchronization. In rendezvous, the synchronization event, also called the rendezvous event, is generated and reacted to in the same micro-step. Thus, we need to introduce the notion of a sending component that generates the sync event, and a receiving component that, in the same micro-step, reacts to the sync event's generation. Template semantics makes the assumption that a transition triggered on a rendezvous-sync event has only that sync event as a triggering event (i.e., only one triggering event).

HTS enabled_check() Supporting Rendezvous Synchronization

The HTS enabled_check() method needed to support rendezvous operators, is shown in Figure 3.21. The method has two new parameters: gen_events, which stores rendezvous sync events that are generated by enabled transitions, and trig_events, which store rendezvous sync events whose generation would cause some transition to become enabled. The HTS contains two member-variable maps: rend_GenTrans, which maps sync events to a list of enabled transitions that generate that event, and rend_TrigTrans, which maps sync events to a list of transitions that would be enabled if that event were to be generated. The gen_event parameter and rend_GenTrans map are used in the same manner as the env_event parameter and env_trans map are used to support environmental sync. If an enabled transition generates a sync event e, then e is added to the gen_events parameter, and the transition, keyed on e, is added to the rend_GenTrans map (lines 8-10).

The trig_events parameter and the rend_TrigTrans member variable are set in a unique fashion (lines 22-27). The trig_events parameter stores sync events whose generation would cause a transition in the HTS to become enabled. These rendezvous events trigger transitions that are not actually enabled, but would become enabled if that sync event were generated. Thus, the enabledness check on line 23, which checks transitions that could execute in a rendezvous synchronization, doesn't check the en_event condition because it's up the rendezvous operator (not the HTS) to match this transition up with another transition that generates the

```
{Member Variables}
VAR set enabled_transitions
VAR map rend_GenTrans, rend_TrigTrans
 1: enabled_check(set gen_events,set trig_events,bool enabled)
 2: max_pri_seen=-1
 3: {find the the set of priority-enabled transitions}
 4: for each transition τ in descending order of priority do
 5:    if pri(τ)>=max_pri_seen then
 6:       if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
 7:          max_pri_seen = pri(τ)
 8:          if τ generates a rend sync event e then
 9:             gen_events.add(e)
10:             rend_GenTrans.put(e,τ)
11:          else
12:             enabled=true
13:             max_pri=pri(τ)
14:             enabled_transitions.add(τ)
15:          end if
16:       end if
17:    else
18:       break {lower priority transition encountered, stop the checking loop}
19:    end if
20: end for
21: {check if we can react to rendezvous-sync events being generated in other HTSs}
22: for each transition τ triggered by a rendezvous event do
23:    if en_state(τ) ∧ en_cond(τ) ∧ pri(τ)>=max_pri_seen then
24:       trig_events.add(e)
25:       rend_TrigTrans.put(e,τ)
26:    end if
27: end for
```

Figure 3.21: HTS Enabledness Check Under Rendezvous

proper sync event. By definition, the enabling event will not be present in the snapshot elements; it is present only if the rendezvous operator chooses to execute the rendezvous in this micro-step. These potentially enabled rendezvous transitions are stored in the rend_TrigTrans map, keyed on the sync event that triggers the transition. The HTS uses the two member-variable maps to execute its part of a rendezvous synchronization, if instructed to do so by an ancestor operator.

```
{Auxiliary Variables}
VAR bool leftEnabled, rightEnabled
VAR set leftGen_Events, rightGen_Events
VAR set leftTrig_Events, rightTrig_Events
 1: enabled_check(set gen_events,set trig_events,bool enabled)
 2: {Phase 1: find the enabledness info for our children}
 3: left.enabled_check(leftGen_events,rightGen_events,leftEnabled)
 4: right.enabled_check(rightGen_events,rightTrig_events,rightEnabled)
 5: {Phase 2: pass pertinent enabledness info to our parent}
 6: gen_events = leftGen_events ∪ rightGen_events
 7: trig_events = leftTrig_events ∪ rightTrig_events
 8: {Determine our enabledness}
 9: leftSenderEnabled = leftGen_Events ∩ rightTrig_Events ∩ syncEvents ≠ ∅
10: rightSenderEnabled = leftTrig_Events ∩ rightGen_Events ∩ syncEvents ≠ ∅
11: sync = leftSenderEnabled ∨ rightSenderEnabled
12: unsync = leftEnabled ∨ rightEnabled
13: enabled = sync ∨ unsync
```

Figure 3.22: Rendezvous Composition Enabledness Check

Rendezvous' enabled_check()

Rendezvous operators require four new member variables. LeftGen_Events and rightGen_Events store the sync events that are generated by enabled transitions in the left and right components. Thus, these events identify available rendezvous senders. Member variables leftTrig_Events and rightTrig_Events store sync events that trigger transitions that would be enabled if that sync event were generated in the opposite component. Thus, these events identify available rendezvous receivers.

A rendezvous operator's enabled_check() method, assuming that the model contains only rendezvous operators, is shown in Figure 3.22. This method manipulates the member variables in the usual way: the left and right children are asked for their enabledness information, and the operator passes their combined results up to its parent operator (lines 3-7). The major difference between rendezvous sync and environmental sync is how the operator determines if it is enabled in the sync case. Rendezvous-sync requires that a sender of a sync event be enabled in one child component and that a receiver prepared, to receive the same event, be enabled in the other child component. Thus, a rendezvous sync case is enabled only

if one of the operator's sync events could be generated by one child component and can be received and reacted to in the other child component. This case is checked by testing if the intersection of (right/left)Gen_Events, (left/right)Trig_Events and syncEvents is not empty (lines 9-11).

Rendezvous' execute()

A rendezvous operator's execute() method is shown in Figure 3.23. It contains the same three cases as environmental sync's execute() method. In the first case (lines 2-20), the operator responds to ancestral constraints. The execute method has two parameters (triggerEvent and generateEvent), as two types of constraints that can be passed down from ancestor rendezvous operators. An ancestor can direct the operator to execute a transition that generates the event in generateEvent or to execute a transition that is triggered by the event in triggerEvent; it will never ask the operator to do both. When an ancestor imposes a rendezvous, it will ask the operator to perform only half of the rendezvous (the other half to be performed by the ancestor's other child). Only one of the operator's children ever reacts to an ancestral constraint, as rendezvous dictates that only one sender and one receiver can be involved in a rendezvous. In either case, the operator reacts to a constraint by passing the constraint on to the appropriate child component. Lines 3-10 choose the child component to execute a transition triggered on triggerEvent (i.e., if both child components are enabled on triggerEvent, then the operator randomly chooses one child to execute), and lines 12-19 choose the child component to execute a transition that generates generateEvent.

The second case in rendezvous' execution is when the operator receives no ancestral constraints and it chooses to rendezvous on one of its own sync events (lines 30-47). On line 31, the component chooses an event that it is prepared to rendezvous on (an event that has an enabled receiver in one component and an enabled sender in the other component). To execute a rendezvous on the chosen event, the operator sets the generateEvent parameter of the sender's execute() method, thereby instructing the sending child to execute a transition that generates the chosen event. The operator also sets the triggerEvent parameter of the receiver's

```
   VAR bool leftEnabled, rightEnabled
   VAR set leftGen_Events, rightGen_Events
   VAR set leftTrig_Events, rightTrig_Events
 1: execute(event triggerEvent,event generateEvent)
 2: if triggerEvent is not null {Case 1: Respond to ancestral constraints} then
 3:    if triggerEvent ∈ leftTrig_Events ∧ triggerEvent ∈ rightTrig_Events then
 4:       compToExecute = randomly choose left or right child
 5:       compToExecute.execute(triggerEvent,null)
 6:    else if triggerEvent ∈ leftTrig_Events then
 7:       left.execute(triggerEvent,null)
 8:    else if triggerEvnet ∈ rightTrig_Events then
 9:       right.execute(triggerEvent,null)
10:    end if
11: else if generateEvent is not null then
12:    if generateEvent ∈ leftGen_Events ∧ generateEvent ∈ rightGen_Events then
13:       compToExecute = randomly choose left or right child
14:       compToExecute.execute(null,generateEvent)
15:    else if generateEvent ∈ leftGen_Events then
16:       left.execute(null,generateEvent)
17:    else if generateEvent ∈ rightGen_Events then
18:       right.execute(null,generateEvent)
19:    end if
20: else
21:    {Cases 2,3: unconstrained execution}
22:    sync_set = ( leftGen_Events ∩ rightTrig_Events ∩ syncEvents ) ∪
                  ( rightGen_Events ∩ leftTrig_Events ∩ syncEvents )
23:    if sync_set ≠ Ø ∧ (leftEnabled ∨ rightEnabled) then
24:       choice = randomly choose one of sync or unsync
25:    else if sync_set ≠ Ø then
26:       choice = sync
27:    else if leftEnabled ∨ rightEnabled then
28:       choice = unsync
29:    end if
30:    if choice == sync {Case 2: Local rendezvous synchronization} then
31:       syncEvent = randomly choose an event from sync_set
32:       if (syncEvent ∈ leftGen_Events ∧ syncEvent ∈ rightTrig_Events) ∧
             (syncEvent ∈ rightGen_Events ∧ syncEvent ∈ leftTrig_Events) then
33:          sender = randomly choose left or right
34:       else if syncEvent ∈ leftGen_Events ∧ syncEvent ∈ rightTrig_Events then
35:          sender = left
36:       else
37:          sender = right
38:       end if
39:       if sender==left then
40:          {Right ordered to be triggered by syncEvent, Left ordered to generate syncEvent}
41:          begin parallel execution simulation
42:             left.execute(null,syncEvent)
43:             right.execute(syncEvent,null)
44:          end parallel execution simulation
45:       else if sender==right then
46:          {Omitted: Right to generate syncEvent, Left to be triggered by syncEvent}
47:       end if
48:    else if choice == unsync then
49:       {Case 3: Interleaving - Unchanged from env sync interleaving case}
50:    end if
51: end if
```

Figure 3.23: Rendezvous Composition Execution

```
    {Member Variables}
    VAR list enabled_transitions
    VAR map rend_GenTrans, rend_TrigTrans
 1: execute(event triggerEvent,event generateEvent)
 2: { Phase 1: Select an enabled transition to execute}
 3: if triggerEvent is not null then
 4:    exe_trans = pick a random transition from rend_TrigTrans.get(triggerEvent)
 5: else if generateEvent is not null then
 6:    exe_trans = pick a random transition from rend_GenTrans.get(generateEvent)
 7: else
 8:    exe_trans = pick a random transition from enabled_transitions
 9: end if
10: { Phase 2: Execute the selected transition}
11: ...
```

Figure 3.24: HTS Execution Under Rendezvous

execute() method, thereby instructing the receiving child to execute a transition triggered on the chosen event. Lines 40-44 show how this is accomplished if the left component is the sender of the rendezvous event and line 46 indicates the symmetric case in which the right component is the sender of the rendezvous event (the pseudocode for this case has been omitted). The unsync case (case 3) is unchanged from environmental synchronization's unsync case.

HTS execute() Supporting Rendezvous Synchronization

To support rendezvous operators, the HTS execute() method needs to process the two new parameters, triggerEvent and generateEvent, in the same manner that environmental sync's new parameter (env_event) was handled. This processing is shown in Figure 3.24. If the triggerEvent parameter is set, then the HTS retrieves from the rend_TrigTrans map the list of transitions that are triggered on triggerEvent and will randomly pick one of them to execute (line 4). If the generateEvent parameter is set, the HTS retrieves from the rend_GenTrans map the transitions that generate the event in generateEvent and randomly picks one of them to execute (line 6). If neither parameter is set, the HTS executes an enabled transition that is neither triggered by a sync event nor generates a sync event (i.e., a transition stored in enabled_transitions), as shown on line 8.

We have shown how each composition operator can be mapped to Java under the assumption that the composition tree contains only operators of that type. We will now look at how we modify the algorithms to support models that use more than one type of composition.

3.5 Mapping Heterogeneous Compositional Hierarchies to Java

The overall scheduling methodology, in which enabledness information flows up the tree and execution decisions flow down the tree, is unchanged in models that have **heterogeneous composition hierarchies**, containing more then one type of composition operator. In this section, we discuss how the algorithms for each operator's enabled_check() and execute() methods must change to accommodate heterogeneous composition hierarchies. We discuss the changes needed in the most general case, in which an operator has at least one ancestor composition operator of every type. Thus, the operator's enabled_check() method needs a parameter to pass up the composition tree enabledness information that every type of operator requires, and the operator's execute() method needs a parameter for every possible execution constraint another type of operator could impose. In practice, our code generator generates operator classes that have only those parameters that an ancestral composition operator requires. For example, if in the model's composition hierarchy a particular operator does not have an interrupt operator as an ancestor, then its enabled_check() method will not have a parameter for reporting the list of enabled-interrupt transitions and its execute() method will not have a parameter for the interrupt constraint.

3.5.1 Heterogeneous Enabledness Checking

In interleaving composition (Figure 3.6), the only information that is passed up the hierarchy is a boolean flag indicating that the component has a transition it would

like to execute. There are four composition operators that require extra enabledness information, beyond a simple enabled flag, to flow up the composition hierarchy. Interrupt composition needs to know the set of enabled interrupt transitions and it also needs to know the priority value of the highest-priority enabled transition. Environmental synchronization needs to know the set of sync events that trigger enabled transitions. Rendezvous needs two extra pieces of information: the set of sync events that are generated by enabled transitions and the set of sync events whose generation would cause some transition to become enabled. Sequence needs to know if a child has finished execution. Thus, in the most general case, in which all of the composition operators need to be supported, every node in the hierarchy must pass along and store the following information:

- A flag indicating a highest-priority enabled transition in the component.
- The priority value of the highest-priority enabled transition in the component.
- The set of highest-priority enabled interrupt transitions in the component.
- The set of environmental-sync events that trigger some highest-priority enabled transition in the component.
- The set of rendezvous-sync events whose generation would cause some highest-priority transition to become enabled.
- The set of rendezvous-sync events that are generated by some highest-priority enabled transition in the component.
- A flag indicating that the component has finished execution.

Thus, in general, there are a total of seven enabledness parameters that each composition operator could be responsible for collecting from its child components and passing on to its parent component.

Because a composition operator needs to store the enabledness information for both its left and right components, each operator needs two variables (left and right) for each of enabled_check()'s seven parameters. A 15th variable, hasControl, keeps track of which child component has control of the execution, for those composition operators (interrupt, sequence, choice) that coordinate sequential execution of their child components. Thus, in general, each node in the composition tree will need a

```
{Member Variables}
VAR boolean leftFinal, rightFinal
VAR boolean leftEnabled, rightEnabled
VAR integer leftMaxPri, rightMaxPri
VAR set leftRend_TEvents, rightRend_TEvents {T for Triggering events}
VAR set leftRend_GEvents, rightRend_GEvents {G for Generated events}
VAR set leftEnv_Events, rightEnv_Events
VAR set leftInterr, rightInterr
VAR {left, right} hasControl
 1: enabled_check(set rend_TEvents,set rend_GEvents,set env_Events,set interr_trans,boolean
    enabled,integer maxPri,bool final_state)
 2: {Phase 1: find the enabledness information for our left and right children}
 3: left.enabled_check(leftRend_TEvents,leftRend_GEvents,leftEnv_Events,leftInterr,leftEnabled,leftMaxPri,
    leftFinal)
 4: right.enabled_check(rightRend_TEvents,rightRend_GEvents,rightEnv_Events,rightInterr,rightEnabled,
    rightMaxPri,rightFinal)
 5: {Phase 2: pass pertinent enabledness information to our parent}
 6: enabled = OPERATOR SPECIFIC
 7: maxPri = OPERATOR SPECIFIC
 8: final_state = leftFinal ∧ rightFinal
 9: rend_TEvents = leftRend_TEvents ∪ rightRend_TEvents
10: rend_GEvents = leftRend_GEvents ∪ rightRend_GEvents
11: env_Events = leftEnv_Events ∪ rightEnv_Events
12: interr_trans = leftInterr ∪ rightInterr
```

Figure 3.25: Heterogeneous Enabledness Outline

maximum of 15 member variables, as shown at the top of Figure 3.25. An operator can leave out any variable if there is no ancestral operator that requires it to store this information.

An outline of enabledness checking in the most general case is shown in Figure 3.25. Just as in the homogeneous case, the operator stores its children's enabledness information in member variables in Phase 1 and uses this information to determine its own enabledness in Phase 2. The operator must store and pass on to its parent all seven pieces of enabledness information listed above. The only operator-specific part of the enabledness check is on lines 6 and 7. Each operator sets the enabled and maxPri parameters differently. The previous operator-specific sections describe how each operator sets the enabled parameter. The maxPri parameter is set to the priority value of the highest-priority enabled transition in the component's subtree.

We have shown how interrupt operators compute maxPri (Figure 3.13). For the other operators, the computation of maxPri is closely related to the computation of enabled. For example, interleaving composition sets enabled to the disjunction of leftEnabled and rightEnabled, and would therefore set maxPri to the maximum of leftMaxPri and rightMaxPri. It is straightforward to determine how the other operators would calculate maxPri by looking at how they calculate enabled.

Note that Figure 3.25 outlines a heterogeneous enabled_check() in which both of the operator's children are checked for enabled transitions. Because in interrupt, sequence, and choice composition, only one of the child components is ever active at a time, these composition operators check only the active component in Phase 1 and deal with only the active child's enabledness information in Phase 2.

3.5.2 Heterogeneous Execution

The execute() pseudocode for homogeneous composition hierarchies considered only the constraints that ancestral operators of the same type could impose on a given operator. The most general execution algorithm must handle all possible constraints that an ancestor of any type could impose on an operator, as outlined in Figure 3.26. There are four possible types of constraints, as represented by the four parameters to the execute() method. These parameters encode the following constraints:

- execute only enabled transitions that are triggered by env_Event
- execute a single enabled transition that generates rend_GEvent
- execute a single enabled transition that is triggered by rend_TEvent
- execute the interrupt transition in interrupt

An operator will receive at most one of the first three constraints at any time. The interrupt constraint can be set at anytime (i.e., it is possible to constrain an operator to execute an interrupt transition that is triggered on a particular sync event).

A constraint is a restriction on the types of decisions that the composition operator can make when determining whether the left or right child should execute. For

```
1: execute(event env_Event,event rend_GEvent,event rend_TEvent,transition interrupt)
2: {Case 1: schedule child components to execute based on constraints in the parameters }
3: if interrupt is non null then
4:    OPERATOR SPECIFIC
5: else if env_Event is non null then
6:    OPERATOR SPECIFIC
7: else if rend_GEvent is non null then
8:    OPERATOR SPECIFIC
9: else if rend_TEvent is non null then
10:   OPERATOR SPECIFIC
11: else
12:   {Case 2: if no constraints, schedule based on the operators semantics}
13:   OPERATOR SPECIFIC
14: end if
```

Figure 3.26: Heterogeneous Execution Outline

example, if env_Event is set, then the operator will select for execution only child component(s) that have some transition triggered by that event. How the operator decides which child component(s) to execute depends on the type of composition operator it is. For example, parallel composition would execute both left and right child components if they were both enabled on env_Event, whereas interleaving composition would execute one of the two children enabled on env_event. If no scheduling constraints are set, the operator is free to schedule and place constraints on its child components based on its own semantics, considering all of the enabledness information that was passed up from its children. This is case 2 of the generic execute() algorithm and it corresponds to cases 2 and 3 of the operator-specific execute() algorithms given in Section 3.4. The order in which the constraints are checked is significant. The interrupt constraint must be checked first as it is a more specific constraint than the other three: an interrupt constraint specifies exactly which single (interrupt) transition must be executed. The other three constraints (env_Event, rend_GEvent, rend_TEvent) can be checked in any order as only one of them can ever be set in a given micro-step (i.e., it is impossible to execute a rendezvous synchronization and an environmental synchronization in the same micro-step because rendezvous composition dictates that only one sending transition and one receiving transition can participate in a rendezvous synchronization).

Execution Priority in Heterogeneous Hierarchies

By allowing each unconstrained composition operator to randomly choose between executing its own transitions or allowing its child component(s) to execute unconstrained (cases 2 and 3 of most operator's execute() method), we ensure that the position of an operator in the composition tree has no effect on its execution priority. That is, even though an operator higher in the composition tree has its execute() method called first and thus has an earlier chance to make an execution decision, this higher operator could randomly decide to defer execution and allow a descendant operator to execute unconstrained. For example, a higher-level environmental sync operator can randomly decide not to synchronize and to allow one its children to execute unconstrained (allowing it to make its own execution decisions).

Chapter 4 introduces some techniques for eliminating nondeterminism that result in operators higher in the composition tree having higher execution priority. The only exception to this rule are interrupt operators, which use the transition priority scheme to determine whether a higher-level operator will defer execution to a lower-level operator. The reason for this distinction is that interrupt composition doesn't represent a traditional form of concurrency. Instead, it combines two components by explicitly defining how control passes between them via interrupt transitions. Thus, the component rooted at an interrupt operator can be thought of as one heavy-weight HTS sharing a single priority scheme.

3.6 Mapping Hierarchical Transition Systems to Java

This section shows the full HTS enabled_check() and execute() pseudocode that would be needed if an HTS had operators of every type as ancestors in a model's composition hierarchy. It combines the pseudocode modifications given for each composition operator above into one complete pseudocode description.

3.6.1 HTS Enabledness Checking

Figure 3.27 shows the full HTS enabled_check() pseudocode that supports all composition operators. The first phase of the algorithm (lines 8-31) involves finding the set of highest-priority enabled transitions. When an enabled transition is found, the relevant enabledness information for that transition is communicated to the operator's parent.

This communication is carried out by setting the appropriate parameter values, depending on the characteristics of the enabled transition. There are four pieces of information about an enabled transition that our parent may be interested in:

- Any rendezvous-sync events that the transition generates (rend_GEvents parameter)
- Any environmental-sync events that the transition is triggered on (evn_Events parameter)
- Whether the transition is an interrupt transition (interr_trans parameter)
- Whether the transition is a noninterrupt, nonrendezvous, nonenvironmental synchronization transition (enabled parameter)

An HTS also stores the enabled transition in one of its four member variables, based on the characteristics of the enabled transition, for later use during the execute() method.

Phase 2 is only required if this HTS has an ancestor that is a rendezvous operator. A rendezvous operator needs to know if a transition would become enabled if a particular rendezvous event were to be generated by another HTS in the current micro-step. If any such transitions are found, the HTS communicates the sync event it is triggered on to the HTS's parent (line 35). The HTS also stores the transition in the member variable rend_TTrans (Line 36) for potential use, later, in the execute() method.

3.6.2 HTS Execution

```
{Member Variables}
VAR set enabled_transitions
VAR map env_trans, rend_GTrans, rend_TTrans
 1: enabled_check(set rend_TEvents,set rend_GEvents,set evn_Events,set interr_trans,bool enabled,bool final_state)
 2: max_pri_seen=-1
 3: if  CS contains a final state then
 4:    final_state=true
 5:    return
 6: end if
 7: {Phase 1: find the enabledness information for the set of enabled transitions}
 8: for each transition τ in descending order of priority do
 9:    if pri(τ)>=max_pri_seen then
10:       if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
11:          max_pri_seen = pri(τ)
12:          if τ generates a rendezvous event e then
13:             rend_GEvents.add(e)
14:             rend_GTrans.put(e,τ)
15:          end if
16:          if τ is an interrupt transition then
17:             interr_trans.add(τ)
18:          end if
19:          if τ is triggered by a environmental sync event e then
20:             env_Events.add(e)
21:             env_Trans.put(e,τ)
22:          end if
23:          if τ is a non-interrupt,non-env-sync and non-rendezvous transition then
24:             enabled=true
25:             enabled_transitions.add(τ)
26:          end if
27:       end if
28:    else
29:       break {lower priority transition encountered, stop the checking loop}
30:    end if
31: end for
32: {Phase 2: check if we can react to rendezvous-sync events being generated in other HTSes}
33: for each transition τ triggered by a rendezvous event do
34:    if en_state(τ) ∧ en_cond(τ) ∧ pri(τ)>=max_pri_seen then
35:       rend_TEvents.add(e)
36:       rend_TTrans.put(e,τ)
37:    end if
38: end for
```

Figure 3.27: Full HTS Enabledness Check

```
    {Member Variables}
    VAR list enabled_transitions
    VAR map env_trans, rend_GTrans, rend_TTrans
 1: execute(event env_Event,event rend_GEvent,event rend_TEvent,transition interrupt)
 2: { Phase 1: Select a transition to execute based on scheduling information in the parameters}
 3: if interrupt is non null then
 4:    exe_trans = interrupt
 5: else if rend_GEvent is non null  then
 6:    exe_trans = pick a random transition from rend_GTrans.get(rend_GEvent)
 7: else if rend_TEvent is non null then
 8:    exe_trans = pick a random transition from rend_TTrans.get(rend_TEvent)
 9: else if env_Event is non null then
10:    exe_trans = pick a random transition from env_Trans.get(env_Event)
11: else
12:    exe_trans = pick a random transition from enabled_transitions
13: end if
14: { Phase 2: Execute the selected transition}
15: CS_NEXT(exe_trans)
16: CSa_NEXT(exe_trans)
17: AV_NEXT(exe_trans)
18: AVa_NEXT(exe_trans)
19: Q_NEXT(exe_trans)
20: Qa_NEXT(exe_trans)
21: IE_NEXT(exe_trans)
22: IEa_NEXT(exe_trans)
23: Ia_NEXT(exe_trans)
24: if  exe_trans is an interrupt transition then
25:    executeInterrupt(exe_trans)
26: end if
```

Figure 3.28: Full HTS Execution

The pseudocode shown in Figure 3.28 shows the HTS execution algorithm with support for all composition operators. In Phase 1, the HTS chooses one transition to execute from its member variables, based on the constraints imposed in the parameters. The order in which the constraints (parameters) are checked is significant. Only one of the first three parameters will ever be set, but the interrupt parameter can be set at any time. The interrupt parameter overrides any other parameter, as it imposes a more specific constraint (the actual transition to execute) than the first three parameters. For example, if the env_Event and interrupt parameters are both set, then this indicates that an interrupt transition triggered on env_Event has been set by an ancestor interrupt operator. Thus, the HTS should execute the interrupt transition and ignore the env_Event constraint (it is implicitly satisfied). Once a transition has been chosen from the proper member variable, the second phase of the algorithm executes the transition following the template-semantics apply (i.e., next_XX) parameter values.

3.7 The Root of the Composition Tree

The root of the composition tree has an extra task that other nodes in the tree do not have. The root has the job of resetting the snapshot elements at the start of a new macro-step, as described by the user-provided macro_semantics and ten reset_XX template-semantics parameters. The root component doesn't have an enabled_check() method; it instead contains a run() method. This method performs a single micro-step in a model's Java implementation. The method's form depends on whether the notation is using simple or stable macro-step semantics. An outline of the run method for simple semantics is shown in Figure 3.29. Since it follows simple semantics, the root resets the snapshot elements before every call to execute() (i.e., before every micro-step). The operator-specific enabledness check on line 12 is identical to the heterogeneous enabledness check outline presented in Figure 3.25. This enabledness check involves calling enabled_check() on one or both of the root's children to store their enabledness information in member variables. This stored information is used during execute() (line 13). The root calls

```
 1: run()
 2: RESET_IE
 3: RESET_IEa
 4: RESET_CS
 5: RESET_CSa
 6: RESET_AV
 7: RESET_AVa
 8: RESET_Q
 9: RESET_Qa
10: RESET_Ia
11: RESET_O
12: OPERATOR SPECIFIC ENABLED CHECK
13: execute(null,null,null,null)
14: if leftFinal ∧ rightFinal then
15:    exit the system
16: end if
```

Figure 3.29: Simple Run Method

```
 1: run()
 2: OPERATOR SPECIFIC ENABLED CHECK
 3: if ¬enabled then
 4:    RESET_IE
 5:    RESET_IEa
 6:    RESET_CS
 7:    RESET_CSa
 8:    RESET_AV
 9:    RESET_AVa
10:    RESET_Q
11:    RESET_Qa
12:    RESET_Ia
13:    RESET_O
14: else
15:    execute(null,null,null,null)
16: end if
17: if leftFinal ∧ rightFinal then
18:    exit the system
19: end if
```

Figure 3.30: Stable Run Method

its own execute() method with all null parameters, so that execute() can make an unconstrained execution decision. The RESET_XX() procedures (lines 2-11) in the pseudocode represent inline code substitutions of the Java implementations of the reset_XX template parameters (i.e., the generated code does not contain these procedures, instead reset_XX's Java implementation is substituted wherever RESET_XX is used in the pseudocode). The version of run() for stable semantics (Figure 3.30) is modified slightly to only reset the snapshot elements when the model has no transition to execute (the model is stable). The model is stable if the root operator's enabledness check (line 2) failed to find anything to execute (i.e., enabled was set to false by the root's enabledness check).

3.8 Optimizations

We have implemented several optimizations to the scheduling methodology described above. The three biggest optimizations are flattening the composition tree, carrying out static computations, and removing nondeterminism. We describe the first two optimizations in the following sections. Chapter 4 investigates removing nondeterminism.

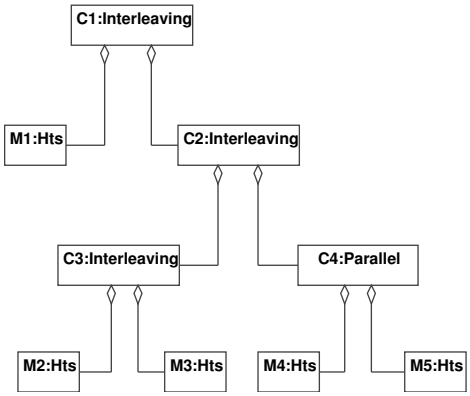


Figure 3.31: Binary Operators

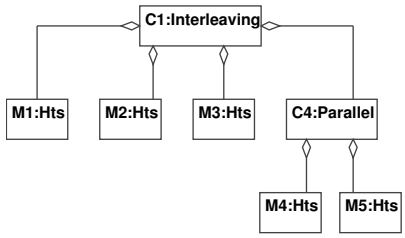


Figure 3.32: General Operators

3.8.1 Flattening the Composition Tree

Up until this point, a model's composition hierarchy has been a binary tree of composition operators. Some composition operators are associative, which means that they don't need a binary hierarchy to identify which operations should be performed first. In such cases, we can flatten the hierarchy by allowing associative operators to have an arbitrary number of children, thereby forming a general tree. The operators we flatten are interleaving, parallel, choice, and sequence. Environmental-sync operators could also be flattened if the operators all have the same set of sync events, although this is currently not implemented.

An example hierarchy is shown in Figure 3.31. To flatten this hierarchy, we remove all operators of an associative type, whose parent is an operator of the same type. Removing an operator involves promoting the operator's children by making them children of the operator's parent and then removing the operator itself from the hierarchy. For example, in Figure 3.31, operator C3 and its parent C2 are both interleaving operators. Thus, C3 is removed from the hierarchy and its children (M2 and M3) are promoted making them children of C2. Operator C2 and its parent C1 are also both interleaving operators, so C2 is removed from the hierarchy and its children are promoted, making them children of C1. This brings us to the final flattened hierarchy shown in Figure 3.32. To accommodate

general composition trees, the enabled_check() and execute() methods of associative operators are modified to work on a set of child components instead of a left and right child. Each operator class contains a set of children and a set of enabled flags, one for each child. The enabled_check() method asks all children for their enabledness information and uses this information to compute the operator's own enabledness, which it passes to its parent. An associative operator's execute() method then chooses which children amongst the set of enabled children to execute. For example, with parallel composition, all enabled children are allowed to execute in the same micro-step, where as interleaving composition chooses one enabled child to execute. The shallower a model's composition hierarchy is, the faster the generated code for that model will execute. The savings comes from fewer recursive calls and less caching of enabledness information, in that the generated code for the flattened model caches no information for the removed operators. If the root-composition operator is an associative operator, then we can also make some optimizations to the run() method. These optimizations are implementation details; the actual algorithms remain unchanged.

3.8.2 Static Computations

In the Java implementations of the execute() and enabled_check() methods, as much information as possible is computed statically by the code generator. For example, in the presented pseudocode, the synchronization operators calculate set intersections and unions with the set syncEvents and check if the resulting set is empty (see Figures 3.18 and 3.22). The code generator knows statically the sync events in syncEvents; instead of calculating the intersections and unions and checking for set emptiness, it generates if-then-else statements that look for a single sync event in the target set. For example, to check

$$\text{EventsSet1} \cap \text{EventSet2} \cap \text{syncEvents} \neq \emptyset$$

the generator outputs a long listing of if-then-else statements containing one if statement for each event in syncEvents, checking if that event is in both EventSet1 and EventSet2. As soon as one sync event is found to be contained in both sets, the

if-then-else statement completes (and returns success). The generated code ends up being longer, but it executes much faster.

As a second example, template semantics has the notion of a state hierarchy. This state hierarchy is static in the model and is completely flattened for enabledness checking and execution. The current states (CS) snapshot element is always just a set of states; we don't care if those states are related via the state hierarchy. The state hierarchy is encoded as part of the transition-execution code for the CS snapshot element (representing the next_CS template parameter). The code generator statically calculates which states in the state hierarchy are entered and exited when a transition executes, as oppose to generating code that does this calculation at runtime. The result is more efficient generated code for transition execution.

As a third example, finding the set of highest-priority enabled transitions in the HTS enabled_check() method (Figure 3.27) also takes advantage of statically available information about transition priorities. Instead of finding the set of enabled transitions and then determining which transitions in this set have highest priority, the code generator orders the transitions so that their enabledness is checked by the generated code in descending order of priority; once a transition is found to be enabled, all transitions of lower priority are not checked. Thus, the code generator does all the work of dealing with the priority scheme by sorting the transitions based on their priorities at code-generation time.

Chapter 4

Elimination of Nondeterminism

Template semantics can describe notations that can be used to build nondeterministic models. This presents a problem when generating source code. Outside of multi-threading, Java semantics is inherently deterministic. In the previous chapter, we described the sense in which our code generators generate a Java representation of a model that preserves the model's nondeterminism. This approach is useful for directly simulating the model, but not for producing a deployable implementation. This chapter identifies all possible sources of nondeterminism in a model, and establishes mechanisms for eliminating them, so that our code generators can produce more efficient deterministic Java implementations. Moreover, we want to give the modeller as much control as possible over how the nondeterminism is resolved. In cases where resolving the nondeterminism would involve asking the modeller for too much information, we propose a global rule for resolving the nondeterminism. The CGG allows the modeller to choose between two different types of code generator. The first type will generate code that preserves the nondeterminism in the model. The second type uses the techniques described in this chapter to generate code that eliminates all nondeterminism in the model.

We are concerned with two categories of nondeterminism. The first category is HTS-level nondeterminism. This is nondeterminism in the next-state relation for an HTS (i.e., HTSs having more then one transition enabled in the same micro-

step). In the pseudocode for translating HTSs to Java, presented in section 3.6, a lot of extra complexity was introduced to simulate an HTS's nondeterminism in the generated code. In the deterministic version of the model, we want each HTS to have at most one enabled transition per micro-step. The second category of nondeterminism is at the composition level. We look at the nondeterminism that a composition operator introduces even if both of its children are themselves deterministic. These are points in the operator's execute() method where the operator has a choice regarding which execution decision of multiple possible execution decisions to make.

4.1 Nondeterminism Elimination at the HTS Level

The only source of nondeterminism at the HTS level comes from simultaneously enabled transitions. In template semantics, two transitions can be simultaneously enabled if their source states, triggering events, and guard conditions can be satisfied in the same snapshot, where "satisfied" is specified by template parameters en_cond, en_events, and en_states.

We have developed a static check that will warn the modeller when two of an HTS's transitions can be simultaneously enabled, so that the modeller can decide how to resolve that nondeterminism in the model. If two equal-priority transitions exit the same source state and have jointly satisfiable guard conditions, then the two transitions could both be simultaneously enabled. Thus, the checking algorithm performs a pairwise satisfiability check on the guard conditions of all equal-priority transitions that originate from the same source state. The checking algorithm that we use is shown in Figure 4.1. Because template semantics has the notion of a state hierarchy, Phase 1 of the algorithm finds the set of transitions exiting a given basic state (a state having no sub-states) or exiting any of the basic state's ancestor super states (lines 4-7). Phase 2 finds all pairs of equal-priority transitions, from the set found in Phase 1, whose guards can be simultaneously satisfied (line 9). The implementation of the algorithm makes use of the Stanford Validity Checker[5] to solve the satisfiability sub-problem (line 9). This check does not involve the transition's

```
1: for each basic state s do
2:    add all transitions leaving s to transList
3:    s = s.outer {s.outer is the super state which contains s}
      {Phase 1: find all transitions which exit s or any of its ancestor states}
4:    while s is not the topmost state in the state hierarchy do
5:       add all transitions exiting s to transList
6:       s = s.outer
7:    end while
      {Phase 2: pairwise mutually enabled check}
8:    for each t1 and t2 in transList with pri(t1)=pri(t2) do
9:       if en_cond(t1) ∧ en_cond(t2) is satisfiable then
10:         t1 and t2 can be mutually enabled
11:      end if
12:   end for
13: end for
```

Figure 4.1: HTS Level Nondeterminism Detection

triggering events, because if multiple enabling events can be active in the model at the same time, then transitions triggered by different events can be enabled simultaneously. The algorithm is complete but not sound. A full reachability analysis would be required to check for sure if two transitions can be simultaneously enabled. This type of reachability analysis is far too expensive to be feasible. Thus, false-positives are possible when using this algorithm. The idea is to point out to the modeller pairs of transitions that are potential sources of nondeterminism. The modeller can choose to ignore the warnings if the reported pair of transitions will never be simultaneously enabled (i.e., the pair represents a false-positive), or if the modeller does not care how the nondeterminism resulting from the pair of transitions is resolved.

Once a pair of transitions has been detected as a source of nondeterminism, the modeller can use explicit transition-priority to identify which transition should execute if both are enabled. If the modeller doesn't care how HTS-level nondeterminism is resolved in a model's implementation, then the detection algorithm can be disabled. If the modeller doesn't prioritize transitions that can be simultaneously enabled, or if the model's priority scheme doesn't specify a total ordering

amongst transitions, then the code generator uses the order in which the transitions are listed in the model's XML representation to prioritize transitions.

By forcing a total ordering amongst transitions that can be simultaneously enabled, we ensure that only one transition per HTS will be enabled in a micro-step. This knowledge allows us to redesign the algorithms for HTS-level enabledness checking and execution. The goal of this redesign is to make the algorithms as efficient as possible. These modified algorithms are shown in the following sections.

4.1.1 Deterministic HTS Enabledness Checking

In this section, we assume that there exists a total ordering amongst transitions that can be enabled in the same micro-step, such that at most one transition per HTS will be enabled in a micro-step. The redesigned enabledness checking pseudocode is shown in Figure 4.2. The redesigned HTS class has only two member variables (enabled_transitions and rend_TTrans) compared to the four member variables used in the pseudo-nondeterministic HTS enabled_check() (Figure 3.27). The enabled_transition variable stores the one enabled transition (line 9). Since at most one transition is ever enabled in a micro-step, the HTS doesn't need different member variables to separate synchronization related transitions from nonsynchronization transition, as was done in the pseudo-nondeterministic HTS enabled_check(). As for the other member variable (rend_TTrans), recall that if an HTS has an ancestor rendezvous-composition operator, then it stores the transitions that would be enabled if the proper sync event were to be generated. These transitions are not actually enabled unless a rendezvous synchronization on the proper sync event occurs. Thus, an HTS stores these transitions separately, just as in the nondeterministic case. This is the purpose of the map rend_TTrans at the top of Figure 4.2.

The simplified enabled_check() method is shown in Figure 4.2. The first phase of the algorithm finds the one highest-priority enabled transition and communicates, via enabled_check()'s parameters, the same four pieces of enabledness information about this transition as the HTS did in the nondeterministic pseudocode. The sec-

```
{Member Variables}
VAR transition enabled_transition
VAR map rend_TTrans
 1: enabled_check(set rend_TEvents,set rend_GEvents,set env_Events,set interr_trans,bool en-
    abled,bool final_state)
 2: if  CS contains a final state then
 3:    final_state=true
 4:    return
 5: end if
 6: {Phase 1: Search for the one enabled transition}
 7: for each transition τ in descending order of priority do
 8:    if en_state(τ) ∧ en_cond(τ) ∧ en_events(τ) then
 9:       enabled_transition=τ
10:       if τ generates a rendezvous event e then
11:          rend_GEvents.add(e)
12:       end if
13:       if τ is an interrupt transition then
14:          interr_trans.add(τ)
15:       end if
16:       if τ is triggered by a environmental sync event e then
17:          env_Events.add(e)
18:       end if
19:       if τ is a non-interrupt,non-env-sync and non-rendezvous transition then
20:          enabled=true
21:       end if
22:       break {Stop the checking loop, we found the one enabled transition}
23:    end if
24: end for
25: {Phase 2: check if we can react to rendezvous-sync events being generated in other HTSes}
26: for each transition τ triggered by a rendezvous event do
27:    if en_state(τ) ∧ en_cond(τ) ∧ pri(τ)>=max_pri_seen then
28:       rend_TEvents.add(e)
29:       rend_TTrans.put(e,τ)
30:    end if
31: end for
```

Figure 4.2: Deterministic HTS Enabledness Check

ond phase is identical to the second phase of the pseudo-nondeterministic algorithm: the HTS checks if any rendezvous-event-triggered transitions are prepared to take part in a rendezvous. These transitions are not actually enabled unless a rendezvous occurs and wouldn't be found by the enabledness check in Phase 1. This deterministic enabled_check() method is more efficient than the pseudo-nondeterministic

```
   {Member Variables}
   VAR transition enabled_transition
   VAR map rend_TTrans
 1: execute(event rend_TEvent)
 2: { Phase 1: Select a transition to execute based on scheduling information in the parameters}
 3: if  rend_TEvent is non null  then
 4:    exe_trans = rend_TTrans.get(rend_TEvent)
 5: else
 6:    exe_trans = enabled_transition
 7: end if
    { Phase 2: Execute the selected transition }
 8: CS_NEXT(exe_trans)
 9: CSa_NEXT(exe_trans)
10: AV_NEXT(exe_trans)
11: AVa_NEXT(exe_trans)
12: Q_NEXT(exe_trans)
13: Qa_NEXT(exe_trans)
14: IE_NEXT(exe_trans)
15: IEa_NEXT(exe_trans)
16: Ia_NEXT(exe_trans)
```

Figure 4.3: Deterministic HTS Execution

enabled_check() method. The biggest savings come in Phase 1; the HTS only identifies the one highest-priority enabled transition, compared to the nondeterministic case, where there may exist a set of high-priority enabled transitions.

4.1.2 Deterministic HTS Execution

If at most one transition per HTS is enabled in a micro-step, then the HTS execute() method can be redesigned to use fewer parameters. The pseudo-nondeterministic HTS execute() method (Figure 3.28) had to process four possible ancestral constraints. It also randomly selected an enabled transition to execute if more than one enabled transition satisfied the constraints. In the deterministic version of the algorithm (Figure 4.3), the only ancestral constraint that concerns an HTS is a directive to execute a transition that is triggered by the rendezvous event in the rend_TEvent parameter. In all other cases, the enabled transition that the HTS should execute is stored in the enabled_transition member variable; it doesn't mat-

ter if this transition is an interrupt transition or is triggered by a environmental synchronization event. Thus, the other ancestral constraints used in the pseudo-nondeterministic HTS execute() method play no role in the deterministic algorithm, and hence are removed. The HTS either executes a rendezvous or executes its single enabled transition. This is a much more elegant and efficient execution mechanism.

4.2 Nondeterminism Elimination at the Composition Level

Each composition operator, other than parallel and sequence, introduces some new source(s) of nondeterminism. We will show briefly how each of these sources of nondeterminism is resolved in the generated Java. Parallel composition is not mentioned in the following discussion because it forces its left and right child components to execute in parallel if they are both enabled. Thus, parallel composition is deterministic as long as each of its left and right children are deterministic. Sequence composition simply executes each child component in sequence, and the order in which the children are executed is specified in the model, making sequence composition deterministic as well.

4.2.1 Interleaving

Interleaving composition introduces a single source of nondeterminism. When both left and right child components are enabled, an interleaving operator nondeterministically picks one of the two components to execute. To resolve this nondeterminism, the generated code for interleaving composition uses round-robin scheduling to decide which child component executes if both are enabled in a particular micro-step. This decision results in both children of an interleaving operator taking turns executing whenever both are enabled (i.e., if the left child executed last time both children were enabled, then the right child will execute the next time both children are enabled). Round-round scheduling gives us a notion of fairness without having

any measurable effect on the efficiency of execution.

4.2.2 Synchronization

Environmental and rendezvous synchronization introduce a number of sources of nondeterminism. In the unsynchronized case in both operators, if both the left and right children are enabled on transitions not related to the operator's sync events, then the operator behaves as an interleaving operator, nondeterministically selecting a child to execute. This nondeterminism is resolved using round-robin scheduling, in the same fashion as was done for interleaving composition.

A second source of nondeterminism occurs in the synchronized case, if the operator can synchronize on multiple sync events. In the nondeterministic psuedocode (Figures 3.18 and 3.23), the operator randomly chooses which of these events to sync on, and then directs its child components to execute transitions triggered on the chosen event. We use an event-priority scheme to resolve this nondeterminism when generating a deterministic Java implementation. This priority scheme must provide a total ordering on the model's sync events. This single priority scheme is used for all synchronization operators in the model. A synchronization operator will choose to synchronize on a higher-priority sync event over synchronizing on a lower-priority sync event. Ideally, the modeller specifies a priority scheme among synchronization events. If no such ordering on events is specified by the modeller, then the order in which the events are listed in the XML representation of the model will be used to prioritize events when generating deterministic code.

The last source of nondeterminism in synchronization operators occurs when the operator chooses between synchronizing on some event or allowing a child component to execute some transitions unrelated to the operator's sync events. In the pseudo-nondeterministic pseudocode, the operator randomly chooses whether to execute the synchronization or to allow one of its enabled child components to execute unsync transition(s) (i.e., transitions that are not triggered on a sync event, and that do not generate a sync event). When generating deterministic code, we resolve this nondeterminism by changing the operator's execute() method to give the

sync case higher priority than the unsync case. Thus, whenever a synchronization is possible, a sync operator's execute() method will execute that synchronization over any other transitions enabled in its children. This simplifies environmental sync's and rendezvous' execute() method. A side effect of this decision is that synchronization operators that appear higher in the composition tree have priority over synchronization operators that appear lower in the tree: because the flow of execution decisions starts at the root operator of the tree, higher-level synchronization operators will make execution decisions before lower-level synchronization operators. In the pseudo-nondeterministic code, the composition hierarchy does not impose a priority ordering on composition operators because the higher-level operator randomly chooses whether or not to synchronize its child components or to allow one of its children to execute unconstrained (allowing it to make its own execution decisions).

Rendezvous composition introduces one additional source of nondeterminism: if both children could be both a sender and a receiver of the same sync event in the same micro-step, then the operator nondeterministically chooses which child will be the sender and which child will be the receiver. We resolve this nondeterminism in the same manner as interleaving composition, by using round-robin scheduling to alternate, in different micro-steps, which child will send/receive the rendezvous-synchronization event, when both children could do either.

4.2.3 Interrupt

Interrupt composition introduces two sources of nondeterminism. The first source of nondeterminism is when the operator has more than one highest-priority enabled interrupt transition. To eliminate this nondeterminism, the modeller needs to define total orderings over each interrupt operator's interrupt transitions (i.e., a total ordering amongst the transitions in each operator's ourInterrupts set). This ordering ensures that there will be only one highest-priority interrupt transition for any interrupt operator in any micro-step. If the modeller doesn't provide such a priority scheme, the order in which the interrupt transitions are declared in the

XML representation of the model will be used to prioritize the transitions when generating code.

The second source of nondeterminism occurs when one of the operator's interrupt transitions is enabled at the same time as the active child component is enabled. This case occurs only if the highest-priority enabled interrupt transition has the same priority as the highest-priority enabled transition in the active child. We resolve this conflict by giving the operator's enabled interrupt transition precedence over equal-priority enabled transitions in the active child. This decision ensures that interrupt operators that appear higher in the composition tree have priority over operators that appear lower in the tree, if the operator's respective highest-priority enabled transitions have equal priority.

4.2.4 Choice

The choice operator nondeterministically chooses one child component and thereafter executes only that child component. There are two possibilities for how to resolve this nondeterminism. The code generator itself, at code-generation time, could ask the modeller which child component the operator should execute. In this case, the modeller's chosen component executes in every run of the system. Alternatively, the code generator could generate code that asks the user at execution time which component should execute in this run of the system.

4.2.5 Summary

The above measures give the modeller some control over how different sources of nondeterminism are resolved. Priority schemes give the modeller a lot of power in this regard. For some sources of nondeterminism, it would be too much work for the modeller to specify, in all cases, how the nondeterminism should be removed (i.e., asking the modeller to choose which child component of an interleaving operator should execute whenever both children are enabled in the same micro-step).

In these cases, we have prescribed a default mechanism for resolving the nondeterminism, which the modeller can override using explicit priorities. The result is composition operators with simpler and more natural source-code representations, which improves the efficiency of the generated code.

Chapter 5

Validation

We have described a tool for supporting configurable model-driven development, in which we automatically generate software implementations from models of software systems written in model-based notations. In this chapter, we investigate the correctness of our code generation (i.e., does the implementation reflect the same behaviours as the user's model) and the efficiency of our generated code.

5.1 Correctness

To demonstrate that our code generators generate code that matches the behaviour of the model from which it was created, we designed a test suite of models to white-box test the code-generator generator. Different models test different template-semantics parameters and different composition operators. We used the micro-step-level correctness criteria described in Section 1.3 to test the code generated from each test model: if the model and source-code are in equivalent states at the start of a micro-step, then they must be in equivalent states at the end of a micro-step. How a micro-step is actually carried out will never be equivalent in the model and the source-code because there is to no support for parallel execution in Java.

A test in the test suite consists of

- A model

- A template-semantics parameter definition file that defines the model's notation
- A sequence of environmental inputs to be used when executing the model's generated code
- The generated code's expected output

The expected output for a test is given as a sequence of snapshots, one per micro-step, that reflect the execution steps a correct Java implementation should take as it executes given the sequence of input events. As such, the model's execution serves as a test oracle for evaluating the execution steps of the generated code, when fed the same input events. The CGG has an optional flag that, when set, builds code generators that generate Java that outputs this snapshot information after every micro-step's execution. To run a test, the CGG and the test's template-parameter definition file are used to compile a code generator for the test's notation, the test's model is given as input to this generator to produce a matching Java implementation of the model, and then the generated Java is executed using the inputs provided in the test. If the output of this execution matches the test's expected output, then the test is deemed successful.

The test suite contains tests for both deterministic and pseudo-nondeterministic code generation. A test's template-semantics parameter definition file specifies the type of code generation to use in that test. To test pseudo-nondeterministic code generation, the test suite uses deterministic models (i.e., the model has no nondeterminism, so the code never executes a random choice). Using deterministic models to test pseudo-nondeterministic code generation ensures that a test has a single expected output. We also inspected manually the pseudo-nondeterministic generated code for nondeterministic models, over many executions of the generated code, to ensure the output always matched one of the set of expected outputs of the model.

The test suite contains two categories of models. The first category is designed to test our scheduling and execution methodology via models whose composition hierarchies have different combinations of composition operators. For example,

- Models that consist of one HTS (to test HTS-level execute() and enabled_check())

- Models that consist of one composition operator (to test every case of each operator's execute() method)
- Models that contain all interesting hierarchies using two composition operators (to show that each operator can respond to any constraint that another operator may impose on it)
- Several models with deeper hierarchies and interesting combinations of composition operators (i.e., a hierarchy containing multiple interrupt, environment sync, and rendezvous sync operators)

The second category of models in the test-suite are those designed to test the template-parameter values. These models exercise each implemented parameter value by using it to define at least one test's notation. This category of models also ensures that all snapshot element implementations (i.e., the IE snapshot element can be implemented as a set or as a queue) are tested.

5.2 Ground Traffic-Control Case Study

To demonstrate the functionality of the CGG, we define a notation, specify a model in this notation, and generate Java code for this model. The model is based on a ground traffic-control system case study that was developed by Bultan and Yavuc-Kahveci to demonstrate their code-generation technique[33]. The reasons for choosing this case study are two fold. First, it represents a safety-critical system. 20% of commercial airline accidents involving hull loss occur during take-off, taxing, or parking, and 46% of accidents occur during landing[2]. These accidents are associated with airport ground-traffic control. Second, the model of the ground traffic-control system is simple, yet makes use of several of the more complex synchronization operators.

This study models the airport shown in Figure 5.1. It contains two runways, R1 and R2, as well as three taxiways C1, C2, and C3. Airplanes can arrive or depart on either runway. Arriving airplanes landing on runway R1 must taxi using one of taxiways C1-C3 to reach a hanger. The system must satisfy the following properties.

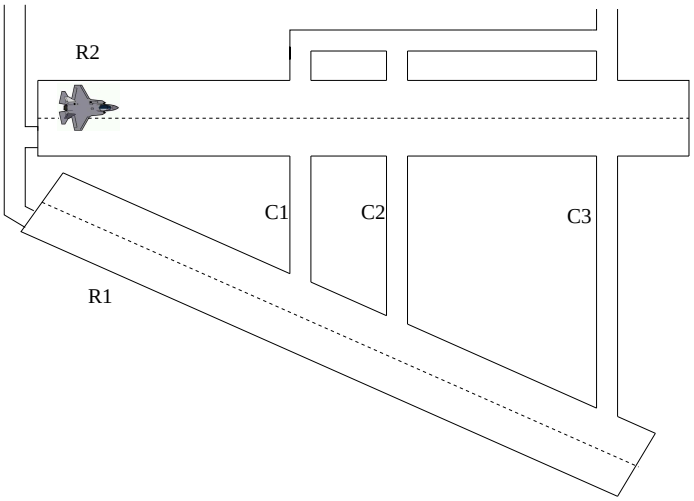


Figure 5.1: Airport Case Study

- Only one airplane can use a runway at a time.
- Only one airplane can use a taxiway at a time.
- An airplane can use runway R1 (R2) only if no airplane is using R2 (R1)
- Planes wanting to land have priority over planes wanting to take off.
- An airplane taxiing on one of C1-C3 can only cross runway R2 if no plane is currently using it.
- An airplane can start using R2 for take-off or landing only if none of taxiways C1-C3 is currently in use.

5.2.1 The Notation

To build our model, we used a notation similar to Statemate's statecharts[13]. The semantics for states and variables are the same as Statemate's semantics. The current variable values (AV) are used to evaluate guard conditions, and only transitions whose source states are elements in the set of current states (CS) are enabled in a micro-step. The main difference between our notation and Statemate's notation is how each deals with events. The template-semantics description of our notation is shown in Table 5.1. In our modelling notation, a generated event, whether it is an

Parameter	Value
next_CS(ss,τ,CS')	CS'=entered(dest(τ))
reset_CS(ss,I)	ss.CS
en_states(ss,τ)	src(τ) $\subseteq$ ss.CS
next_AV(ss,τ,AV')	AV'=assign(ss.AV, eval(ss.AV, incr(asn(τ)))) (τ's actions are evaluated incrementally)
reset_AV(ss,I)	ss.AV
en_cond(ss,τ)	ss.AV $\models$ cond(τ) (variables values in AV must satisfy τ's guard)
next_IE(ss,τ,IE')	IE'=ss.IE\trig(τ) $\cup$ gen(τ) (remove τ's trigger, add τ's generated events)
reset_IE(ss,I)	ss.IE
next_Ia(ss,τ,Ia')	Ia'=ss.Ia\trig(τ) (remove τ's trigger)
reset_Ia(ss,I)	ss.Ia $\cup$ I.gen (add events sensed from the enviornment)
en_events(ss,τ)	trig(τ) $\subseteq$ ss.Ia $\cup$ ss.IE
macro_semantics	Stable
pri	Explicit priority

Table 5.1: Airport Case Study Notation Semantics

internal or external event, remains active in the model, even in a subsequent macro-step, until a transition is triggered by that event. The IE parameters describe this semantics for internal events, and the Ia parameters describe this semantics for external events. Intuitively, this means that an event is considered handled when a transition reacts to it. The notation uses stable semantics; a model senses the environment only when it has no enabled transitions.

5.2.2 The Model

Our model of the ground traffic-control system describes only the logic for controlling the aircraft; it doesn't model the behaviours of the aircrafts themselves. The aircraft are considered part of the model's environment. An aircraft can make a request to takeoff, land, or taxi. These requests are represented by events reqtakeoff, reqland, and reqtaxi. The model responds to a request to land or takeoff by specifying which runway the plane is to use, one of R1 or R2. The model responds

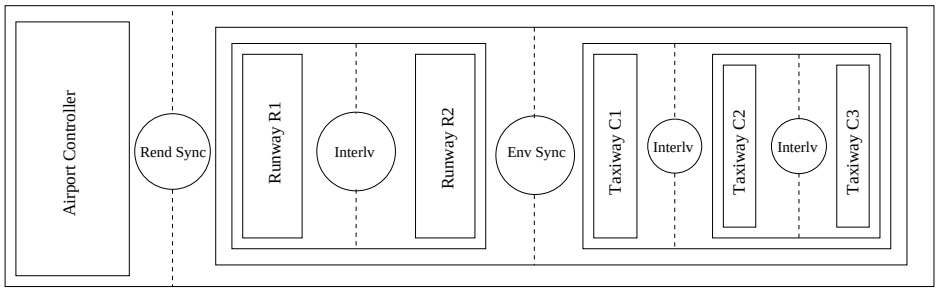


Figure 5.2: Composition Hierarchy

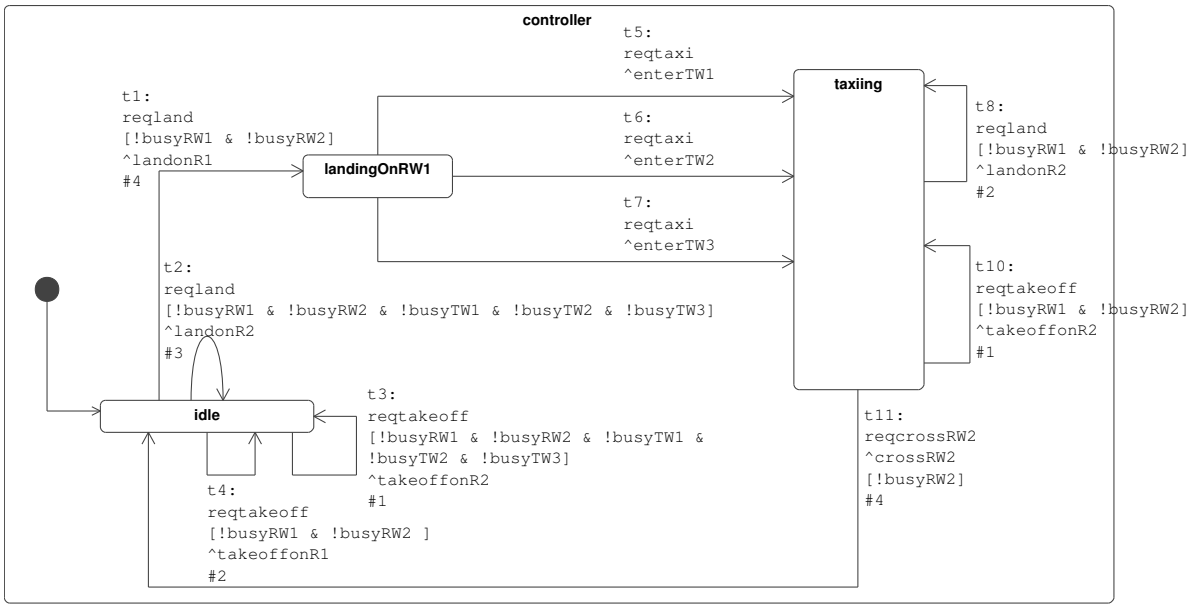


Figure 5.3: Airport Controller

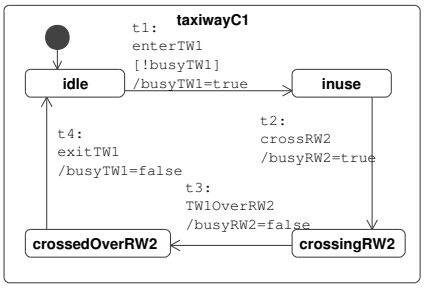


Figure 5.4: Taxiway

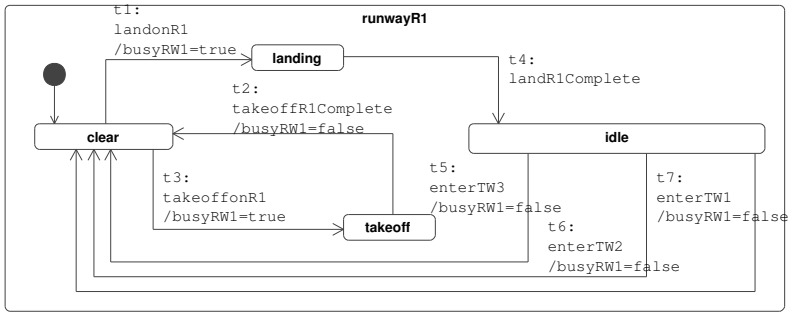


Figure 5.5: Runway

to a request to taxi by specifying which taxiway the plane has permission to use. These responses are represented by events landonRX, takeoffonRX, and enterTWX for runway or taxiway X. There is only one other request that a plane can possibly make: when taxiing towards the hanger, a plane that needs to cross runway R2 must ask, and be granted, permission before crossing. To request a crossing of R2, an aircraft uses event reqcrossR2. The controller replies with event crossR2 when runway R2 is clear for crossing. The boolean flags busyRX and busyTWX indicate if runway X or taxiway X is currently in use. Using these input and output events, we can encode the reactive behaviour of the airport ground-traffic controller.

The controller machine is shown in Figure 5.3. This machine responds to request events from the environment and gives orders to the runway and taxiway machines. The runwayR1 and taxiwayC1 machines are shown in Figures 5.4 and 5.5 (the other runway and taxiway machines are similar). Each of these machines keeps track of the current state of its respective runway or taxiway. The composition hierarchy of the model is shown in Figure 5.2. The rendezvous synchronization between the aircraft controller and the rest of the model ensures that the other components react to the controller's commands in the same micro-step that the commands are issued. For example, when the controller machine generates the event landonR1, the runway machine for R1, in the same micro-step, responds by setting local variables that indicate that the runway is now busy. This synchronization keeps the runway state machines in tight synchronization with the controller as it directs the aircraft. The model uses environmental synchronization to keep the runways and taxiways in sync with each other, especially when an airplane is in the intersection of a taxiway and a runway. This occurs when a plane taxis across runway R2.

We generated the code for this model and tested and inspected it to ensure the code adhered to the specification. Some of the generated code for this model is shown in Appendix B. To test the generated code, we used a set of environmental inputs designed to show that the properties of the ground-traffic control system listed in Section 5.2 hold in the generated code. Ideally, in the future, we would like to explore how source-code model-checking techniques, such as those supported by Java PathFinder[14], could be used to more rigorously prove that the generated

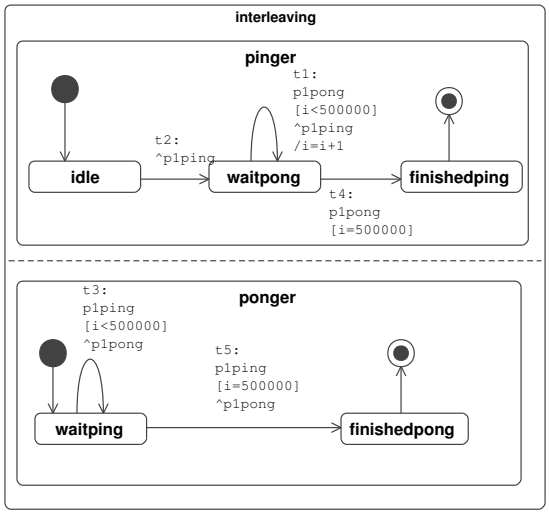


Figure 5.6: PingPongx1 System

code matches the model.

5.3 Efficiency

In this section, we compare the code-generation capabilities of our code generators, created by the CGG, with four commercial code generators that have been developed for specific notations. This comparison is based on a set of four benchmark systems, which can be modelled in each of the commercial tools' modelling notations. The ground traffic-control system case study, as modelled in Section 5.2, is not one of the benchmark models because it uses synchronization operators not supported by any of the commercial tools.

The first of the benchmark models is a simple model, called PingPongx1, with a shallow composition hierarchy. It is shown in Figure 5.6. This model is provided as an example model with the Rational Rose Realtime[18] tool suite. The model has two concurrent machines that carry out a simple request-reply protocol. The pinging machine sends a ping to the ponging machine. The ponging machine sends a pong to the pinger upon receiving the ping. This request-reply protocol is repeated a

set number of times before the model terminates. The type of composition operator used in the model is either interleaving or parallel, whichever is supported by the commercial tool we are building the model in. The second and third benchmark models, PingPongx2 and PingPongx8, were created by composing together two and eight copies of PingPongx1, to form larger models. The type of composition used to form these larger models depends on the type of composition supported by the commercial tool we are building the model in (either parallel or interleaving composition). These two models test the efficiency of the generated code when the input model has a larger number of HTSs. As a fourth comparison, we constructed a model that contains more complicated state machines. The fourth benchmark model is based on a simple distributed mutual-exclusion algorithm: two requestor machines ask a controller for permission to access a critical section. A requestor waits until the controller has granted it permission before accessing the critical section. The requestor notifies the controller when it is finished using the critical section. The controller ensures that only one requestor ever has access to the critical section at any point in time. The critical section we use in this model is simple: a requestor simply increments an integer value before leaving.

We used these four models to compare our CGG-generated code generators against the code generators of four commercial tools. We specified each of the four benchmark models in each of the tools' notations, giving us four versions of each of the four models described above. We also built template-semantics versions of each of the 16 models described above (4 systems modelled in 4 commercial tools), to be used as input to our code generators. Descriptions of the four commercial tools are given in the following sections.

5.3.1 Rational Rose Realtime

Rational Rose Realtime (RoseRT)[18], developed by IBM, follows the semantics of OMG-UML[28]. RoseRT allows the modeller to specify how many threads will be present in the generated code and to assign a model's components to specific threads. Each thread has a global input queue on which all messages to any com-

ponent that executes within the thread are enqueued. For the sake of comparing RoseRT generated code with our generated code, we set RoseRT to generate single-threaded code (since our code generators generate single-threaded code). Thus, our RoseRT models have a single queue for the model via which the model receives sensed input from the environment and through which all inter-component communication (via internal events) occurs. RoseRT also introduces the notion of directed communication between components on ports (i.e., a transition can output an event e on port p, or can be triggered by an input event on port p). In OMG-UML, ports between components are connected by communication channels. Thus, on the surface, it appears as if each of these ports would map to its own event queue in the model's Java implementation. It turns out that ports are merely part of the syntax, and that any event put on any port ends up on the global queue. All transitions in RoseRT, other than those leaving special choice states, must have a triggering event. We defer the discussion of choice states until after the semantics description. The only composition operator supported by RoseRT is interleaving.

Our template-semantics versions of the RoseRT models simulate ports using event names. We create a new event name for each object/port/event combination. Then, if a transition in object A generates an event e and puts it on port p, a preprocessor determines p's destination object and port (e.g., port q of object B) and puts event B.q.e on the global queue. A transition in object B that is triggered by event e on port q is modified to be triggered by event B.q.e on the global queue.

The key template parameters for describing RoseRT's semantics, including the event-related parameters, are shown below. In this description, the global queue for the system is located in the Q snapshot element (Q.q).

- *macro semantics : simple*
  The environment is sensed after every micro-step in the model.

- *en_events(ss,τ) : trig(τ)==head(ss.Q.q)*
  For a transition to be enabled, its triggering event must be at the head of the global queue.

- *next_Q(ss,τ,Q') : Q'.q=ss.Q.q$\frown$gen(τ)*

Any events that an executing transition generates are appended to the back of the global queue.

- *reset_Q(ss,I) :tail(ss.Q.q)⌢I.gen*
  At the start of a macro-step, the head of the global queue is removed and any newly sensed events from the environment are added to back of the queue. Recall that all transitions in RoseRT, other than those leaving choice states, must have a triggering event. At the end of a macro-step, one of two things will be true about the event at the front of the queue. The event either triggered a transition at some point in the macro-step or the event didn't trigger any transitions. If the event triggered a transition, then the front of the queue should be popped as this event has been reacted to. If the event didn't trigger any transitions, the front of the queue should be popped to allow the model to make forward progress (otherwise, the model would be deadlocked). Thus, during reset, the front of the global queue is always popped.

RoseRT supports choice states that are not directly captured by our template-semantics description of RoseRT. A choice state is an intermediate state in a compound transition (see Figure 5.7). A transition entering a choice state is a regular transition triggered by an event and possibly having a guard condition and actions. A transition exiting a choice state cannot have a triggering event; it has only a guard condition and possibly some actions. The guard conditions of all transitions exiting a given choice state must be mutually exclusive and complete, so that exactly one of these exiting transitions will always be enabled (i.e., there is no nondeterminism and it is impossible for the execution to be deadlocked in a choice state). The model doesn't sense new events from the environment while in a choice state.

We could simulate choice states by setting the macro_semantics parameter to stable semantics, to represent the compound transition as a sequence of micro-steps and to keep from starting a new macro-step (i.e., sensing the environment) in the middle of the compound transition. This decision, however, would result in highly inefficient generated code (see Table 5.4).

Instead, we use simple macro-step semantics in our template-semantics description of RoseRT and handle choice states as part of a syntactic translation that

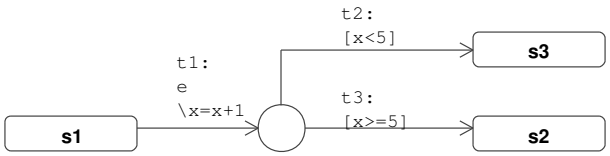


Figure 5.7: A Choice State

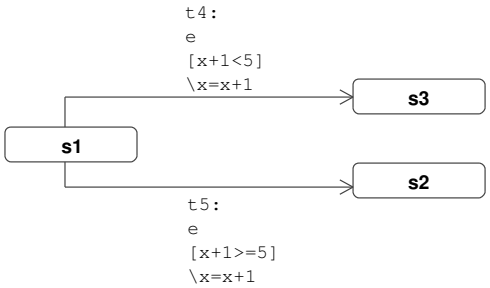


Figure 5.8: Choice State Removed

removes choice states from the model. To remove a choice state, we replace each compound transition (i.e., each pair of transitions in which one transition enters the choice state and the second transition exits the choice state) with a single transition whose

- source state is the source state of the transition entering the choice state
- destination state is the destination state of the transition exiting the choice state
- triggering events are the same as the triggering events of the transition entering the choice state
- guard condition is the conjunction of the guard condition from the transition entering the choice state and an updated version of the guard condition from the transition exiting the choice state, (how to update the guard is described in the following paragraph)
- generated events are the union of the events generated by both transitions

- variable assignments are the sequential composition of the variable assignments from the transition entering the choice state appended with the updated variable assignments from the transition exiting the choice state, (how to update the variable assignments is described in the following paragraph)

Because the expressions (in guard conditions and variable assignments) in transitions that exit a choice state are evaluated with respect to current variable values, which might have been modified by the transition that entered the choice state, we use updated versions of these expressions when creating new transitions. The updated versions substitute any reference to a variable that appears in an expression in the exiting transition and that was assigned a value in the entering transition with the expression it was assigned in the entering transition.

Figure 5.7 shows an example model containing a choice state, and Figure 5.8 shows the equivalent model after the choice state has been removed, and its compound transitions have been replaced with new transitions with merged labels. The modified model includes two new transitions, t4 and t5, which are the combinations of transitions t1, t2 and t1, t3, respectively, from the original model. To create the merged transition t4, which is the combination of transitions t1 and t2, we update t2's guard condition (x<5) by replacing x with x+1 to reflect the assignment x:=x+1 made by transition t1. This substitution simulates the evaluation of t2's guard after t1's variable assignment. This same substitution (x with x+1) is made when creating merged transition t5's guard. This type of model preprocessing allows us to simulate choice states and compound transitions without changing the semantics of the model.

5.3.2 Rhapsody

Rhapsody, developed by I-Logix[17], also follows UML-OMG semantics. We use the same template-semantics description we used for RoseRT to capture its semantics. The biggest difference between Rhapsody and RoseRT is that Rhapsody supports both interleaving and parallel composition. Thus, we created two versions of each benchmark model in Rhapsody, one that uses interleaving composition and

one that uses parallel composition. For each model created in Rhapsody, we created an equivalent template-semantics model that matched the Rhapsody model's semantics, to act as input to our code generators.

5.3.3 BetterState

BetterState[32] is a statecharts[12] variant with a simple semantics. It has no support for internal events, and external events persistent for only a single micro-step of the model's execution before being thrown away. BetterState uses simple semantics, sensing the environment for input at the end of every micro-step. The only composition operator supported by BetterState is parallel composition.

A partial template-semantics description simulating BetterState's semantics is shown below. The state- and variable-related parameters are unchanged from those of RoseRT.

- *macro semantics : simple*
  The environment is sensed at the beginning of every micro-step of the model.
- *en_events(ss,τ) : trig(τ) $\subseteq$ ss.Ia*
  A transition is triggered only by external events in Ia.
- *next_Ia(ss,τ,Ia') : Ia'=$\emptyset$*
  When a transition executes, all previously sensed external events are thrown away.
- *reset_Ia(ss,I) : I.gen*
  At the start of a macro-step, Ia is set to the events sensed from the environment

5.3.4 SmartState

SmartState[4] is another statecharts variant whose semantics are shown in Table 5.2. We previously described SmartState's semantics in detail in Section 2.1.4. To briefly review, SmartState has support for both internal and external events.

next_IE(ss,τ,IE')	ss.Ia $\neq \emptyset \Rightarrow$ IE'=gen(τ) $\wedge$ ss.Ia $= \emptyset \Rightarrow$ IE'=ss.IE\trig(τ)
reset_IE(ss,τ)	$\emptyset$
next_Ia(ss,τ,Ia')	Ia'=$\emptyset$
reset_Ia(ss,I)	I.gen
en_events(ss,τ)	trig(τ) $\subseteq$ ss.IE $\cup$ ss.Ia
macro semantics	stable

Table 5.2: SmartState Semantics

Model	Rose RT (seconds)	Optimizied Deterministic (seconds)	Deterministic (seconds)	Nondeterministic (seconds)
JVM startup	1.1	0.50	0.50	0.50
PingPongx1	4.1	4.4 (x1.07)	9	13
PingPongx2	5.8	8.3 (x1.43)	17	28
PingPongx8	42.5	55 (x1.29)	172	358
MutualEx	12	14.5 (x1.21)	30	40

Table 5.3: RoseRT Results (seconds)

External events are sensed at the start of a macro-step and remain active until they trigger a transition or until the end of the macro-step, whichever comes first. Internal events can be generated only by the first transition that an HTS executes at the start of a macro-step. This initial transition will always be triggered by an external event. An internal event may trigger at most one transition in the same macro-step in which it was generated. The only composition operator is parallel composition.

5.3.5 Results

The execution times of the commercial tools' generated code, compared to the execution times of our generated code for the benchmark models, are shown in Tables 5.3, 5.5, and 5.6. The measurements are given in seconds and were taken on a Sun Ultra-AXi2 running SunOS 5.8.

Table 5.3 shows the execution times for RoseRT's generated code compared to the execution times of three different implementations created by different modes of

code generation possible with the CGG. The first mode is optimized-deterministic code generation. This mode uses the techniques described in Chapter 4 to eliminate nondeterminism in the model and also has some optimization flags set to disable support for dealing with shared-variable conflicts (since RoseRT doesn't support shared variables). The second mode, regular-deterministic generation shown in the fourth column, includes support for dealing with shared variable conflicts, even though this support is not needed for RoseRT models. The third mode, nondeterministic generation shown in the rightmost column, uses the pseudo-nondeterministic mapping that faithfully preserves a model's nondeterminism.

A baseline test (JVM startup) was performed to measure the time it takes the Java virtual machine (JVM) to startup when running code generated by the tools. This was done by creating code for an empty model (i.e., a model with a single state, which is both a start state and a final state). RoseRT's generated code takes more than twice as long to start up, probably because it forces the JVM to load extra library classes for threading support. The execution times given for the other benchmark models include the time taken for the JVM to start up. The overall results are not surprising. Our optimized deterministic code is twice as fast as our unoptimized code, and over four times as fast as our pseudo-nondeterministic code. RoseRT's code runs faster than our generated code on all of the benchmark models. Our optimized-deterministic generated code is not too far behind, taking on average 1.25 times as long as RoseRT's code to execute. As expected, on deeper hierarchies (PingPongx2 and PingPongx8), our code generation becomes less efficient, even with the flattening optimizations. This is because RoseRT's generated code doesn't conduct a global enabledness check in every micro-step like our generated code does; it instead conducts an enabledness check based on the event at the head of the global queue.

We mentioned above that to fully match RoseRT's semantics, the template-semantics description would need to use stable macro-step semantics to capture choice states. We described above how we simulate RoseRT's choice states by collapsing compound transitions and by employing simple macro-step semantics. RoseRT's code generator performs a comparable optimization: if a transition enters

Model	Rose RT (seconds)	Optimized Deterministic Gen-Stable (seconds)
JVM startup	1.1	0.50
PingPongx1	4.1	7.5 (x1.83)
PingPongx2	5.8	21.5 (x3.70)
PingPongx8	42.5	360 (x8.47)
MutualEx	12	23.9 (x1.99)

Table 5.4: Stable Semantics for RoseRT (seconds)

a choice state, the generated code immediately determines which of the choice state's outgoing transitions to execute next, instead of checking the entire model for a transition to execute next.

If we model choice states as compound transitions (i.e., a sequence of micro-steps) and we use stable-semantics for our macro-step execution semantics (to avoid sensing the environment while in the middle of a compound transition), then the generated code is correct, but highly inefficient, as shown in Table 5.4. This table shows the performance of our optimized-deterministic code generated using a template-semantics description that is similar to the one given for RoseRT's semantics in Section 5.3.1, except that the macro-step semantics is switched to stable semantics. This generated code, takes, on average, four times as long as RoseRT's generated code to execute. The reason for this drop in efficiency is that every macro-step contains an extra micro-step, in which the model does nothing but an enabledness check to detect that it is stable (i.e., no transitions are enabled). When we used simple semantics, the generated code always sensed the environment after every micro-step, thereby avoiding the extra enabled check.

Table 5.5 shows the results of two studies that compare our generated code against Rhapsody's generated code. Columns two and three compare Rhapsody's generated code against our generated code (Gen-para), when all benchmark models are built using only parallel composition; and columns four and five compare Rhapsody's generated code and our generated code (Gen-intrl) when the benchmark models are built using only interleaving composition. In both cases, our code generators generate optimized-deterministic code for the benchmark models.

Model	Rhapsody Parallel (seconds)	Gen-para (seconds)	Rhapsody Interleaving (seconds)	Gen-intrl (seconds)
JVM startup	0.70	0.50	0.70	0.50
PingPongx1	5.7	5.6 (x0.98)	6.3	4.4 (x0.70)
PingPongx2	12.2	14.6 (x1.20)	10.8	8.3 (x0.77)
PingPongx8	85	240 (x2.82)	37.5	55 (x1.46)
MutualEx	16	18 (x1.13)	14.4	14.5 (x1.00)

Table 5.5: Rhapsody Results (seconds)

Model	SmartState (seconds)	Gen-SS (seconds)	BetterState (seconds)	Gen-BS (seconds)
JVM startup	0.50	0.50	0.50	0.50
PingPongx1	7.7	6.0 (x0.78)	0.80	3.0 (x3.75)
PingPongx2	17.7	17.5 (x0.99)	1.2	6.0 (x5)
PingPongx8	300	335 (x1.12)	xx	xx
MutualEx	xx	xx	xx	xx

Table 5.6: SmartState and BetterState Comparisons (seconds)

In both comparisons, our generated code is faster or of comparable speed on the smaller models (PingPongx1, PingPongx2) and performs worse on the larger models (PingPongx8, MutualEx). As another observation, our generated code is more competitive on models that use interleaving composition as opposed to models that use parallel composition. On models with deep hierarchies of parallel operators, our generated code performs three times slower than Rhapsody's generated code. On average, our CGG-created generators produce code that runs 0.99 times as fast as Rhapsody's generated code when applied to benchmark models that use interleaving composition, and 1.67 times as fast as Rhapsody's generated code when applied to benchmark models that use parallel composition.

Table 5.6 shows the results of the comparison studies between our generated code and that of SmartState's and BetterState's code generators. The missing times ("xx"s) in the table are due to limitations in the trial versions of SmartState and BetterState that we were using (i.e., they limit the size of the models you can generate code from). Our generated code for SmartState's semantics (Gen-SS)

runs faster than that produced by SmartState's code generator on the shallower models, and runs only slightly slower than SmartState's generated code for the deeper models. This is due to SmartState's lack of optimizations. SmartState's code generator doesn't flatten either the state hierarchy or the compositional hierarchy like our generator does. BetterState, on the other hand, has a very simple semantics that allows for a highly optimized code generator. Our code generator created for BetterState's semantics (Gen-BS) produces code that runs much faster than the code generated by other CGG-created code generators, but still several orders of magnitude slower than BetterState's generated code.

Overall, the cost of the parameterized approach varies significantly with the semantics of the notation, the composition operators used, and the size of a model's composition hierarchy. On smaller hierarchies, our generated code is competitive with the code generated by the commercial tools, but on larger hierarchies our generated code falls off the pace. Larger hierarchies are costlier due to the cost of our global enabledness check. We have begun investigating the possibility of caching enabledness information between micro-steps as a potential solution to this problem. Our generated code also performs better on models that use interleaving composition as opposed to models that use parallel composition, due to interleaving's more natural match with Java semantics. In most of the benchmark models, our generated code is well within twice the runtime of the code generated by notation-specific commercial tools.

5.4 Extensibility

One of our goals was to structure the CGG in such a way that it could be extended to support new parameter values. The extendibility of the CGG hinges on the distribution of preprocessor directives in the CGG source code. These directives tell the compiler which parts of the CGG source are specific to particular parameter values.

Each apply (i.e., next_XX parameter), reset (i.e. reset_XX parameter), and enabling parameter has preprocessor directives in two locations in the CGG code.

The first location is in a procedure, in the CGG source, named after the parameter; this procedure converts each value for that parameter to Java code (the procedure includes one preprocessor directive per parameter value). The second location is in a procedure in the CGG code named after the snapshot element the parameter is manipulating; this procedure creates the abstract data type (e.g., set, list, queue) that stores the contents of the associated snapshot element. When extending the CGG to support a new template-semantics parameter value, the modeller needs to ensure that the code generator supports the abstract data type that the new parameter value uses to store its associated snapshot element.

For example, to add a new parameter value for the next_IE parameter, in which IE is modelled as a set of events, the NextIE() procedure in the CGG source needs to be extended with a new preprocessor directive to generate Java for the new parameter value, and the printIE() procedure may need to be modified to output Java for a set abstract data type, if the CGG does not already support a set data type with the operations that the new parameter needs.

The macro_semantics, pri, and resolve parameters are each implemented in a single location in the CGG source code; this location needs to be extended to handle a new parameter value (i.e., one new preprocessor directive).

It is more difficult to extend the CGG to support a new composition operator. One needs to write the code to generate Java for the new operator's execute() and enabled_check() methods. This involves specifying how the operator will react if the various execution constraints that other operators can impose are imposed on this operator. If the new operator can impose new types of constraints, as a result of its execution decisions, then all of the other operators' execute() methods will have to be modified to process these constraints.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

We have described a tool for parameterized code generation from template semantics. Our tool creates a Java code generator for a model-based notation from the notation's template-semantics description. We have demonstrated that models using several different composition operators can be translated to Java code. We have also shown several techniques to optimize the generated code, by flattening the state and composition hierarchies and by using statically computed information. We have described how to deal with nondeterminism in a model when generating code. We compared our generated code to that of commercial tools to get a sense of the cost of supporting parameterized code generation. On average, our generated code was well within twice the runtime of code generated by commercial notation-specific code generators. We also demonstrated how a model's generated implementation has the same execution semantics as the model.

6.2 Limitations

Extending the CGG requires manually editing the CGG source code. It is time consuming to extend the CGG to support a new composition operator. Adding

support for a new parameter value, while much easier than adding a new composition operator, still requires making changes to the CGG code. It should be possible to automate much of the work needed to add a new parameter value to the CGG, to make this task quicker and easier.

Our generated code does not run as quickly as the code generated by notation-specific code generators. This is because the optimizations included in our code generators are limited to those optimizations that can be performed on all notations in the family of notations supported by template semantics. If the efficiency of the generated code is the modeller's primary concern, then notation-specific tools may be more appropriate.

We made an argument for the correctness of the generated code by testing and inspecting the code generated from a test suite of models. We did not show a formal correctness proof demonstrating that a model matches its generated code. Thus, we are limited to making informal claims about the correctness of the generated code.

6.3 Related Work

Many of the individual tools used in the benchmark comparison support some limited parameterization in their code generators. None of them support full parameterization of the modelling notation's semantics as we do. Their parameters are concerned with features of the generated code. RoseRT and Rhapsody both have options to choose the language (C/C++/Java) of the generated code or whether the code will be single-threaded or multi-threaded. SmartState has options that effect the structure of the generated code (generate for efficiency vs re-use). BetterState supports choices among priority mechanisms and some simple notational semantics options. For example, it allows the user to choose whether the state-exit code for a transition's source or the state-entry code for a transition's destination is executed first when a transition fires. These parameters allow the modeller control over the notation and code generation process, but our tool gives the modeller much more power in this regard.

There is some related work investigating the translation of a specific modelling notation (mostly a statecharts variant) to code in a given programming language. For example, SCOPE[31], built by Andrzej Wasowski based on the results in his PhD thesis, generates optimized C code for embedded systems from a statechart description of the embedded system. This type of code generation is unique in that the generated code is designed to run on a platform with tight memory constraints (i.e., 8-16 kilobytes of memory). SCADE[30], developed by Esterel technologies, is another commercial tool used extensively in the aviation industry that supports both C and Ada code generation for embedded systems. SCADE's notation is based on LUSTRE[11] ,which was developed for modelling reactive systems.

Statecharts code generation techniques that are not aimed at embedded systems have more flexibility in how they structure the generated code. Perhaps the most well known structure is the state design pattern[9, 3]. This technique maps the state and composition hierarchies in the model into an inheritance hierarchy in an object-oriented language (i.e., a state in the model maps to a class that is a subclass of the state's superstate's class, and a child of a composition operator maps to a class that is a subclass of the child's parent operator's class). Each transition in the model maps to a virtual function declared in the root class of the inheritance hierarchy and overridden by the class representing the transition's source state. Calling the function associated with a given transition on a state object executes the transition. SmartState takes this approach. This is an elegant way of encoding the state and composition hierarchies in an object-oriented language. This technique results in more readable code, but due to the codes object-oriented design and reliance on virtual functions, it has high memory usage. Quantum programming[29] is a less object-oriented approach using pointers to functions. The idea is to remove the cost of virtual function calls from the state design pattern. An even less object-oriented approach is to make heavy use of nested switch statements. Our code generators as well as both Rhapsody's and Rational Rose's code generators take this approach. The generated code is less readable, but uses less memory and executes faster. Another interesting approach used by Bond and Goguen at AT&T was to design a statecharts variant whose semantics is well suited to the generation of efficient

code. The result is the notation ECLIPSEcharts[6] and a Java code generator for ECLIPSEcharts. This notation avoids features of other statecharts variants, such as nondeterminism, that are difficult to represent in Java. ECLIPSEcharts is used to specify telephony features whose generated coded is deployed commercially by AT&T.

Model-Driven Architecture (MDA)[26], developed by the Object Management Group (OMG), is a set of standards related to code generation. MDA states that systems should be specified by platform-independent models (PIMs) written in a modelling notation described in OMG's Meta-Object-Facility (MOF)[25]. MDA translators are built to translate a PIM into an equivalent platform-specific model (PSM) for a given platform. MDA doesn't describe how to build these translators; it is more concerned with building an argument for their usefulness.

Meta-modelling technology, such as OMG's MOF, offers a form of parameterized code generation. Template semantics is essentially a meta-modelling notation for describing model-based notations. Given a template semantics description (a meta-model) of a notation we can produce a code generator for that notation. MOF is more general then template semantics in that it's not restricted to model-based notations. It has the ability to describe any modelling notation. The ability to take a MOF model of a notation and automatically create a code generator for this notation is limited. OMG talks of automated translations to interface definitions languages (IDLs) but not fully executable source. OMG is also developing a new standard called Query/View/Transformation (QVT) for defining transformations among notations and languages described in MOF [27]. QVT is a developing standard and it's not clear how much work it will be to define a transformation in QVT from a given notation to a source language and whether it will be possible to automatically create a code generator from the QVT description.

6.4 Future Work

It may be useful to support translations to other languages besides Java. A translation to C++ would require minimal syntactic changes to the generated code, due

to the similarities between C++ and Java syntax. The generated code does not use any advanced object-oriented techniques that couldn't be simulated by non-object-oriented languages. Thus, translations to languages such as C are also possible with minimal effort.

We would like to add support for automating the addition of new template-parameter values. If we scoped the language that the parameter values are written in, to a notation such as set theory, we could explore having the CGG create a code generator directly from these set-theory descriptions. As long as the input language for parameter values is scoped appropriately, the work required to implement this change would be at the implementation level. The CGG would need to parse the parameter-value expressions and translate them into equivalent Java code. With the current CGG, the modeller can only select from a pre-defined list of values for each parameter, and adding a new parameter value to this list is non-trivial.

We would also like to add support for multi-threaded-code generation. Currently, the generated code is entirely single threaded. For the reasons discussed in Section 3.1, mapping each HTS to its own thread is far too expensive. However, we would like to allow the modeller some flexibility to assign HTSs to threads. We would need to introduce restrictions on which HTSs are allowed to execute in separate threads. For example, all descendant HTSs of a synchronization or parallel operator would need to execute in the same thread, to prevent the costly overhead in synchronizing threads, as described in Section 3.1. The modeller would only be allowed to assign HTSs to different threads if they have no common ancestor operator that is a synchronization or parallel operator. If these restrictions are followed, then the scheduling methodology could be maintained within each thread and we would avoid excessive synchronization among the threads.

There is a lot of room to optimize the generated code while still maintaining the same scheduling methodology. Specifically, the enabled_check() method currently does a full enabledness check of the entire composition tree even if only a single variable has changed value since the last enabledness check was performed. In this case, all that really needs to be checked are the HTSs and transitions whose enabledness could have potentially changed as a result of that single variable chang-

ing value. Thus, the generated code could cache enabledness information between micro-steps instead of starting from scratch each time. This change would require a data-flow analysis to determine relationships amongst variables, events, and the transitions that use them as guards and triggers. Such an optimization would help our generated code to perform as well on deeper hierarchies as code generated by commercial tools, by avoiding unnecessary enabledness checking.

We would like to investigate optimizations that are possible due to relationships amongst template-parameter values. Currently, our code generators are optimized according to individual parameter values (i.e., we generate optimized code for a given parameter value independently of the values of the other template parameters). There may also be optimizations that are possible due to particular combinations of template-parameter values. For example, if en_events states that a transition's triggering event must be in the Ia or IE snapshot element for it to be enabled, and if none of the next_IE, reset_IE, next_Ia, and reset_Ia parameters empty their respective snapshot elements, then the generated code needs only one set of current events and this set is manipulated by the generated code for the next_IE, reset_IE, next_Ia, and reset_Ia parameters. Currently, our code generator always uses separate sets of events for the Ia and IE snapshot elements when generating code for the next_XX and reset_XX parameters since this ensures that the code for these parameters will work for any value of the other template parameters. Finding other relationships between parameter values would allow for other optimizations.

Appendix A

Template Parameter Values

This appendix lists the implemented values for each template-semantics parameter.

A.1 Enabling Parameters

- *en_cond*
  - The variable values in AV must satisfy the transition's guard condition
  - The variable values in AVa must satisfy the transition's guard condition
- *en_event*
  - The transition must have a triggering event in IEa
  - The transition must have a triggering event in Ia or IE
  - The transition must have a triggering event in Ia or IEa
  - The transition must have a triggering event at the head of the queue in IE (i.e., one queue per HTS in the IE snapshot element)
  - The transition must have a triggering event at the head of the global queue in Q (i.e., one queue per model)
  - The transition must have a triggering event at the head of proper queue in Q (i.e., Q contains a set of named queues)
- *en_state*
  - The transition's source state must be in CSa

  - The transition's source state must be in CS
  - True, for any transition (i.e., this notation doesn't make use of states)

A.2 Apply Parameters

- *next_CS*
  - The CS snapshot element is set to the transition's destination state along with any default substates, or ancestor superstates of that destination state (preserving the state hierarchy).
  - The CS snapshot element is set to the transitions destination state.
- *next_CSa*
  - Empty CSa
  - Leave CSa unchanged
- *next_AV*
  - The variable values in AV are updated with the transition's actions (i.e., execute the actions). If a variable is assigned a value in more then one action, only the last such assignment takes effect.
  - If a variable is assigned a value in more then one action, the actions are executed in the order they are given in the transition. Thus, later assignments to a variable are evaluated with respect to, and potentially overwrite, earlier assignments to the same variable.
- *next_AVa*
  - AVa snapshot element is unchanged
- *next_Q*
  - Remove the head of the queue in Q and enqueue the events the transition generates
  - Remove the head of the proper named queue in Q and enqueue the events the transition generates onto the proper named queue (i.e., Q is a set of names queues, and transitions explicitly state which queue and event they are triggered on, and which queue an event is to be generated on)

- *next_Qa*
  - Empty Qa
  - Leave Qa unchanged
- *next_IE*
  - The IE snapshot element is set to the events generated by the transition
  - The events generated by the transition are added to the IE snapshot element
  - The transition's triggering event is removed from IE, and the events it generates are added to IE
  - Dequeue the queue in IE, and enqueue the events the transition generates to the queue
- *next_IEa*
  - Add the transitions triggering events to IEa
  - Add the events generated by the transition to IEa if Ia is empty
  - Set IEa to empty
- *next_Ia*
  - Make no change to Ia
  - Add the events generated by the transition to Ia
  - Remove the transition's triggering event(s) from Ia
  - Set Ia to empty

A.3 Reset Parameters

- *reset_CS*
  - Make no change to CS
- *reset_CSa*
  - The CSa snapshot element is set to the contain all the states in the CS snapshot element

  - Empty CSa

- *reset_Q*
  - Enqueue the events sensed from the environment onto the back of the global queue in Q
  - Dequeue the global queue in Q and enqueue the events sensed from the environment onto the back of the global queue
  - Enqueue the events sensed from the environment onto the proper queue in Q (i.e., we sense (queue, event) pairs from the environment)

- *reset_Qa*
  - Set Qa to the head of a random queue in Q
  - Set Qa to the head of the global queue in Q
  - Set Qa to empty

- *reset_IE*
  - Make no change to IE
  - Set IE to empty
  - Enqueue events sensed from the environment onto the queue in IE

- *reset_IEa*
  - Set IEa to the head of a random queue in Q
  - Set IEa to the head of the global queue in Q
  - Set IEa to empty

- *reset_AV*
  - Make no change to AV
  - Update the variables in AV with values sensed from the environment

- *reset_AV*
  - Set the variable values in AVa to be same as those in AV
  - Update the variables in AVa with values sensed from the environment

- *reset_Ia*

  - Set Ia to the events sensed from the environment
  - Add the events sensed from the environment to those already in Ia
  - set Ia to empty

A.4 Miscellaneous Parameters

- *macro_semantics*
  - Simple semantics: A macro-step is at most a single micro-step in length (execute the reset parameters after every micro-step)
  - Stable semantics: A macro-step is a series of micro-steps ending in a stable snapshot (one in which no transition is enabled)
- *pri*
  - Explicit priority: Transitions have explicit priority values associated with them
  - Highest ranked source: Transitions whose source states are deeper in the state hierarchy have higher priority
  - Lowest ranked source: Transitions whose source states are higher in the state hierarchy have higher priority
- *resolve*
  - nondeterministically choose which value a shared variable will take on after being assigned multiple values in parallel

Appendix B

Ground Traffic Control Generated Java

This appendix contains some of the generated code for the ground -traffic control system introduced in Section 5.2. We show the code for the rendezvous operator (Appendix B.3), environmental synchronization operator (Appendix B.6) and one of the interleaving operators (Appendix B.4). We also show the generated code for the controller HTS (Appendix B.2), the Variable (AV) snapshot element class (Appendix B.5), and the system class which creates the composition tree (Appendix B.1).

B.1 GeneratedSystem.java

```
class GeneratedSystem
{
  public static void main(String[] args)
  {
    //create the required global snapshot elements
    Procs PD = new Procs();
    Vars AV = new Vars();
    GlobalCS GCS = new GlobalCS();
    GlobalIa GIa = new GlobalIa();
    EnvSensor ES = new EnvSensor(AV,GIa);

    //build the composition tree and established references to global snapshot elements
    microRendSync_rend rend;
    HTS_controller_system controller_system = new HTS_controller_system(PD,AV,GCS,GIa);
    HTS_runway1_system runway1_system = new HTS_runway1_system(PD,AV,GCS,GIa);
    HTS_runway2_system runway2_system = new HTS_runway2_system(PD,AV,GCS,GIa);
    microIntl_intrl3 intrl3 = new microIntl_intrl3(AV ,runway1_system,runway2_system);
    intrl3.storeSubComponentIEs(runway1_system.IE,runway2_system.IE);
```

```
    HTS_taxiway1_system taxiway1_system = new HTS_taxiway1_system(PD,AV,GCS,GIa);
    HTS_taxiway2_system taxiway2_system = new HTS_taxiway2_system(PD,AV,GCS,GIa);
    HTS_taxiway3_system taxiway3_system = new HTS_taxiway3_system(PD,AV,GCS,GIa);
    microIntl_intrl2 intrl2 = new microIntl_intrl2(AV,taxiway1_system,taxiway2_system
        ,taxiway3_system);
    intrl2.storeSubComponentIEs(taxiway1_system.IE,taxiway2_system.IE,taxiway3_system.IE);

    microEnvSync_env env = new microEnvSync_env(AV,Gintrl3 ,intrl2);
    env.storeSubComponentIEs(runway1_system.IE,runway2_system.IE,taxiway1_system.IE,
        taxiway2_system.IE,taxiway3_system.IE);

    rend = new microRendSync_rend(AV,GCS,GIa,ES,controller_system,env);
    rend.storeSubComponentIEs(controller_system.IE,runway1_system.IE,runway2_system.IE,
        taxiway1_system.IE,taxiway2_system.IE,taxiway3_system.IE);
    ES.storeSubComponentIEs(controller_system.IE,runway1_system.IE,runway2_system.IE,
        taxiway1_system.IE,taxiway2_system.IE,taxiway3_system.IE);

    //start the execution loop
    ES.senseEnv();
    for(;;)
    {
        rend.run();
    }
  }
}
```

B.2 HTS_controller_system.java

```
import java.util.HashSet;
import java.util.HashMap;
import java.util.Vector;

class HTS_controller_system
{
   //snapshot elements
   private Vars AV;
   public Events IE;
   private AuxI Ia;
   private Procs PD;
   private Vector CS;

   //enabledness info storing members
   private HashMap rend_TTrans;
   private Integer enabled_transition;
   private double max_pri;

   //constants
   public static final Integer state_controller = new Integer(0);
   public static final Integer state_controlleridle = new Integer(1);
   public static final Integer state_landingonrw1 = new Integer(2);
   public static final Integer state_taxingon1 = new Integer(3);
   private static final Integer trans_ct1 = new Integer(0);
   private static final Integer trans_ct4 = new Integer(1);
   private static final Integer trans_ct2 = new Integer(2);
   private static final Integer trans_ct13 = new Integer(3);
   private static final Integer trans_ct7 = new Integer(4);
```

```
private static final Integer trans_ct5 = new Integer(5);
private static final Integer trans_ct6 = new Integer(6);
private static final Integer trans_ct14 = new Integer(7);
private static final Integer trans_ct8 = new Integer(8);
private static final Integer trans_ct10 = new Integer(9);
private static final Integer trans_null = new Integer(-1);

public HTS_controller_system(Procs pd,Vars av,GlobalCS gcs,GlobalIa gia)
{
   AV = av;
   PD = pd;
   IE = new Events();
   Ia = new AuxI();
   CS = new Vector();
   gcs.CS_controller_system=CS;
   gia.Ia_controller_system=Ia;
   Ia = ia;
   rend_TTrans = new HashMap();
   CS.add(state_controller);
   CS.add(state_controlleridle);
}

public void setState(Integer state)
{
   CS.add(state);
}
public void reset()
{
   enabled_transition=trans_null;
   rend_TTrans.clear();
}
public void initCS()
{
   CS.add(state_controller);
   CS.add(state_controlleridle);
}
public void resetCS()
{
   CS.clear();
}

public void execute(Integer rend_TEvent,HashSet outputs)
{
   //choose a transition to execute
   Integer trans = new Integer(0);
   if(rend_TEvent != null)
   {//execute a transition which is triggered by rendezvous event rend_TEvent
      trans = (Integer)rend_TTrans.get(rend_TEvent);
   }
   else
   {
      trans = enabled_transition;
   }

   //execute the chosen transition (Next_XX parameters encoding)
   switch(trans.intValue())
   {
      case 0: {
         CS.remove(state_controlleridle);
```

```
   CS.add(state_landingonrw1);
   IE.genEvent(IE.eventlandon1);
   outputs.add(IE.eventlandon1);
   IE.eatEvent(IE.eventreqland);
   Ia.eatEvent(IE.eventreqland);
   break;
}
case 1: {
   CS.remove(state_landingonrw1);
   CS.add(state_taxingon1);
   IE.genEvent(IE.evententertw1);
   outputs.add(IE.evententertw1);
   IE.eatEvent(IE.eventreqtaxi);
   Ia.eatEvent(IE.eventreqtaxi);
   break;
}
case 2: {
   CS.remove(state_controlleridle);
   CS.add(state_controlleridle);
   IE.genEvent(IE.eventtakeoffon2);
   outputs.add(IE.eventtakeoffon2);
   IE.eatEvent(IE.eventreqtakeoff);
   Ia.eatEvent(IE.eventreqtakeoff);
   break;
}
case 3: {
   CS.remove(state_controlleridle);
   CS.add(state_controlleridle);
   IE.genEvent(IE.eventlandon2);
   outputs.add(IE.eventlandon2);
   IE.eatEvent(IE.eventreqland);
   Ia.eatEvent(IE.eventreqland);
   break;
}
case 4: {
   CS.remove(state_taxingon1);
   CS.add(state_controlleridle);
   IE.genEvent(IE.eventcrossrw1);
   outputs.add(IE.eventcrossrw1);
   IE.eatEvent(IE.eventreqcrossrw1);
   Ia.eatEvent(IE.eventreqcrossrw1);
   break;
}
case 5: {
   CS.remove(state_landingonrw1);
   CS.add(state_taxingon1);
   IE.genEvent(IE.evententertw2);
   outputs.add(IE.evententertw2);
   IE.eatEvent(IE.eventreqtaxi);
   Ia.eatEvent(IE.eventreqtaxi);
   break;
}
case 6: {
   CS.remove(state_landingonrw1);
   CS.add(state_taxingon1);
   IE.genEvent(IE.evententertw3);
   outputs.add(IE.evententertw3);
   IE.eatEvent(IE.eventreqtaxi);
   Ia.eatEvent(IE.eventreqtaxi);
```

```
            break;
         }
         case 7: {
            CS.remove(state_controlleridle);
            CS.add(state_controlleridle);
            IE.genEvent(IE.eventtakeoffon1);
            outputs.add(IE.eventtakeoffon1);
            IE.eatEvent(IE.eventreqtakeoff);
            Ia.eatEvent(IE.eventreqtakeoff);
            break;
         }
         case 8: {
            CS.remove(state_taxingon1);
            CS.add(state_taxingon1);
            IE.genEvent(IE.eventlandon2);
            outputs.add(IE.eventlandon2);
            IE.eatEvent(IE.eventreqland);
            Ia.eatEvent(IE.eventreqland);
            break;
         }
         case 9: {
            CS.remove(state_taxingon1);
            CS.add(state_taxingon1);
            IE.genEvent(IE.eventtakeoffon2);
            outputs.add(IE.eventtakeoffon2);
            IE.eatEvent(IE.eventreqtakeoff);
            Ia.eatEvent(IE.eventreqtakeoff);
            break;
         }
      }
   }

   public void enabled_check(HashMap rend_TEvents,HashMap rend_GEvents,Doub other,
                             Bool final_state)
   {
      reset();
      max_pri = -1;
      //find the single highest priority enabled transition
      for(int k = 0;k<CS.size();k++)
      {
         switch(((Integer)CS.elementAt(k)).intValue())
         {
            case 0:{
               break;
            }
            case 1:{
               if((Ia.checkEvent(IE.eventreqland) || IE.checkEvent(IE.eventreqland))
                  && (!AV.getbusyrw1()&&!AV.getbusyrw2())&& max_pri <= 4.0)
               {
                  enabled_transition = trans_ct1;
                  max_pri = 4.0;
               }
               else if((Ia.checkEvent(IE.eventreqland) || IE.checkEvent(IE.eventreqland))
                       && (!AV.getbusyrw1()&&!AV.getbusyrw2()&&!AV.getbusyrw2()&&!AV.getbusytw1()
                       && !AV.getbusytw2()&&!AV.getbusytw3())&& max_pri <= 3.0)
               {
                  enabled_transition = trans_ct13;
                  max_pri = 3.0;
               }
```

```
            else if((Ia.checkEvent(IE.eventreqtakeoff) || IE.checkEvent(IE.eventreqtakeoff)) &&
                   (!AV.getbusyrw1()&&!AV.getbusyrw2())&& max_pri <= 2.0)
            {
               enabled_transition = trans_ct14;
               max_pri = 2.0;
            }
            else if((Ia.checkEvent(IE.eventreqtakeoff) || IE.checkEvent(IE.eventreqtakeoff)) &&
                   (!AV.getbusyrw1()&&!AV.getbusyrw2()&&!AV.getbusytw1()&&!AV.getbusytw2()&&
                   !AV.getbusytw3())&& max_pri <= 1.0)
            {
               enabled_transition = trans_ct2;
               max_pri = 1.0;
            }
            break;
         }
         case 2:{
            if((Ia.checkEvent(IE.eventreqtaxi) || IE.checkEvent(IE.eventreqtaxi)) &&
              (!AV.getbusytw3())&& max_pri <= 0.0)
            {
               enabled_transition = trans_ct6;
               max_pri = 0.0;
            }
            else if((Ia.checkEvent(IE.eventreqtaxi) || IE.checkEvent(IE.eventreqtaxi)) &&
                   (!AV.getbusytw2())&& max_pri <= 0.0)
            {
               enabled_transition = trans_ct5;
               max_pri = 0.0;
            }
            else if((Ia.checkEvent(IE.eventreqtaxi) || IE.checkEvent(IE.eventreqtaxi)) &&
                   (!AV.getbusytw1())&& max_pri <= 0.0)
            {
               enabled_transition = trans_ct4;
               max_pri = 0.0;
            }
            break;
         }
         case 3:{
            if((Ia.checkEvent(IE.eventreqcrossrw1) || IE.checkEvent(IE.eventreqcrossrw1)) &&
              (!AV.getbusyrw1())&& max_pri <= 3.0)
            {
               enabled_transition = trans_ct7;
               max_pri = 3.0;
            }
            else if((Ia.checkEvent(IE.eventreqland) || IE.checkEvent(IE.eventreqland)) &&
                   (!AV.getbusyrw1()&& !AV.getbusyrw2())&& max_pri <= 2.0)
            {
               enabled_transition = trans_ct8;
               max_pri = 2.0;
            }
            else if((Ia.checkEvent(IE.eventreqtakeoff) || IE.checkEvent(IE.eventreqtakeoff)) &&
                   (!AV.getbusyrw1()&& !AV.getbusyrw2())&& max_pri <= 1.0)
            {
               enabled_transition = trans_ct10;
               max_pri = 1.0;
            }
            break;
         }
      }
   }
```

```
      //communicate the enabledness information for the single
      //enabled transition (if there is one)
      switch(enabled_transition.intValue())
      {
         case 0:{
            rend_GEvents.put(IE.eventlandon1,new Doub(4.0));
            break;
         }
         case 1:{
            other.value = 0.0;
            break;
         }
         case 2:{
            rend_GEvents.put(IE.eventtakeoffon2,new Doub(1.0));
            break;
         }
         case 3:{
            rend_GEvents.put(IE.eventlandon2,new Doub(3.0));
            break;
         }
         case 4:{
            rend_GEvents.put(IE.eventcrossrw1,new Doub(3.0));
            break;
         }
         case 5:{
            other.value = 0.0;
            break;
         }
         case 6:{
            other.value = 0.0;
            break;
         }
         case 7:{
            rend_GEvents.put(IE.eventtakeoffon1,new Doub(2.0));
            break;
         }
         case 8:{
            rend_GEvents.put(IE.eventlandon2,new Doub(2.0));
            break;
         }
         case 9:{
            rend_GEvents.put(IE.eventtakeoffon2,new Doub(1.0));
            break;
         }
      }
   }
}
```

B.3 microRendSync_rend.java

```
import java.util.Vector;
import java.util.HashSet;
import java.util.HashMap;

class microRendSync_rend
{
```

```
//child components
private microEnvSync_env right;
private HTS_controller_system left;

//this is the root component — it stores the global snapshot elements
private Vars AV;
private Events IE;
private GlobalIa GIa;
private GlobalCS GCS;
private EnvSensor ES;

//member variables for storing enabledness info
private HashSet outputs = new HashSet();
private HashSet tempOutputs = new HashSet();
private double other_left,other_right;
private HashMap leftRend_TEvents,leftRend_GEvents,rightRend_TEvents,rightRend_GEvents;
private HashMap rightEnv_Events,leftEnv_Events;
private HashSet leftInterr,rightInterr;
private boolean left_goes = true;
private Doub o_left,o_right;
private Bool f_left,f_right;

public Events IEcontroller_system;
public Events IErunway1_system;
public Events IErunway2_system;
public Events IEtaxiway1_system;
public Events IEtaxiway2_system;
public Events IEtaxiway3_system;

public microRendSync_rend(Vars av,GlobalCS gcs,GlobalIa gia,EnvSensor es,
                          HTS_controller_system l, microEnvSync_env r)
{
  leftRend_GEvents = new HashMap();
  leftRend_TEvents= new HashMap();
  rightRend_GEvents = new HashMap();
  rightRend_TEvents= new HashMap();
  rightEnv_Events  = new HashMap();
  leftEnv_Events = new HashMap();
  leftInterr = new HashSet();
  rightInterr = new HashSet();
  other_right = -1;
  other_left = -1;
  o_right = new Doub(-1);
  o_left = new Doub(-1);
  f_right = new Bool(false);
  f_left = new Bool(false);
  AV = av;
  IE= new Events();
  GCS = gcs;
  GIa = gia;
  ES = es;
  left = l;
  right = r;
}

public void initCS()
{
  left.initCS();
  right.initCS();
```

```
}

public void storeSubComponentIEs(Events iecontroller_system,Events ierunway1_system,Events
                                 ierunway2_system,Events ietaxiway1_system,Events
                                 ietaxiway2_system,Events ietaxiway3_system)
{
  IEcontroller_system = iecontroller_system;
  IErunway1_system = ierunway1_system;
  IErunway2_system = ierunway2_system;
  IEtaxiway1_system = ietaxiway1_system;
  IEtaxiway2_system = ietaxiway2_system;
  IEtaxiway3_system = ietaxiway3_system;
}
public void communicate(HashSet outputs,boolean which)
{
  if(which)
  {
    IEcontroller_system.genEvents(outputs);
  }
  else
  {
    IErunway1_system.genEvents(outputs);
    IErunway2_system.genEvents(outputs);
    IEtaxiway1_system.genEvents(outputs);
    IEtaxiway2_system.genEvents(outputs);
    IEtaxiway3_system.genEvents(outputs);
  }
}

//execute a single micro-step
public void run()
{
  left.enabled_check(leftRend_TEvents,leftRend_GEvents,o_left,f_left);
  right.enabled_check(rightRend_TEvents,rightRend_GEvents,o_right,f_right);
  if(f_left.value && f_right.value)
  {
    System.out.println("All HTSes have reached final states - exiting"); System.exit(1);
  }
  other_left = o_left.doubleValue();
  other_right = o_right.doubleValue();

  //Round Robin scheduling (left_goes indicates if the left component should
  //execute if both components are enabled)
  if(other_left>=0 && other_right>=0 && left_goes)
  {
    other_right = -1;
    left_goes = false;
  }
  else if(other_left>=0 && other_right>=0 && !left_goes)
  {
    other_left = -1;
    left_goes = true;
  }

  if(!execute(null,null,null,null,outputs))
  {
    //Reset
    ES.senseEnv();
  }
```

```
}

public boolean execute(Integer env_Event,Integer rend_GEvent,Integer rend_TEvent,
                       Integer interrupt,HashSet outputs)
{
    //Execute unconstrained (this is the root component, it has no
    //ancestors to impose constraints)

    boolean executed = false;
    //check if an rendezvous is ready to be executed (check in descending order
    //of event priority)
    if( (leftRend_GEvents.size() > 0 && rightRend_TEvents.size() > 0) ||
        (leftRend_TEvents.size() > 0 && rightRend_GEvents.size() > 0) )
    {
      if(leftRend_GEvents.containsKey(IE.eventlandon1) &&
         rightRend_TEvents.containsKey(IE.eventlandon1))
      {
        executed = true;
        left.execute(null,IE.eventlandon1,null,null,outputs);
        right.execute(null,null,IE.eventlandon1,null,tempOutputs);
        communicate(outputs,false);
        communicate(tempOutputs,true);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(rightRend_GEvents.containsKey(IE.eventlandon1) &&
              leftRend_TEvents.containsKey(IE.eventlandon1) )
      {
        executed = true;
        right.execute(null,IE.eventlandon1,null,null,outputs);
        left.execute(null,null,IE.eventlandon1,null,tempOutputs);
        communicate(outputs,true);
        communicate(tempOutputs,false);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(leftRend_GEvents.containsKey(IE.eventlandon2) &&
              rightRend_TEvents.containsKey(IE.eventlandon2)  )
      {
        executed = true;
        left.execute(null,IE.eventlandon2,null,null,outputs);
        right.execute(null,null,IE.eventlandon2,null,tempOutputs);
        communicate(outputs,false);
        communicate(tempOutputs,true);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(rightRend_GEvents.containsKey(IE.eventlandon2) &&
              leftRend_TEvents.containsKey(IE.eventlandon2)  )
      {
        executed = true;
        right.execute(null,IE.eventlandon2,null,null,outputs);
        left.execute(null,null,IE.eventlandon2,null,tempOutputs);
        communicate(outputs,true);
        communicate(tempOutputs,false);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(leftRend_GEvents.containsKey(IE.eventtakeoffon1) &&
```

```
        rightRend_TEvents.containsKey(IE.eventtakeoffon1)  )
{
  executed = true;
  left.execute(null,IE.eventtakeoffon1,null,null,outputs);
  right.execute(null,null,IE.eventtakeoffon1,null,tempOutputs);
  communicate(outputs,false);
  communicate(tempOutputs,true);
  outputs.addAll(tempOutputs);
  tempOutputs.clear();
}
else if(rightRend_GEvents.containsKey(IE.eventtakeoffon1) &&
       leftRend_TEvents.containsKey(IE.eventtakeoffon1)  )
{
  executed = true;
  right.execute(null,IE.eventtakeoffon1,null,null,outputs);
  left.execute(null,null,IE.eventtakeoffon1,null,tempOutputs);
  communicate(outputs,true);
  communicate(tempOutputs,false);
  outputs.addAll(tempOutputs);
  tempOutputs.clear();
}
else if(leftRend_GEvents.containsKey(IE.eventtakeoffon2) &&
       rightRend_TEvents.containsKey(IE.eventtakeoffon2)  )
{
  executed = true;
  left.execute(null,IE.eventtakeoffon2,null,null,outputs);
  right.execute(null,null,IE.eventtakeoffon2,null,tempOutputs);
  communicate(outputs,false);
  communicate(tempOutputs,true);
  outputs.addAll(tempOutputs);
  tempOutputs.clear();
}
else if(rightRend_GEvents.containsKey(IE.eventtakeoffon2) &&
       leftRend_TEvents.containsKey(IE.eventtakeoffon2)  )
{
  executed = true;
  right.execute(null,IE.eventtakeoffon2,null,null,outputs);
  left.execute(null,null,IE.eventtakeoffon2,null,tempOutputs);
  communicate(outputs,true);
  communicate(tempOutputs,false);
  outputs.addAll(tempOutputs);
  tempOutputs.clear();
}
else if(leftRend_GEvents.containsKey(IE.eventcrossrw1) &&
       rightRend_TEvents.containsKey(IE.eventcrossrw1)  )
{
  executed = true;
  left.execute(null,IE.eventcrossrw1,null,null,outputs);
  right.execute(null,null,IE.eventcrossrw1,null,tempOutputs);
  communicate(outputs,false);
  communicate(tempOutputs,true);
  outputs.addAll(tempOutputs);
  tempOutputs.clear();
}
else if(rightRend_GEvents.containsKey(IE.eventcrossrw1) &&
       leftRend_TEvents.containsKey(IE.eventcrossrw1)  )
{
  executed = true;
  right.execute(null,IE.eventcrossrw1,null,null,outputs);
```

```
          left.execute(null,null,IE.eventcrossrw1,null,tempOutputs);
          communicate(outputs,true);
          communicate(tempOutputs,false);
          outputs.addAll(tempOutputs);
          tempOutputs.clear();
        }
      } //if no sync to execute, check if one of the children are prepared to execute
      else if(other_left>=0)
      {
        executed = true;
        left.execute(null,null,null,null,outputs);
        communicate(outputs,false);
      }
      else if(other_right>=0)
      {
        executed = true;
        right.execute(null,null,null,null,outputs);
        communicate(outputs,true);
      }
     reset();
     return executed;
  }

  private void reset()
  {
    o_right.value=-1;
    o_left.value=-1;
    leftRend_TEvents.clear();
    leftRend_GEvents.clear();
    rightRend_TEvents.clear();
    rightRend_GEvents.clear();
  }
}
```

B.4 microIntl_intrl2.java

```
import java.util.Vector;
import java.util.HashSet;
import java.util.HashMap;

class microIntl_intrl2
{
  //This operators set of children (the operator has been flattened)
  private HTS_taxiway1_system child1;
  private HTS_taxiway2_system child2;
  private HTS_taxiway3_system child3;

  //member variables for storing children's enabledness information
  private Vars AV;
  public Events IE;
  private HashSet tempOutputs = new HashSet();
  private Doub o_child1;
  private Doub o_child2;
  private Doub o_child3;
  private HashMap child1Rend_TEvents,child1Rend_GEvents;
  private HashMap child2Rend_TEvents,child2Rend_GEvents;
  private HashMap child3Rend_TEvents,child3Rend_GEvents;
```

```
private HashMap child1Env_Events;
private HashMap child2Env_Events;
private HashMap child3Env_Events;
private int round_robin = 1;
private Bool f_child1;
private Bool f_child2;
private Bool f_child3;
public Events IEtaxiway1_system;
public Events IEtaxiway2_system;
public Events IEtaxiway3_system;
private HashSet outputs1 = new HashSet();
private HashSet outputs2 = new HashSet();
private HashSet outputs3 = new HashSet();

public microIntl_intrl2(Vars av,HTS_taxiway1_system c1,HTS_taxiway2_system c2,
                        HTS_taxiway3_system c3)
{
  child1Rend_GEvents = new HashMap();
  child1Rend_TEvents = new HashMap();
  child2Rend_GEvents = new HashMap();
  child2Rend_TEvents = new HashMap();
  child3Rend_GEvents = new HashMap();
  child3Rend_TEvents = new HashMap();
  child1Env_Events  = new HashMap();
  child2Env_Events  = new HashMap();
  child3Env_Events  = new HashMap();
  o_child1 = new Doub(-1);
  o_child2 = new Doub(-1);
  o_child3 = new Doub(-1);
  f_child1 = new Bool(false);
  f_child2 = new Bool(false);
  f_child3 = new Bool(false);
  AV = av;
  IE= new Events();
  child1 = c1;
  child2 = c2;
  child3 = c3;
}
public void storeSubComponentIEs(Events ietaxiway1_system,Events ietaxiway2_system,
                                 Events ietaxiway3_system)
{
  IEtaxiway1_system = ietaxiway1_system;
  IEtaxiway2_system = ietaxiway2_system;
  IEtaxiway3_system = ietaxiway3_system;
}

public void communicate(HashSet outputs,int which)
{
  if(which != 1)
  {
    IEtaxiway1_system.genEvents(outputs);
  }
  if(which != 2)
  {
    IEtaxiway2_system.genEvents(outputs);
  }
  if(which != 3)
  {
    IEtaxiway3_system.genEvents(outputs);
```

```
  }
}
public void initCS()
{
  child1.initCS();
  child2.initCS();
  child3.initCS();
}

public void enabled_check(HashMap e_Events,HashMap rend_TEvents,HashMap rend_GEvents,Doub other,
                          Bool final_state)
{
  //find children's enabledness
  reset();
  child1.enabled_check(child1Env_Events,child1Rend_TEvents,child1Rend_GEvents,o_child1,f_child1);
  child2.enabled_check(child2Env_Events,child2Rend_TEvents,child2Rend_GEvents,o_child2,f_child2);
  child3.enabled_check(child3Env_Events,child3Rend_TEvents,child3Rend_GEvents,o_child3,f_child3);
  //priority value of the highest priority enabled transition
  if(o_child1.value>=0)
  {
    other.value = o_child1.value;
  }
  if(o_child2.value>other.value)
  {
    other.value = o_child2.value;
  }
  if(o_child3.value>other.value)
  {
    other.value = o_child3.value;
  }
  if(f_child1.value&& f_child2.value&& f_child3.value)
  {
    System.out.println("All HTSes have reached final states - exiting"); System.exit(1);
  }
  //pass enabledness info in the children onto the parent
  rend_TEvents.putAll(child1Rend_TEvents);
  rend_GEvents.putAll(child1Rend_GEvents);
  rend_TEvents.putAll(child2Rend_TEvents);
  rend_GEvents.putAll(child2Rend_GEvents);
  rend_TEvents.putAll(child3Rend_TEvents);
  rend_GEvents.putAll(child3Rend_GEvents);
  e_Events.putAll(child1Env_Events);
  e_Events.putAll(child2Env_Events);
  e_Events.putAll(child3Env_Events);
}

public boolean execute(Integer env_Event,Integer rend_GEvent,Integer rend_TEvent,
                       Integer interrupt,HashSet outputs)
{
  boolean executed = false;
  //process ancestral constraints
  if(env_Event != null)
  {
    if(child1Env_Events.containsKey(env_Event))
    {
      child1.execute(env_Event,null,null,null,outputs);
      communicate(outputs,1);
      executed = true;
    }
```

```
    else if(child2Env_Events.containsKey(env_Event))
    {
      child2.execute(env_Event,null,null,null,outputs);
      communicate(outputs,2);
      executed = true;
    }
    else if(child3Env_Events.containsKey(env_Event))
    {
      child3.execute(env_Event,null,null,null,outputs);
      communicate(outputs,3);
      executed = true;
    }
  }
  else if(rend_GEvent != null)
  {
    if(child1Rend_GEvents.containsKey(rend_GEvent))
    {
      child1.execute(null,rend_GEvent,null,null,outputs);
      communicate(outputs,1);
      executed = true;
    }
    else if(child2Rend_GEvents.containsKey(rend_GEvent))
    {
      child2.execute(null,rend_GEvent,null,null,outputs);
      communicate(outputs,2);
      executed = true;
    }
    else if(child3Rend_GEvents.containsKey(rend_GEvent))
    {
      child3.execute(null,rend_GEvent,null,null,outputs);
      communicate(outputs,3);
      executed = true;
    }
  }
  else if(rend_TEvent != null)
  {
    if(child1Rend_TEvents.containsKey(rend_TEvent))
    {
      child1.execute(null,null,rend_TEvent,null,outputs);
      communicate(outputs,1);
      executed = true;
    }
    else if(child2Rend_TEvents.containsKey(rend_TEvent))
    {
      child2.execute(null,null,rend_TEvent,null,outputs);
      communicate(outputs,2);
      executed = true;
    }
    else if(child3Rend_TEvents.containsKey(rend_TEvent))
    {
      child3.execute(null,null,rend_TEvent,null,outputs);
      communicate(outputs,3);
      executed = true;
    }
  }
  else // round robin scheduling amongst enabled children
  {
    //quick round robin implementation (no data-structure usage)
    int loop = 0;
```

```
    for(;;)
    {
      switch(round_robin)
      {
        case 1: {
          if(1 >= round_robin && o_child1.value>=0)
          {
            child1.execute(null,null,null,null,outputs);
            communicate(outputs,1);
            loop = 2;
            executed=true;
            round_robin=2;
            break;
          }
        }
        case 2: {
          if(2 >= round_robin && o_child2.value>=0)
          {
            child2.execute(null,null,null,null,outputs);
            communicate(outputs,2);
            loop = 2;
            executed=true;
            round_robin=3;
            break;
          }
        }
        case 3: {
          if(3 >= round_robin && o_child3.value>=0)
          {
            child3.execute(null,null,null,null,outputs);
            communicate(outputs,3);
            loop = 2;
            executed=true;
            round_robin=1;
            break;
          }
        }
        default:{
          if(loop==1){loop++; break;}
          loop++;
          round_robin = 1;
        }
      }       if(loop==2){break;}
    }
  }
   reset();
   return executed;
}
private void reset()
{
  o_child1.value=-1;
  o_child2.value=-1;
  o_child3.value=-1;
  child1Rend_TEvents.clear();
  child1Rend_GEvents.clear();
  child2Rend_TEvents.clear();
  child2Rend_GEvents.clear();
  child3Rend_TEvents.clear();
  child3Rend_GEvents.clear();
```

```
    child1Env_Events.clear();
    child2Env_Events.clear();
    child3Env_Events.clear();
  }
}
```

B.5 Vars.java

```
import java.util.HashMap;
import java.util.Iterator;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

class Vars
{
  //used to resolve conflicts amongst shared variables
  private boolean parallel;
  private HashMap paraValues;

  //the variables declared in the system
  public boolean busyrw1 = false;
  public boolean busyrw2 = false;
  public boolean busytw1 = false;
  public boolean busytw2 = false;
  public boolean busytw3 = false;

  public Vars()
  {
    paraValues = new HashMap();
    parallel = false;
  }

  //This set parallel execution mode
  //in parallel execution mode, all assignments to variables are stored
  //in the paraValues hashmap, when parallel mode ends we select one
  //of the potential values to be the value for each shared variable
  public void setParallelMode(boolean val)
  {
    parallel = val;
    if(!val) //resolve shared vars (nondet resolve)
    {
      Pair p;
      p = (Pair)paraValues.get(new String("busyrw1"));
      if(p != null)
      {busyrw1 = ((Boolean)p.first).booleanValue();}
      p = (Pair)paraValues.get(new String("busyrw2"));
      if(p != null)
      {busyrw2 = ((Boolean)p.first).booleanValue();}
      p = (Pair)paraValues.get(new String("busytw1"));
      if(p != null)
      {busytw1 = ((Boolean)p.first).booleanValue();}
      p = (Pair)paraValues.get(new String("busytw2"));
      if(p != null)
      {busytw2 = ((Boolean)p.first).booleanValue();}
      p = (Pair)paraValues.get(new String("busytw3"));
      if(p != null)
```

```
      {busytw3 = ((Boolean)p.first).booleanValue();}
      paraValues.clear();
    }
  }

  //assignments made within a transitions actions need to be visible to other
  //actions associated with that transition, but not to other transitions executing
  //in simulated parallel with this transition
  public void endTrans()
  {
    for(Iterator iter = paraValues.values().iterator();iter.hasNext();)
    {
      ((Pair)iter.next()).second = Boolean.FALSE;
    }
  }

  public void setbusyrw1(boolean val)
  {
    if(!parallel)
    {
      busyrw1= val;
    }
    else //if in parallel mode, store the this potential value for future use
    {
      paraValues.put( new String("busyrw1"), new Pair(new Boolean(val),Boolean.TRUE));
    }
  }

  public boolean getbusyrw1()
  {
    if(!parallel)
    {
      return busyrw1;
    }
    else
    {
      Pair val = (Pair) paraValues.get(new String("busyrw1"));
      if(val != null && val.second == Boolean.TRUE)
      {
        return ((Boolean)val.first).booleanValue();
      }
      else
      {
        return busyrw1;
      }
    }
  }
  \\get and set methods for the rest of the variables not shown
  .......
}
```

B.6 microEnvSync.java

```
import java.util.Vector;
import java.util.HashSet;
import java.util.HashMap;
```

```
class microEnvSync_env
{
  private microIntl_intrl2 right;
  private microIntl_intrl3 left;

  private Vars AV;
  private Events IE;
  private HashSet tempOutputs = new HashSet();
  private double other_left,other_right;
  private HashMap leftRend_TEvents,leftRend_GEvents,rightRend_TEvents,rightRend_GEvents;
  private HashMap rightEnv_Events,leftEnv_Events;
  private HashSet leftInterr,rightInterr;
  private boolean left_goes = true;
  private Doub o_left,o_right;
  private Bool f_left,f_right;

  public Events IErunway1_system;
  public Events IErunway2_system;
  public Events IEtaxiway1_system;
  public Events IEtaxiway2_system;
  public Events IEtaxiway3_system;
  public microEnvSync_env(Vars av,microIntl_intrl3 l,microIntl_intrl2 r)
  {
    leftRend_GEvents = new HashMap();
    leftRend_TEvents= new HashMap();
    rightRend_GEvents = new HashMap();
    rightRend_TEvents= new HashMap();
    rightEnv_Events  = new HashMap();
    leftEnv_Events = new HashMap();
    leftInterr = new HashSet();
    rightInterr = new HashSet();
    other_right = -1;
    other_left = -1;
    o_right = new Doub(-1);
    o_left = new Doub(-1);
    f_right = new Bool(false);
    f_left = new Bool(false);
    AV = av;
    IE= new Events();
    left = l;
    right = r;
  }
  public void storeSubComponentIEs(Events ierunway1_system,Events ierunway2_system,
              Events ietaxiway1_system,Events ietaxiway2_system,Events ietaxiway3_system)
  {
    IErunway1_system = ierunway1_system;
    IErunway2_system = ierunway2_system;
    IEtaxiway1_system = ietaxiway1_system;
    IEtaxiway2_system = ietaxiway2_system;
    IEtaxiway3_system = ietaxiway3_system;
  }
  public void communicate(HashSet outputs,boolean which)
  {
    if(which)
    {
      IErunway1_system.genEvents(outputs);
      IErunway2_system.genEvents(outputs);
    }
    else
```

```
  {
    IEtaxiway1_system.genEvents(outputs);
    IEtaxiway2_system.genEvents(outputs);
    IEtaxiway3_system.genEvents(outputs);
  }
}

public void initCS()
{
  left.initCS();
  right.initCS();
}

public void enabled_check(HashMap rend_TEvents,HashMap rend_GEvents,Doub other,Bool final_state)
{
  reset();
  left.enabled_check(leftEnv_Events,leftRend_TEvents,leftRend_GEvents,o_left,f_left);
  right.enabled_check(rightEnv_Events,rightRend_TEvents,rightRend_GEvents,o_right,f_right);
  final_state.value = f_left.value && f_right.value;
  other_left = o_left.doubleValue();
  other_right = o_right.doubleValue();

  //round robin scheduling
  if(other_left>=0 && other_right>=0 && left_goes)
  {
    other_right = -1;
    left_goes = false;
  }
  else if(other_left>=0 && other_right>=0 && !left_goes)
  {
    other_left = -1;
    left_goes = true;
  }

  other.value =  other_left >= other_right?other_left:other_right;
  rend_TEvents.putAll(leftRend_TEvents);
  rend_TEvents.putAll(rightRend_TEvents);
  rend_GEvents.putAll(leftRend_GEvents);
  rend_GEvents.putAll(rightRend_GEvents);
  if(other.value<0)
  {
    if(leftEnv_Events.containsKey(IE.evententertw1) &&
      rightEnv_Events.containsKey(IE.evententertw1))
    {
      double v1 = ( (Doub)leftEnv_Events.get(IE.evententertw1) ).doubleValue();
      double v2 = ( (Doub)rightEnv_Events.get(IE.evententertw1) ).doubleValue();
      other.value = v1>v2?v1:v2;
    }
    else if(leftEnv_Events.containsKey(IE.evententertw2) &&
          rightEnv_Events.containsKey(IE.evententertw2))
    {
      double v1 = ( (Doub)leftEnv_Events.get(IE.evententertw2) ).doubleValue();
      double v2 = ( (Doub)rightEnv_Events.get(IE.evententertw2) ).doubleValue();
      other.value = v1>v2?v1:v2;
    }
    else if(leftEnv_Events.containsKey(IE.evententertw3) &&
          rightEnv_Events.containsKey(IE.evententertw3))
    {
      double v1 = ( (Doub)leftEnv_Events.get(IE.evententertw3) ).doubleValue();
```

```
      double v2 = ( (Doub)rightEnv_Events.get(IE.evententertw3) ).doubleValue();
      other.value = v1>v2?v1:v2;
    }
  }
}

public boolean execute(Integer env_Event,Integer rend_GEvent,Integer rend_TEvent,
                       Integer interrupt,HashSet outputs)
{
  boolean executed = false;
  if (rend_GEvent != null)
  {
    if(rightRend_GEvents.containsKey(rend_GEvent))
    {
      right.execute(null,rend_GEvent,null,null,outputs);
      communicate(outputs,true);
      executed = true;
    }
    else if(leftRend_GEvents.containsKey(rend_GEvent))
  {
      left.execute(null,rend_GEvent,null,null,outputs);
      communicate(outputs,false);
      executed = true;
    }
  }
  else if(rend_TEvent != null)
  {
    if(rightRend_TEvents.containsKey(rend_TEvent))
    {
      right.execute(null,null,rend_TEvent,null,outputs);
      communicate(outputs,true);
      executed = true;
    }
    else if(leftRend_TEvents.containsKey(rend_TEvent))
    {
      left.execute(null,null,rend_TEvent,null,outputs);
      communicate(outputs,false);
      executed = true;
    }
  }
  else
  {
    if(leftEnv_Events.containsKey(IE.evententertw1) &&
      rightEnv_Events.containsKey(IE.evententertw1))
    {
      executed = true;
      left.execute(IE.evententertw1,null,null,null,outputs);
      right.execute(IE.evententertw1,null,null,null,tempOutputs);
      communicate(outputs,false);
      communicate(tempOutputs,true);
      outputs.addAll(tempOutputs);
      tempOutputs.clear();
    }
    else if(leftEnv_Events.containsKey(IE.evententertw2) &&
           rightEnv_Events.containsKey(IE.evententertw2))
    {
      executed = true;
      left.execute(IE.evententertw2,null,null,null,outputs);
      right.execute(IE.evententertw2,null,null,null,tempOutputs);
```

```
        communicate(outputs,false);
        communicate(tempOutputs,true);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(leftEnv_Events.containsKey(IE.evententertw3) &&
             rightEnv_Events.containsKey(IE.evententertw3))
      {
        executed = true;
        left.execute(IE.evententertw3,null,null,null,outputs);
        right.execute(IE.evententertw3,null,null,null,tempOutputs);
        communicate(outputs,false);
        communicate(tempOutputs,true);
        outputs.addAll(tempOutputs);
        tempOutputs.clear();
      }
      else if(other_left>=0)
      {
        executed = true;
        left.execute(null,null,null,null,outputs);
        communicate(outputs,false);
      }
      else if(other_right>=0)
      {
        executed = true;
        right.execute(null,null,null,null,outputs);
        communicate(outputs,true);
      }
    }
     reset();
     return executed;
  }
  private void reset()
  {
    o_right.value=-1;
    o_left.value=-1;
    leftRend_TEvents.clear();
    leftRend_GEvents.clear();
    rightRend_TEvents.clear();
    rightRend_GEvents.clear();
    rightEnv_Events.clear();
    leftEnv_Events.clear();
  }
}
```

Bibliography

[1] ITU-T. Recommendation Z.100. Specification Description Language (SDL). Technical report, International Telecommunications Union - Standardization sector, 1999.

[2] Statistical summary of commercial jet airplane accidents world wide operations 1959-2001. Technical report, Boeing Comercial Airplane group, Airplane Safety Engineering, 2002.

[3] J. Ali and J. Tanaka. Converting statecharts into java code. In *5th International Conference on Integrated Design and Process Technology (IDPT'99)*, page See p. 42, 1999.

[4] ApeSoft. Smartstate. http://www.smartstatestudio.com, 2005.

[5] C. W. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (Palo Alto, CA)*, volume 1166, pages 187–201. Springer, 1996.

[6] G. W. Bond and H. Goguen. ECLIPSEcharts: a visual bridge from design to implementation. 2001.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 117–126, New York, NY, USA, 1983. ACM Press.

[8] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] R. J. Erich Gamma, Richard Helm and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Welsey, 1995.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The 3rd Edition*. Addison-Wesley Publishing Co., New York, NY, USA, 2005.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] D. Harel. On the formal semantics of statecharts. *Logic in Computer Science*, pages 54–64, 1987.

[13] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[14] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer,2(4)*, 2004.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Methodol.*, 5(3):231–261, 1996.

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

[17] I-Logix. Rhapsody. http://www.ilogix.com/rhapsody/rhapsody.cfm, 2005.

[18] IBM. Rational rose realtime. http://www-130.ibm.com/developerworks/rational, 2005.

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

[20] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.*, 20(9):684–707, 1994.

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

[22] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.

[23] J. Niu. *Metro: a semantics-based approach for mapping specification notations to analysis tools*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, ON, Canada, 2005.

[24] J. Niu, J. M. Atlee, and N. A. Day. Template Semantics for Model-Based Notations. *IEEE Transactions on Software Engineering*, 29(10):866–882, October 2003.

[25] Object-Management-Group. The meta object facilitly (MOF) specification. http://www.omg.org/docs/formal/02-04-03.pdf, 2005.

[26] Object-Management-Group. The model driven architecture resources page. http://www.omg.org/mda, 2005.

[27] Object-Management-Group. Revised submission for MOF 2.0 Query/View/-Transformation rfp. http://www.omg.org/docs/ad/05-03-02.pdf, 2005.

[28] Object-Management-Group. The unified modeling language - version 1.5. http://www.omg.org/technology/documents/formal/uml.htm, 2005.

[29] M. Samek. *Practical Statecharts in C/C++*. CMP Books, Lawrence, Kansas, 2002.

[30] E. Technologies. Scade. http://www.esterel-technologies.com/products/scade-suite/overview.html, 2005.

[31] A. Wasowski. *Code Generation and Model Driven Development for Constrained Embedded Software*. PhD thesis, IT University of Copenhagen, 2005.

[32] WindRiver. Betterstate. http://www.windriver.com/portal/server.pt, 2005.

[33] T. Yavuz-Kahveci and T. Bultan. Specification, verification, and synthesis of concurrency control components. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 169–179, New York, NY, USA, 2002. ACM Press.