DEVS/UML – A FRAMEWORK FOR SIMULATABLE UML MODELS

by

Joseph Mooney

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

May 2008

DEVS/UML – A FRAMEWORK FOR

SIMULATABLE UML MODELS

by

Joseph Mooney

has been approved

March 2008

Graduate Supervisory Committee:

Hessam Sarjoughian, Chair
Stephen Yau
Eric Christensen

ACCEPTED BY THE GRADUATE COLLEGE

ABSTRACT

The Discrete Event System Specification (DEVS) formalism excels at modeling complex discrete event systems. A framework capable of simulating a DEVS model is presented via Unified Modeling Language (UML) state machines. A set of rules is enumerated for the creation of UML models. Adherence to these rules results in models that are both DEVS and UML compliant. Resultant UML models are executable within DEVS simulation frameworks such as DEVSJAVA. Such an approach to modeling in UML represents a significant improvement over alternative approaches since it enables earlier simulation and verification of a design. The thesis asserts that simulation fulfills an important role within the architecture of a system and can be readily employed within UML models. The specifics of simulation can be naturally expressed in UML models, and simulatable models are a natural precursor to the more detailed models developed during design and implementation.

ACKNOWLEDGEMENTS

I thank my wife, Sandra, and daughter, Saoirse, for their patience, love and support during the past five years. A most sincere and profound thanks to my committee chair, Dr. Hessam Sarjoughian, Department of Computer Science and Engineering, Arizona State University without whom I may never have completed my studies, *go raibh maith agat* - a thousand thanks. Finally, I would like to thank both Dr. Eric Christensen, Chief Software Engineer (CND), General Dynamics, and Dr. Stephen Yau, Department of Computer Science and Engineering, Arizona State University, for their assistance with this thesis.

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

Figure Page

GLOSSARY

CORBA An OMG standard aimed at allowing access to common objects and services by software written in different programming languages and executing on different operating platforms.

DEVS Discrete Event System Specification is a systems-theoretic mathematical formalism that enables modeling systems using a communication protocol with well-defined semantics yielding system specifications that readily lend themselves to simulation, validation and verification.

DEVJAVA A Java implementation of a DEVS modeling and simulation environment.

Jini Developed by Sun Microsystems. This is a Java-based architectural framework to aid in the development of distributed software systems suitable from small devices to enterprise level applications.

JavaSpaces Developed by Sun Microsystems as part of Jini. This is a mechanism to store and retrieve objects in a distributed environment. The objects can be retrieved from JavaSpaces and their methods invoked.

MDA Model Driven Architecture® (MDA®) is another OMG initiative that seeks to separate business from application logic through the use of platform independent models that can be transformed via a platform definition model into a platform specific model where platforms consist of various middleware architectures such as CORBA and .NET.

OMG The Object Management Groups is a standards consortium responsible for the UML, CORBA, MDA, and other standards that enable the design, execution, and maintenance of software and other processes.

RTC Run-to-completion ensures that each event is processed in an atomic uninterruptible manner. Events may not be half processed due to the arrival of a higher priority event.

UML The Unified Modeling Language is a general-purpose modeling language standardized by the OMG intended to provide a complete modeling framework for all steps in the development, deployment and maintenance of a system.

C H A P T E R 1

INTRODUCTION

Motivation

This thesis presents the specification of various DEVS (Zeigler et al. 2000) models in terms of UML (OMG 2007) state machines. We will show that using UML state machines according to the principles of DEVS yields an executable modeling sub-language suitable for a DEVS framework. Modeling via a combination of DEVS and UML (hereafter DEVS/UML) provides a structured approach for the creation of a UML model that can be simulated under an executable DEVS framework. DEVS makes no pretence at yielding a model that can be simply handed off to developers for implementation. Much design work is still necessary once these models are in place, but also much has been achieved and potentially verified at an early design stage which would otherwise not have been possible using UML alone. Furthermore, the simulatable models have applicability potential throughout the software development lifecycle in areas orthogonal to implementation such as validation and verification. Additionally, DEVS/UML allows software architects to code in the abstract, meaning that within this framework a designer can experiment with code (e.g. Java) without being lulled into detailed design or implementation issues. This is necessary since it is difficult to get to the meat of a problem through the use of purely abstract modeling constructs without the type of feedback that an executable model provides. From a pragmatic perspective a malleable executable model is well served by having the expressive power of a modern high level language allowing details that are difficult to express in purely UML terms.

Contributions

This thesis introduces a novel approach for simulating software systems. For far too long within the software architecture community, simulation has been an underutilized and misunderstood endeavor. This thesis centers on the ability to create executable UML models that represent a simulation of a proposed system such that the UML practitioner requires only a basic understanding of simulation. In adhering to the modeling techniques developed for this thesis, a new approach is identified that allows the modeler to start at a *high level of abstraction* and iteratively develop models of greater and greater detail *all the while allowing the modeler to execute these models*. Such an approach to software modeling in UML should significantly expand the role of simulation beyond that which it occupies today.

Toward the main contribution, the following outcomes were achieved:

- Identification of the challenges in developing models suitable for simulation in UML, particularly in regards to synchronization and delivery of event signals across multiple components.
- The specification of a set of guidelines to be followed when modeling in UML such that those models may be executable in a DEVS framework such as DEVSJAVA.
- The development of a prototype simulator to explore the issues raised by this thesis and help demonstrate the proposed approach.
- A mapping of concepts employed in DEVSJAVA to DEVS/UML.
- Alternative architectural possibilities for the construction of simulator engines.

C H A P T E R 2

BACKGROUND

Why Simulate?

El Sheik et al. (2008) present thirteen incentives for employing simulation summarized here:

- Inexpensive experimentation with many different scenarios.
- Ability to expand and compress time.
- Ability to easily replay events to discover why events occurred.
- Ability to explore different possibilities relatively risk-free.
- Potential for diagnosing problems.
- Identification of constraints and ability to predict obstacles.
- Informed decision making as opposed to hunches.
- Animation of a simulated system can reveal hard to find defects.
- Visualization of the “plan”.
- Builds consensus among the stakeholders.
- Risk reduction for changes to the system – ability to test of what-if scenarios.
- Training opportunities available via simulation.
- Simulation saves time and effort.

Certain objections to simulation of software have been raised (Douglass 2005) including the cost of testing twice, once in the simulated version and then again in the real system, and also the belief that with a modern iterative development approach simulation is no longer necessary since the real system can be evolved incrementally thereby nullifying many of the incentives listed above. This thesis recommends creating a simulation model using the UML wherein most of the aspects of the model particular to simulation are modeled separately wherever possible. Since the UML becomes the common modeling language for both the simulation model and the real software system, the perception of simulation as a disjoint and unnecessary exercise should be reduced. Furthermore, the perceived value of the simulation models will likely be enhanced since these models are the easiest to create and execute due to their relative simplicity. In terms of iteratively developing models, a modeler can begin with a simulatable model and as details are added, evolve toward a model closer to a traditional prototype and then evolve that model on through to production. In so doing we can diminish the resistance to employing simulation during the conceptualization and development of a system. It should be noted that it is possible to evolve a simulatable model into greater and greater levels of precision through decomposition of the model whilst still remaining a simulatable model. Once we digress into the prototype phase, the models become qualitatively different in nature and this evolution should not be seen as a simple one-to-one mapping. Beyond the initial architectural phases, the simulatable models continue to serve as important tools to educate, stimulate and instruct. Whenever changes are under consideration or various what-if scenarios need examination additional experimentation using simulatable models may be employed.

Why DEVS?

According to Zeigler et al (2000, 10) "DEVS is the unique form of representation that underlies any system with discrete event behavior". UML is inherently a discrete modeling language (Douglass 2004). It is therefore natural to consider what forms a DEVS-compliant UML model may take. Models expressed using the Discrete Event System Specification (DEVS) represent a class of systems theoretic models that permit parallel event-based behavior to be expressed concisely and in a manner that lend themselves to formal verification (Zeigler 2000). Although many different simulation formalisms have been advanced over the years, the DEVS formalism has emerged as the preferred formalism due to the fact that other formalisms have been proven to have an equivalent DEVS representation (Zeigler 2000, Choi et al 2003). In particular, a differential equation system specification (DESS) can be simulated by a discrete time system specification (DTSS) through the selection of a sufficiently small constant time interval. A DTSS model, in turn, can be simulated by a DEVS model by constraining the time advance to a constant time. As such, simulations based on DEVS are more general in nature than other approaches such as continuous simulation (Kofman 2003). In DEVS, a system is represented by two types of models: atomic and coupled models. Atomic models are leaf nodes in a hierarchy where coupled models are the non-leaf nodes.

Atomic Models

An atomic model may be represented by the structure:

$$M = (X, Y, S, \delta_{ext}, \delta_{int}, \delta_{conf}, \lambda, ta)$$

Where:

- X is the set of input values.
- Y is the set of output values.
- S is the set of states.
- δ_{ext} : $Q \times X^b \rightarrow S$ is the external transition function, where $Q = \{(s,e) \mid s \in S, 0 \leq e \leq ta(s)\}$ is the total state set and e is the elapsed time since the last transition.
- δ_{int}: $S \rightarrow S$ is the internal transition function.
- δ_{conf}: $Q \times X^b \rightarrow S$ is the confluent transition function

- $\lambda : S \rightarrow Y$ is the output function.
- $ta : S \rightarrow R^{+}_{0,\infty}$ is the set of positive real numbers between 0 and infinity.

This thesis proposes that each atomic model may be expressed by a UML state chart. This in itself is not a novel approach as it has been the preferred representation in UML for a DEVS atomic model in most other research in this area (Schulz et al. 2000, Huang and Sarjoughian 2004, Zinoviev 2005, Risco-Martin et al. 2007). However, this thesis tackles the difficult issue of appropriately accounting for time in UML models through a system-wide protocol of events without relying on the timing mechanisms inherent in the UML. This avoidance of UML time constructs is an important distinction between this thesis and the research of others in this area, notably Zinoviev (2005).

Coupled Models

DEVS is closed under coupling, meaning that when two or more DEVS models are coupled together the coupled model is itself a DEVS atomic model. Coupling is a unidirectional association between two models where the output (events) from one model serve as input (events) to the other model. DEVS employs the concept of input and output ports to represent the connection points between the models. Each port has a name. For each model all output ports are distinctly named and likewise all input ports are distinctly named. Further, a coupled model itself may have unidirectional associations called *external input coupling*, from its own input ports to the input ports of the models it contains, and the coupled model has unidirectional associations called *external output coupling*, from the output ports of the models it contains to its own output ports. In this manner DEVS models may have arbitrary levels of nesting. This thesis maps the port names to UML event type names – again a logical choice employed by others. Unique to this thesis is a specification of a protocol, defined via UML state machines, of event signals among models, such that those models may be simulated using an executable DEVS framework such as DEVSJAVA.

It is important to recognize DEVS models solve a general class of problems, but are by no means suitable for all types of problems. Nonetheless, the approach outlined in this thesis is general purpose with a wide breath of applications for which the UML and state charts in particular are themselves employed.

UML 2.0 Components

Many important changes were introduced with UML 2. Perhaps most notably, the notion of a component has been revised from an artifact such as a file or executable to a logical construct specified during design. Within a system, components now represent logical units that are autonomous and self-encapsulated. UML 2 introduced the component diagram to organize and layout components. The component diagram is significant in that it allows for a hierarchical decomposition of a system wherein components at lower levels may be substituted with alternative components as desired. In UML 1.*x*, the ability to model a system in a component-oriented hierarchical fashion was difficult. In the UML 2 component diagram, components may be decomposed into other sub-components and wired together via connections and ports. This approach is very similar to the coupled and atomic models in DEVS though there are some important distinctions. In simple terms a coupled model may be perceived as a UML 2.0 component diagram with an additional set of rules restricting how the component diagram may be wired together.

Coding in the Abstract

Coding in a modeling and simulation environment is mostly orthogonal to traditional coding. Hence, it should not involve implementation decisions, nor would it be tempting to use this code during implementation. A central objection to prototypical solutions is that they tend to morph into the real product and as such represent a dangerous temptation. Coding in the abstract is not a matter of writing a collection of stubs and interfaces. Java (in the case of DEVSJAVA) becomes a glue language for architectural constructs where some other abstract specification language may have been used in the past. Java has the benefit of being executable, widely used and understood, and it also has a rich syntax to allow the architect the flexibility to express concepts that may otherwise be difficult to capture. These models are not blueprints for implementation but rather they can be seen as vehicles through which we can act out many different scenarios and get a sense of whether our vision is coherent and consistent. Producing an executable DEVS/UML model should be faster and easier than traditional methods because the executable framework is already in place with an abundance of supporting libraries and constructs. Above all DEVS/UML represents a pragmatic approach to rapidly producing an architectural model of a system that involves a high degree of dynamic behavior. Stephen Mellor (Mellor and Balcer 2002) objects to using Java or a similar programming language since modelers are likely to develop specifications that compromise the intended level of abstraction with non-domain specific constructs such as pointers and arrays. Whilst these objections are justified, a pragmatic approach suggests that Java employed in the specification of a DEVS model is not necessarily a poor choice so long as the modeler exercises good choices with respect to its application. In return for employing this pragmatic approach, a model compiler and execution environment is readily available.

Modeling in DEVS

DEVS is a simple, elegant, yet very powerful and scaleable formalism which encourages both a compositional and decompositional approach to creating models for potential systems. Large models of complex systems can be expressed by focusing on atomic components of limited complexity and then assembling these atomic components into systems with emergent yet predictable complexity. Alternatively, components can be viewed as black boxes for which there is expected behavior (expected outputs for given inputs) and later these black boxes can be decomposed into a series of sub-components. DEVS is appealing since it operates at a high level of abstraction yet can yield critical information during an architectural phase that might otherwise not come to light until much later. Further, it has been shown that DEVS models are particularly suited to the expression of many design patterns and allow an architect to employ patterns usefully at an architectural and modeling stage (Ferayorni and Sarjoughian 2007). There is a Classic DEVS and updated Parallel DEVS (Chow 1996) formalism. In this thesis, references to DEVS should be considered references to Parallel DEVS. One way to think of DEVS is as a protocol that specifies how components may be grouped and connected together and how and when they may communicate with one another. DEVS constitutes a minimal set of rules and structures necessary to formulate a model of virtually unlimited sophistication. This thesis asserts that observation of this set of minimal rules during modeling in UML is a powerful enabler of simulation and validation for any modeler in almost any domain. A limitation of DEVS is that it is generally unsuitable for specifying a software application at a level that can implemented as an end product.

UML Drawbacks

The UML is a profound and complex modeling language and in many respects it remains profoundly complex. At a certain level it attempts to be all things to all people, a grandiose, noble attempt to achieve a holy grail of raising the level of software (and systems) specification to an altogether higher plateau. Advocates of UML would argue that it is as simple and complex as it needs to be. As stated earlier, UML 2.*x* includes important improvements that can reduce the growth in complexity of models as a system increases in size. Enhancements in areas such as component diagrams are a major improvement over earlier releases of the UML. Indeed, several important the concepts such as ports and hierarchical composition of models present in DEVS were absent in UML 1.*x* and then adopted by the UML in 2.0. However, despite these important introductions into UML 2.0, the handling of time, especially as required for simulation, remains a significant shortcoming. Indeed, the UML Profile for Schedulability, Performance and Time Specification (OMG 2005) seeks to address this shortcoming and was considered as a basis for this thesis but was ultimately rejected since the profile introduced more complexity than required to achieve a UML representation of a DEVS model.

DEVS/UML

DEVS/UML is a proposed architectural framework in which a modeler employs UML according to a set of rules and is thus enabled to realize an executable model running under a DEVS framework. This allows an architect to generate functioning models of a system at a very early state in its development and thus assist in formalizing and testing a conceptualization. DEVS/UML seeks to employ UML is a manner in keeping with the strictly minimalist approach employed by DEVS. That is to say, elements of the UML are utilized on an as-needed basis and in a progressive and logical manner. In DEVS/UML, we model systems in primarily a reactive, event-driven fashion, although it is possible for models to be autonomous or non-reactive. Non-reactive, non-autonomous behavior is subjugated to being specified within an atomic model.

An objective of DEVS/UML is to help guide the architect/modeler in the creation of models that can be verified. These executable models can be realized, tuned, and corrected at a very early stage in the system development. There are best practices which should be employed to help ensure a successful modeling experience, but it is nonetheless possible that unnecessarily complex models are developed which hinder a successful outcome rather than contribute to an elegant, comprehensible and accurate model for a desired solution. An example of such over-engineering is the area of state machine development. DEVS/UML focuses on the provision of state machines for the specification of the behavior of basic (or atomic) models. The designer must be careful not to get carried away with trying to model every possible state and condition and end up with a model that is not accessible to others who wish to understand the behavior of the model. Multiple incarnations of the model may be necessary to delve deeper into the details. Every DEVS model should encapsulate behavioral details. Externally, each model appears as a black box in its environment with a well defined interface in terms of what messages can be received and what messages may be produced. That said, a DEVS model deliberately does not involve very much by way of implementation detail in terms of its specification. DEVS/UML is intended to identify junction points between DEVS and UML whereat UML artifacts are either produced or consumed. For example, DEVS/UML could generate sequence diagrams based on the execution of an experiment. Alternatively, one can validate that an experiment passed if it is in accordance with an existing sequence diagram.

When modeling, one may elect different levels of abstraction for different models. For example, one may compose an atomic model that models the behavior of a car and consider it as a single entity with a limited number of states, or one may decompose that model into a set of constituent parts. If both models share the same perspective then the set of states and transitions at the level of the car should be the same for each use case scenario. Where an atomic model is decomposed into a coupled model, both models should be polymorphic in nature: substituting one for the other should be possible and the semantics should not be compromised. Ideally, the behavior should be identical but that may not be possible if certain compromises were made during the atomic model creation for the sake of simplicity - in which case each state in the atomic model should represent a subset of states in the coupled model. We can also model from different perspectives which may involve a different decomposition and different set of states of interest. Reflecting on the ultimate goals of the system, all of these models must be reconciled into a holistic solution that captures the full essence of the system in question. Much of this reconciliation must take place further into the development life cycle but this modeling remains a valuable tool. Indeed, it is not as if all DEVS/UML models need be generated exclusively at the initial stages of design and conceptualization – DEVS/UML has practical application through all iterations of system development. For example, these models can be instructive during both implementation and test phases during which further elaboration of these models is desirable to answer questions pertinent during those phases.

Reactive Behavior

Within DEVS the specification of reactive behavior for a basic model is the form of mathematical notation or pseudo-code – there is no formal action language syntax defined as part of DEVS. There are no limitations as to how a model reacts to a given input other than a requirement to only act upon the defined state variables of the model, and all interaction with other components is through the output function. No requirement exists to specify the reactive behavior of the model in terms of a particular state machine formalism etc. (although it has been shown to be equivalent to such formalisms). Within DEVS/UML, it is required that this reactive behavior is captured via a UML state machine obeying a certain set of rules.

Modeling Simultaneous Events

Nonetheless, there is the thorny issue of handling multiple simultaneous events. If we perceive the inputs to an atomic model as events, then we are confronted with the restriction that multiple simultaneous events cannot be expressed in a UML state machine unless they occur in orthogonal regions (OMG 2007). This restriction may appear reasonable where events are processed in close to zero time, but from a DEVS perspective it represents a fundamental hurdle in the UML specification for reactive behavior. DEVS supports multiple events being processed at a given point in time. DEVS also supports time events (e.g. after 10 seconds) occurring simultaneously with other events. Practically speaking, whether two events are truly simultaneous is debatable, but from a modeling perspective it is nonetheless possible, reasonable, and practical to say two events happen say 10 seconds from now.

We then are left with the challenge of how we react to such simultaneous events. Since simultaneous events are only partially supported in UML2.0, DEVS/UML likewise has limited scope for the specification of such events. From one perspective, this is not an onerous limitation since such events present considerable challenges during implementation. That stated, from a simulation perspective when scheduling events, it is often necessary to specify that events occur simultaneously. The corresponding state machines must take this into account. For example, if events e_1 and e_2 are simultaneous, in DEVS, we can model handling these events and ignoring an event e_3 that occurs during the processing of e_1 and e_2. In UML2.0 this is not possible but one may argue that the likelihood of accepting multiple simultaneous events and then rejecting subsequent events does not have many practical applications. Further, simultaneity is only a function of the accuracy of the clock; if the clock were at a much finer grain, simultaneous events may not be simultaneous at all. This raises the possibility of considering events that occur within a given window of time as simultaneous. One beauty of DEVS is that the simulated time can be speeded up or slowed down to accommodate whatever level of granularity we choose to model.

Run-to-Completion

UML presumes that run-to-completion (RTC) event handling encompasses the entire state machine and not just any steps taken in one orthogonal region of the state machine (OMG 2007, 560). It should be noted that embracing the run-to-completion model of execution as specified within UML2.0 state machines does not prohibit DEVS/UML from modeling real-time processing systems. On the contrary, these DEVS/UML models are ideally suited to many such applications. DEVS/UML encourages a very limited utilization of RTC whereby most of the processing remains interruptible by higher priority activities. What is confined to RTC is simply the entry action into such a processing state. Most of the time is spent in a sleep state which is intended to simulate the processing that a real implementation would be performing in that state. Since external events are still recognized in this state, the sleep state is interruptible unless the modeler chooses to ignore all such external events.

Modeling Time

In DEVS, the outputs from (or events generated by) an atomic model are generated in the output function which is invoked *immediately prior* to an internal transition function and *never* in direct response to an external event. This is the primary contractual obligation of a designer creating UML2.0 state charts compatible with DEVS/UML. Whilst this may appear counterintuitive at first, it is natural from a simulation perspective – outputs only occur after some (perhaps zero) amount of processing time. Maintaining this restriction keeps the model specification consistent and reduces complexity for large systems. In DEVS/UML, the outputs are event signals generated as part of a transition triggered on the internal transition event, *evInternalTx,* which is generated when the timer corresponding to an *evSigma* event elapses. In UML, time values can be absolute or relative. The representation is a string without formal semantics other than a representation of time. So a *TimeExpression* (OMG 2007, 451) of value "1" may mean one second, nanosecond, or day. It is preferable to say "1 second" or some such expression that will not be ambiguous. In DEVS/UML, time is usually modeled via the generation of an *evSigma* event which results in a timer being created which upon expiration will trigger an internal transition *evInternalTx* event. This is analogous to the UML *after* event but *after* is not suitable for use in state machines in DEVS/UML since all events must be globally coordinated due to timing considerations.

Design Patterns

DEVS/UML operates at a similar level of abstraction to a design pattern whereby the implementation details are not relevant to model creation. Design patterns can be easily employed and reused within the framework without considerable customization for given applications. In (Ferayorni and Sarjoughian 2007) Composite, Façade, and Observer patterns are applied to a specific domain and used to develop domain-specific simulation models that enable various design possibilities to be demonstrated both early in the conceptualization of a system and also at additional critical phases during the lifecycle of the system development. The beauty of employing design patterns is that they tend to be intuitively appealing to a wide range of stakeholders and lower the complexity threshold when communicating the intent or design of a system. Also, with design patterns significant alternative solutions tend to exist at the level of the design pattern, and in this form these alternative solutions may be more readily communicated to stakeholders. This is important since simulation works hand-in-glove with design patterns allowing not only for a textual or graphical representation of a problem/solution but also through simulation stakeholders can visually witness an animation of the alternatives and the relative merits of the approaches can be objectively quantified through experimentation. The time and cost of presenting this information to stakeholders through demonstration and experimentation is already principally covered since DEVSJAVA and other such DEVS simulation environments are already mature software platforms. What remains is the model specification for the alternatives under consideration which is a vastly reduced task since the model specification for DEVS are stripped of all but the most essential detail.

Another area of interest pertinent to this thesis involves executable architecture description languages such as Rapide (PAVG). These languages offer support for high level architecture analysis in terms of components, but these languages tend to have limited ability to support modern software concepts such as design pattern and object-oriented concepts. These limitations, together with an inability to model autonomous component behavior, mean that these executable architecture description languages are not well suited for creating simulatable models of software systems (Ferayorni and Sarjoughian 2007).

CHAPTER 3

MAPPING DEVS MODELS TO UML (2.*x*) STATE MACHINES

Strategy

As a means of introducing DEVS/UML, we shall first consider basic DEVS models and how they may be represented by a UML2.0 state machine. The first models are taken from (ACIMS). Starting from simple models, we evolve to more complex models, until we ultimately arrive at a more complete specification of a simulatable model. The state machines presented initially are not the actual DEVS/UML specifications for these models, nor are they necessarily valid UML state machines. Instead, they serve as a means by which we explore the issues involved, and we then elaborate upon these until we finally start producing UML state machines capable of supporting a DEVS specification.

Passive Atomic Model

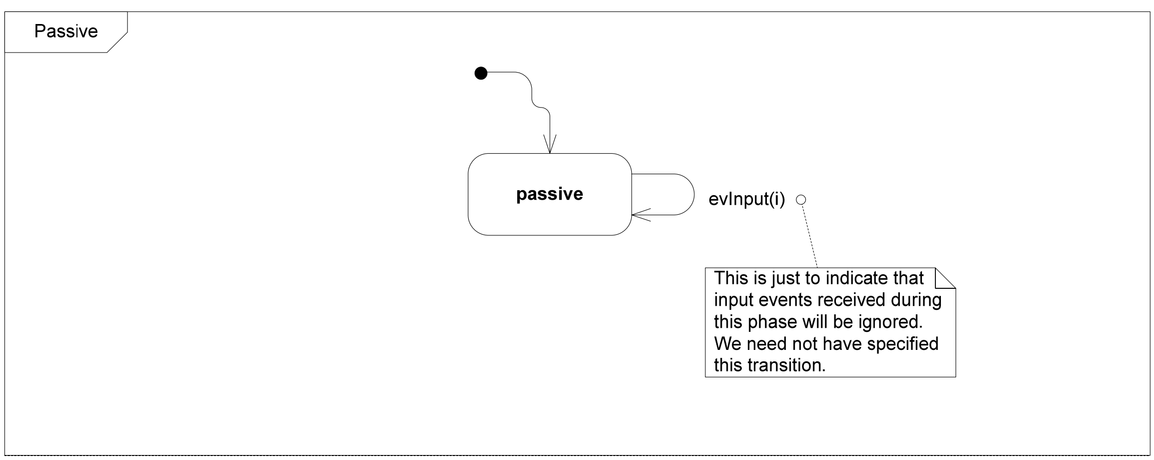

Figure 1 Passive Model (1)

The passive atomic model does nothing. In itself, it is not very interesting, but in comparing it with the corresponding DEVS model, we observe a mapping decision. In DEVS, every component has at least one input port which accepts inputs. We modeled this in the first diagram with a self-transition of evInput().

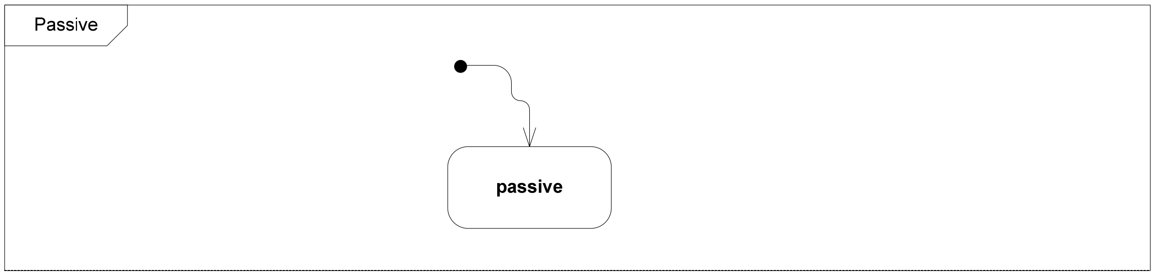

Figure 2 Passive Model (2)

Since such inputs are ignored, we have eliminated this transition in the second state machine diagram. In DEVS, the inputs are formally specified using mathematical notation. Within UML, we can formalize this specification further along with the possibility of employing protocol state machines to specify the sequence of different inputs if necessary. For the moment we will only introduce concepts such as protocol state machines as needed.

Generator Atomic Model

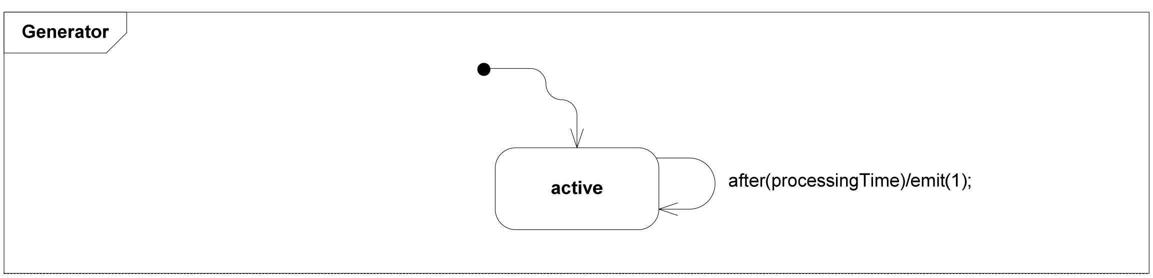

Figure 3 Generator Atomic Model

The generator atomic model simply generates an output every one second or such period. UML supports the concept of relative time events using the *after* operator. We "emit" a "1" after the processing time. This should be a signal, and signals must be sent to an object – we will ignore this objection for the moment [*objection*#1].

Storage Atomic Model

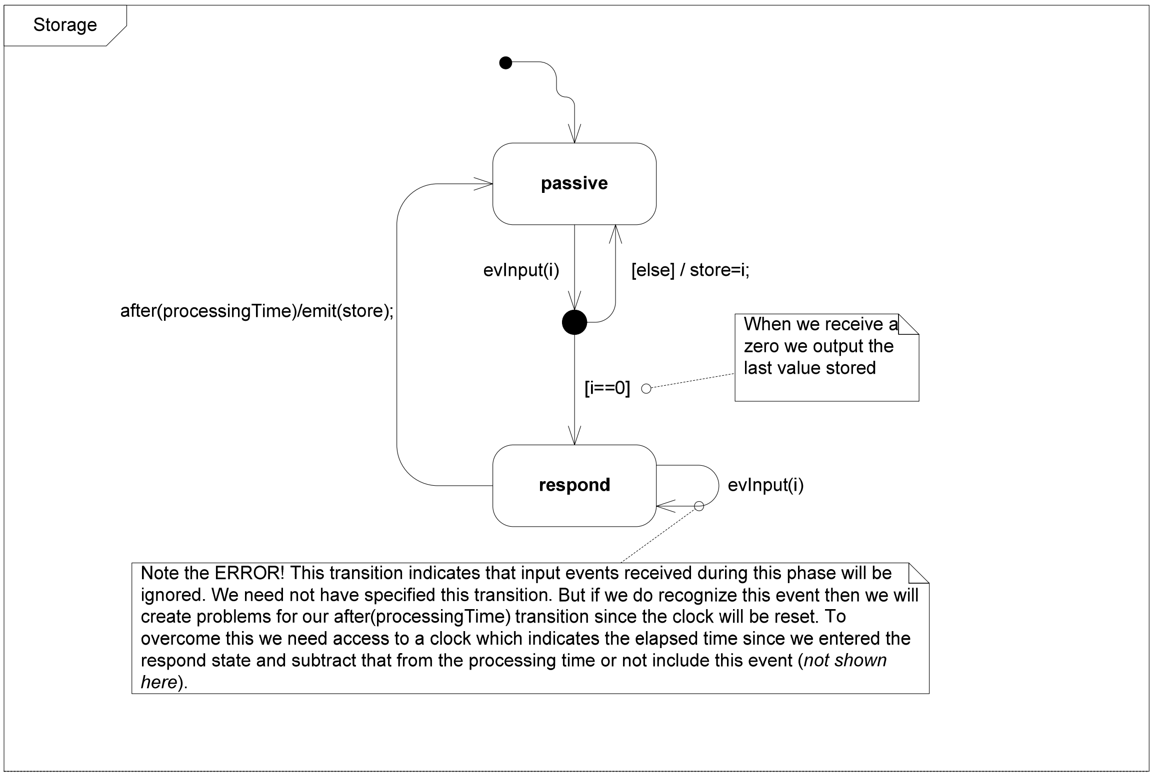

Figure 4 Storage Atomic Model

The storage atomic model stores a non-zero input and outputs this input whenever it receives a zero as input. In DEVS, it is possible for the *after* event and an input event to occur simultaneously [*objection* #2]. We will defer addressing this issue until later. Also accepting an input event in the *respond* state resets the after timer and should not therefore be specified. We could have used an internal UML transition but this too has drawbacks. As noted, there are many subtle variations in the semantics of different state machine specifications and in implementations that can lead to radically divergent behavior for the same set of events. It is therefore desirable to maintain a relatively simple specification in terms of the constructs necessary to represent DEVS in UML notation (Crane and Dingel 2005).

Binary Counter Atomic Model

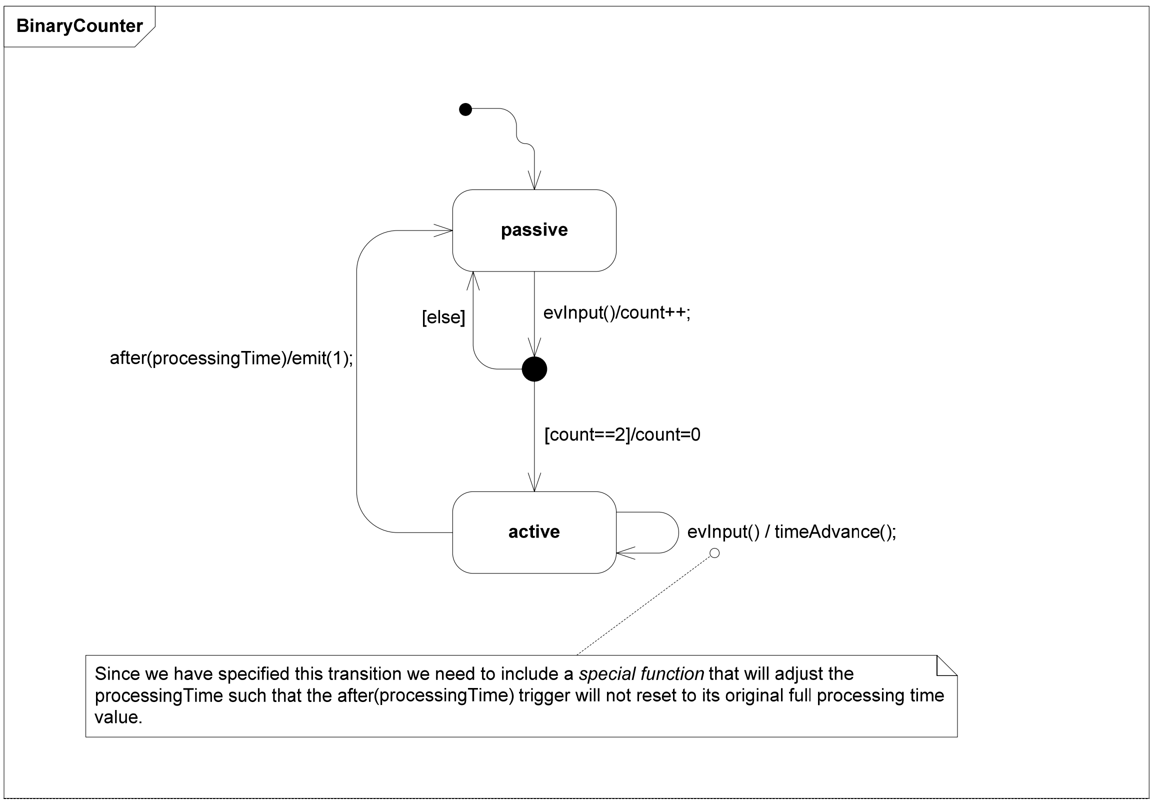

Figure 5 Binary Counter Atomic Model

The binary counter emits a 'one' for every two inputs it receives. As defined, here it also ignores any inputs it receives while in the active state which takes 1 second to complete. To compensate for inputs arriving that are ignored and which reset the processing time back to one second, we introduce a *timeAdvance* function that reduces the processing time by the amount of time elapsed. This presumes that the state machine has access to a global clock which is not defined in UML – we shall ignore this limitation for the moment [*objection* #3].

N-Counter Atomic Model

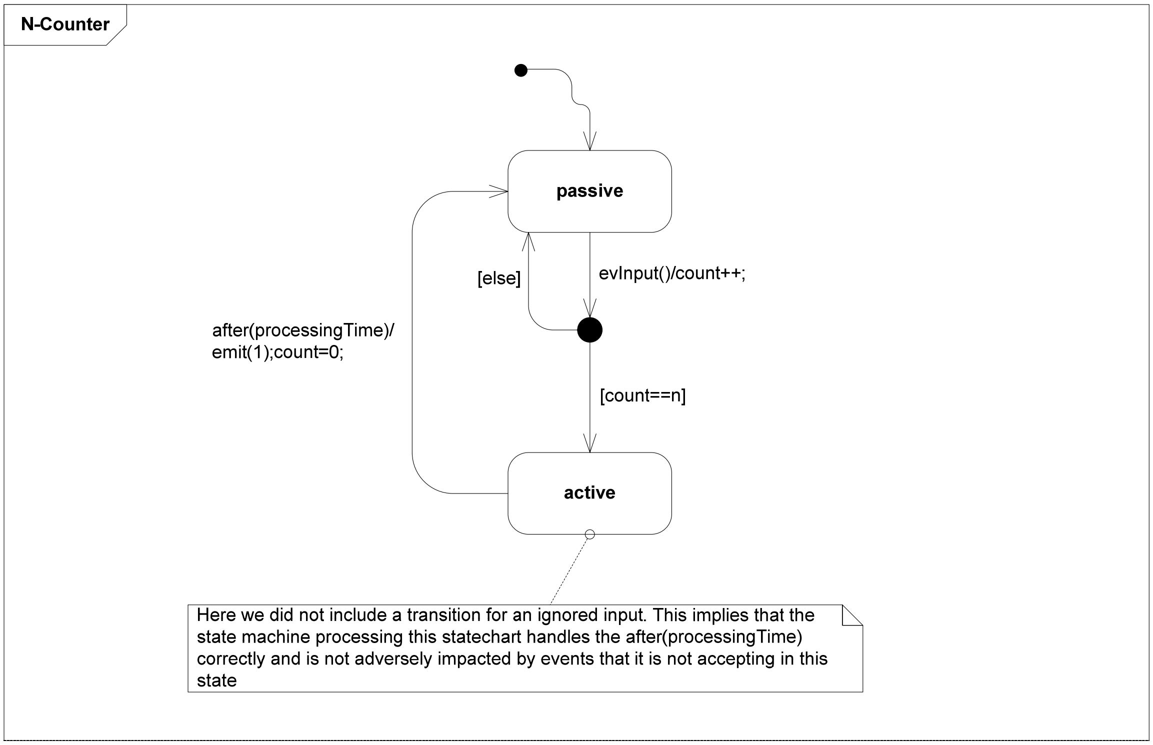

Figure 6 N-Counter Atomic Model

The n counter emits a 'one' for every *n* inputs it receives. As defined here it also ignores any input events it receives while in the process of outputting which takes 1 second to complete. Therefore, we do not need the *timeAdvance* function presented in the binary counter atomic model.

Simple Processor without Queue Atomic Model

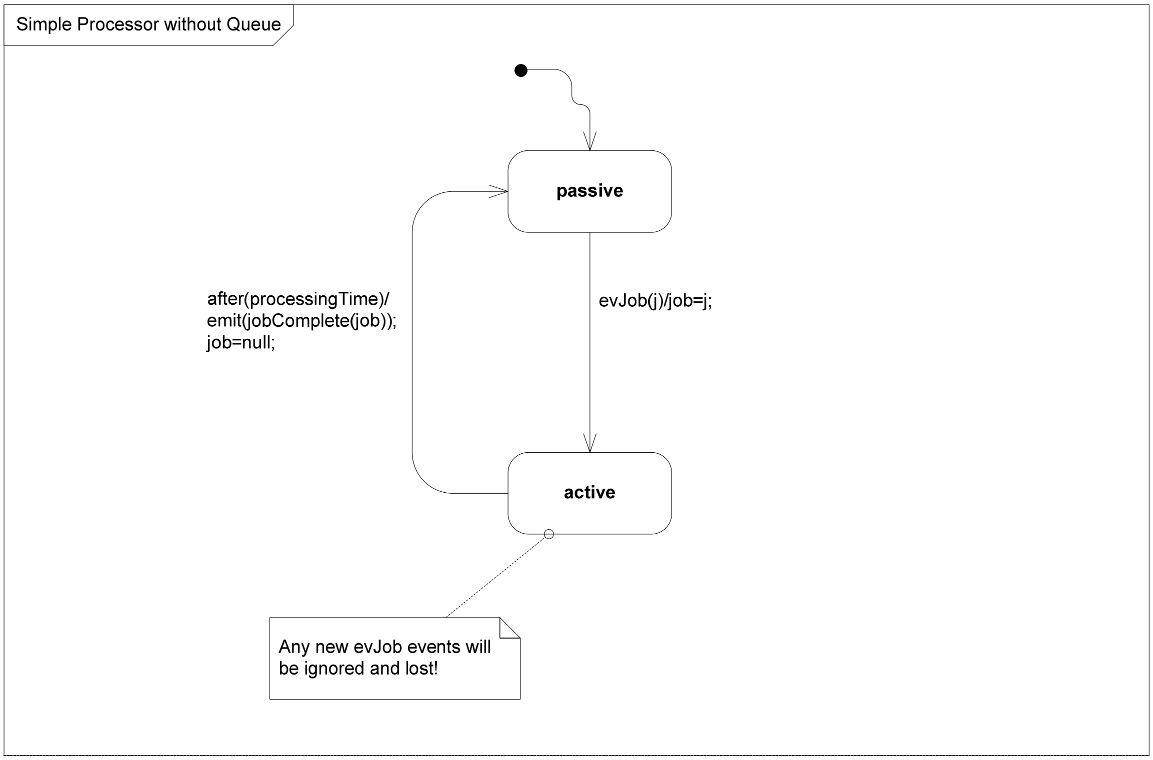

Figure 7 Simple Processor without Queue Atomic Model

The simple processor atomic model accepts jobs all of which have the same *processingTime* duration and emits a *jobComplete* token upon expiration of this time. DEVS allows multiple messages or events to occur simultaneously. In UML, however, events are handled one at a time. In this model we do not allow multiple jobs to arrive at the same time [*objection*#4]. Jobs that arrive during the processing of an existing job are discarded and never processed.

Simple Processor with Queue Atomic Model

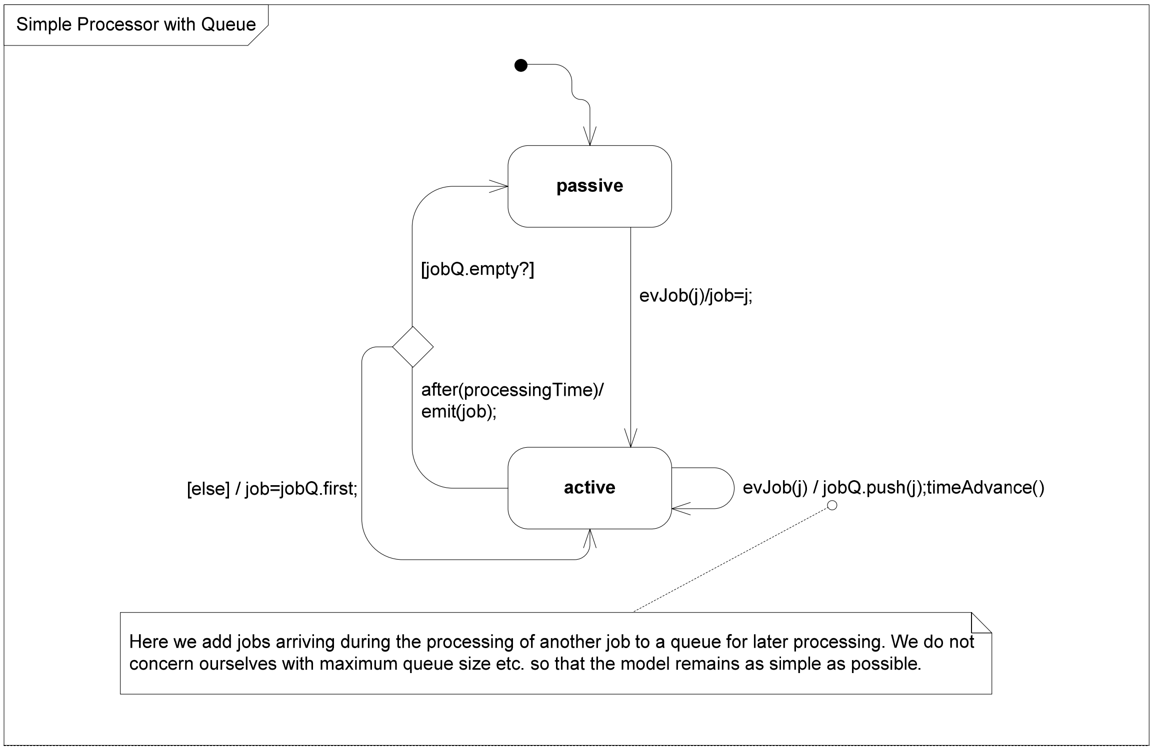

Figure 8 Simple Process with Queue Atomic Model

To ensure that arriving jobs are not lost, we introduce a state variable *jobQ* queue and accept jobs that arrive during the processing of another job. We again make use of the hypothetical *timeAdvance* function. Again the notion of multiple jobs arriving simultaneously in the *passive* state is not considered.

Simple Processor with Bundled Events Atomic Model

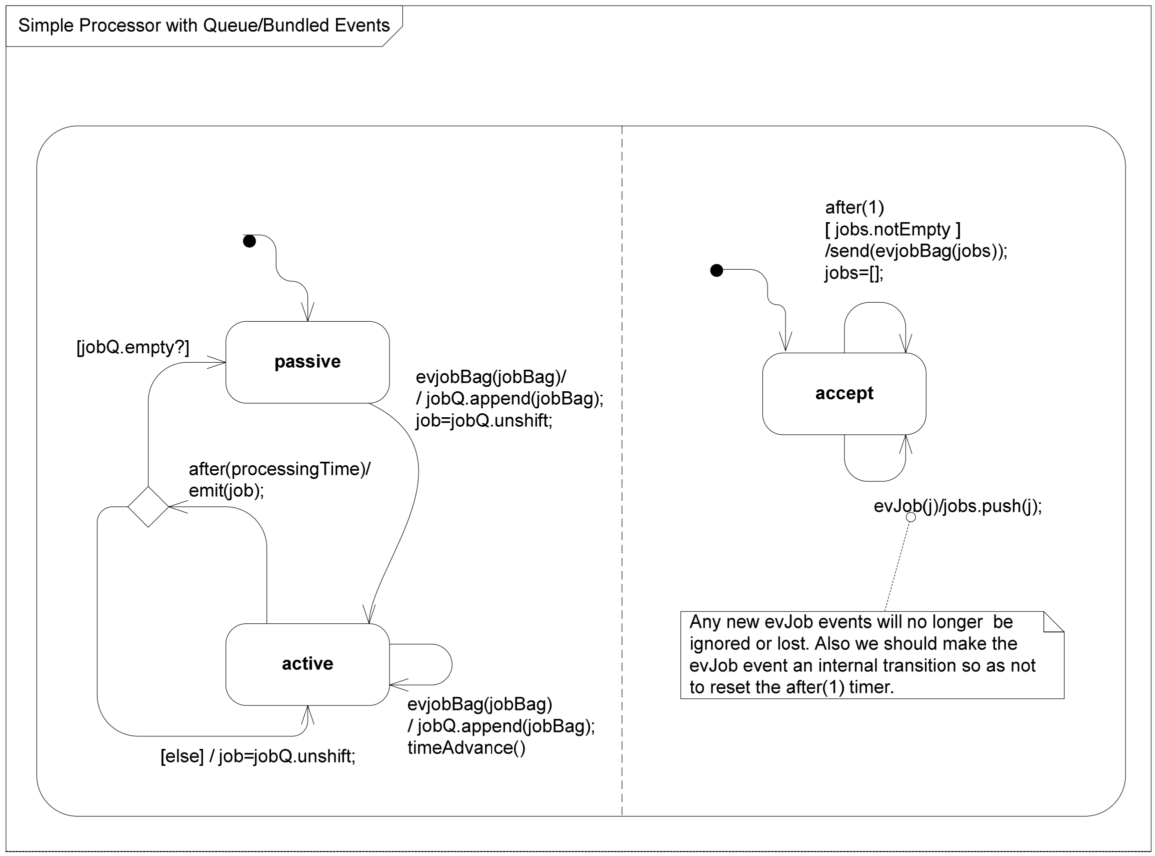

Figure 9 Simple Processor with Queue/Bundled Events(1)

We now try to account for the possibility that multiple events may arrive at the same time [*objection* #4]. Since the *same time* is a function of the ability of the observer to recognize events at a certain level of granularity of time, we make the assumption that the *same time* is a short interval generally a single unit, whether a second, nanosecond, clock period, and so forth. We have introduced an orthogonal region into the state machine specification. Such regions execute in parallel. In the *accept* state, we accept jobs and place them in a *jobs* bag. Then after each clock period (e.g. 1 second) if the jobs bag is non-empty, we generate an *evJobBag* signal. In the other orthogonal region, we await an *evJobBag* event and process all these jobs together.

In this case, we simply place the jobs in the job bag into a job queue and process each job serially, allowing for the fact that other job bags may arrive during this processing that need to be appended to our jobs queue. The reason we use a bag and not a list is simply to represent the fact that the jobs arriving during one clock period have no order since they happen "at the same time" even if during simulation one event precedes another (the clock is stopped). Obviously, this model is somewhat contrived but it demonstrates how we may handle the notion of modeling events being generated at the *same time*.

Note: In the UML there is no action language specification. Also, we could have used a convex pentagon symbol to represents signals but instead we have opted to use send([*recipient*.]*event*(*args*))) as the syntax for signal generation. Within UML, if the recipient is omitted, then the signal is sent to the state machine itself.

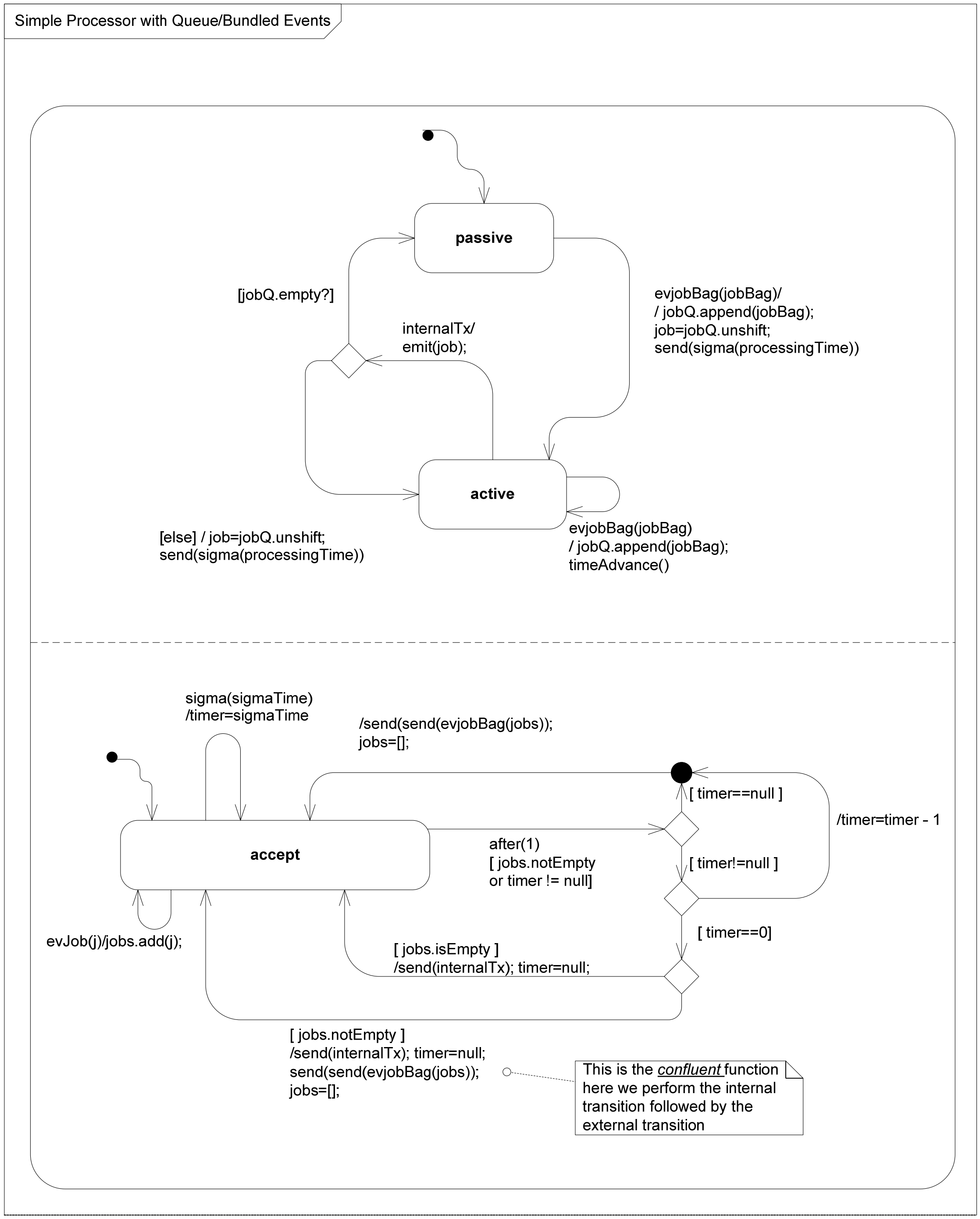

Figure 10 Simple Processor with Queue/Bundled Events(1)

We now elaborate on the region of the statechart that accepts jobs arriving at the same time, that is, within a specified time window. We now handle the situation in which the completion of a job may occur at the same time as the arrival of additional jobs. We removed the *after* event from the left hand/upper orthogonal region and introduced a special event called *evInternalTx,* which corresponds to invocation of the DEVS *internal transition function*. We generate an *evSigma* signal from the upper region which in turn is received by the lower region which sets a *timer* scheduled to elapse after *processingTime*. We then handle the situation where the timer expires at the same time as the arrival of a new job. To handle this situation, we have a number of decision nodes in the state machine to prioritize the *evInternalTx* event ahead of an external event – this corresponds to the DEVS *confluent* function. However, there remains a problem with this state machine in that the *evSigma* event will reset the *after*(1) pulse unless we make it an internal state transition. Next, we separate out this pulse, or tick, from the region accepting new jobs. Later we will encounter other reasons why it is necessary to separate the timekeeping from the state machine altogether.

The following diagram shows that we now have separated the clock into a separate region that generates an *evPulse* (or *evTick*) event every unit of time. This pulse event then is used to control when any existing *timer* will expire. A timer is created as a result of a *evSigma* event. At this point we have made significant progress towards addressing *objection* #2.

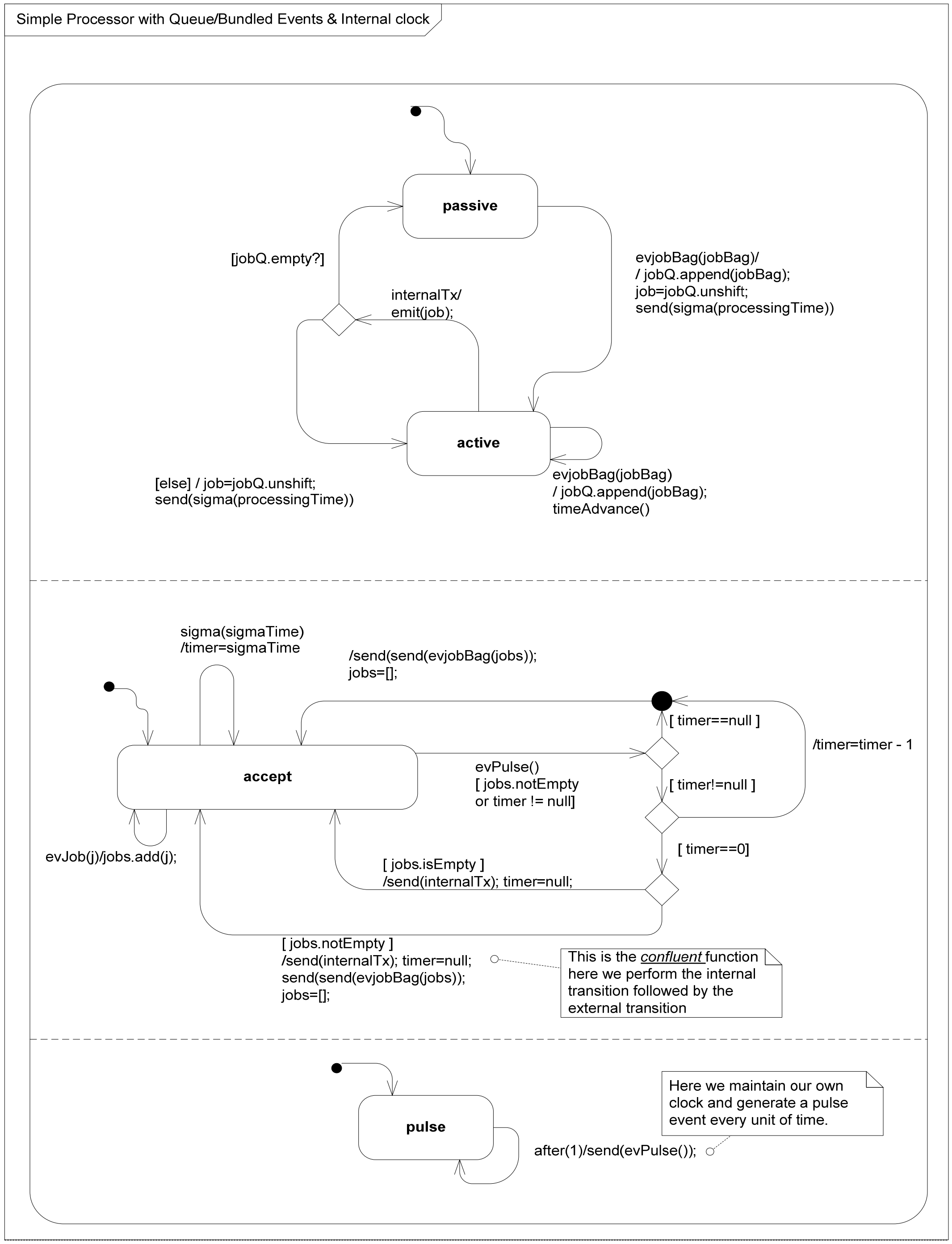

Figure 11 Simple Processor with Queue/Bundled Events & Internal Clock

Simple Processor with Separation of Concerns

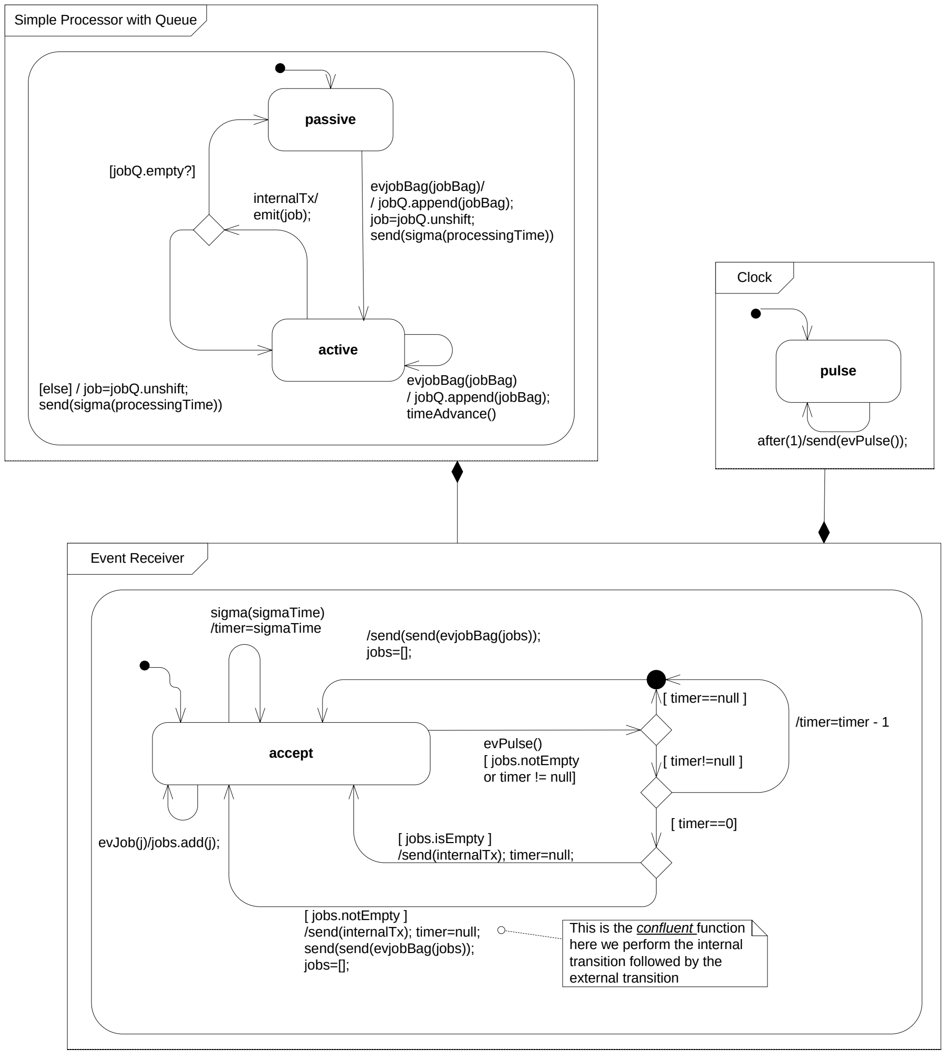

Figure 12 Simple Processor with Separation of Concerns

Here we break the state machine into three separate state machines. This does not change the model significantly and has the advantage of being easier to specify alternatives. But one area that is significant is the run-to-completion semantics. Such semantics in the UML apply to the entire state machine over multiple orthogonal regions. That behavioral guarantee would not apply if we choose to split into three separate state machines and may therefore introduce some unexpected subtleties. What is not addressed in any of the models discussed so far is the issue of synchronization of events across multiple models. For this we need to introduce the concept of a global clock.

Simple Processor with Global Clock

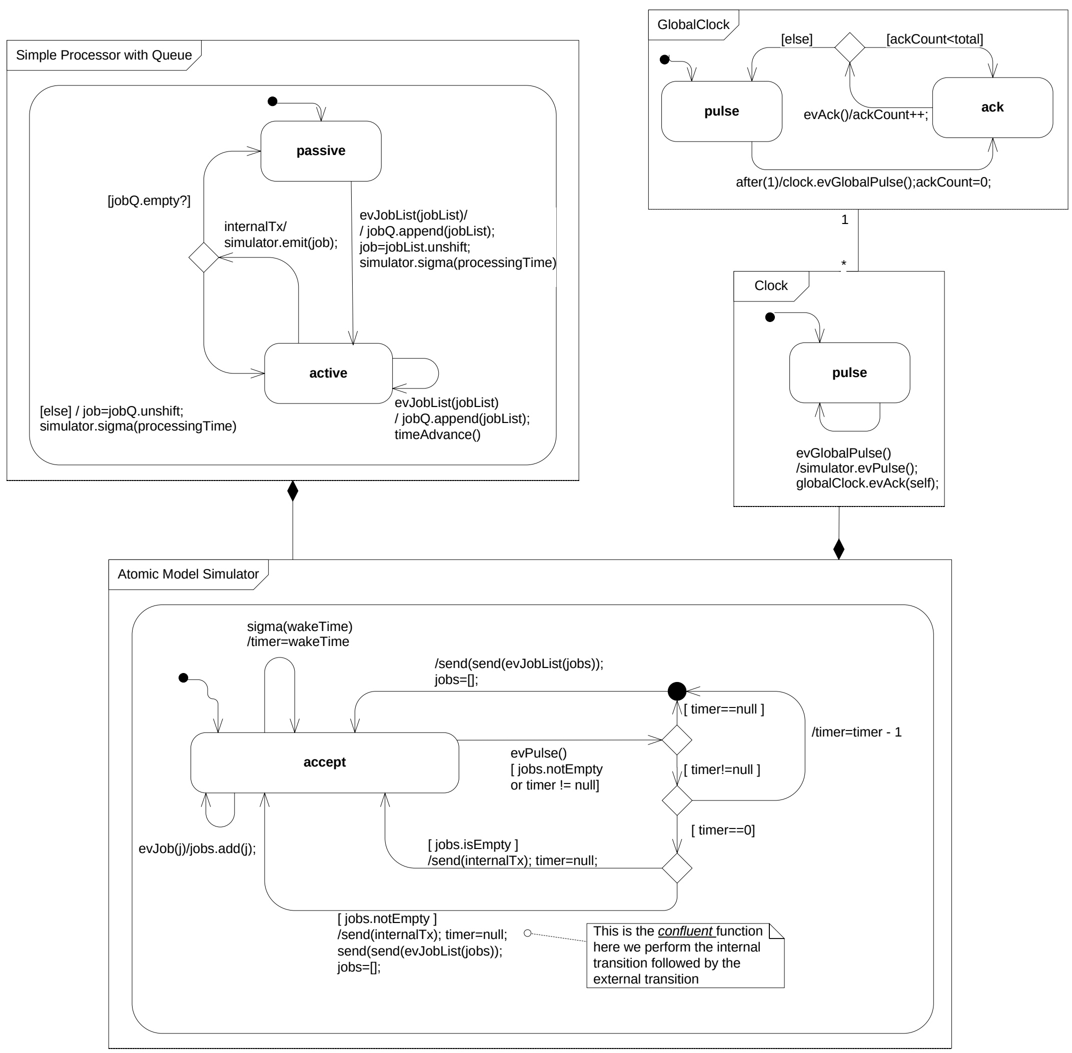

Figure 13 Simple Processor with Global Clock

We now introduce a Global Clock to support multiple different processors and other models which may need to have all their clocks synchronized. From a simulation perspective, we may need to have different models process an event at the same time. Additionally, when we communicate between these models we need to ensure that, where timing is relevant, the passage of time witnessed by both models is the same. The Global Clock only advances to the next time unit when it has received an acknowledgement from each of its client Clocks. It should be noted here that time is now counted in pulses, which are simulations of real time, but the elapsed real time between each pulse may be of varying duration. This is not a cause for alarm since this is simply a mechanism by which we can simulate the state of being in the active state for some duration of time. Simulated time is now synchronized across all other components via the Global Clock. It is possible that instead of time we could simulate CPU cycles. Also, we can get close to real time if the global clock accounts for the amount of time in the *ack* state. Further, we can allow our simulated time progress to be slower or faster than real time by multiplying our *after* time by a factor or omitting the *after* event altogether.

We have, thus far, retained the notion that the passage of time will still be monitored at the atomic model level. This represents a significant processing overhead which can be overcome by delegating responsibility for time progression to the state machine controlling the containing coupled model or optimally to the outermost coupled model.

Applied Example ~ Hummingbird Feeder

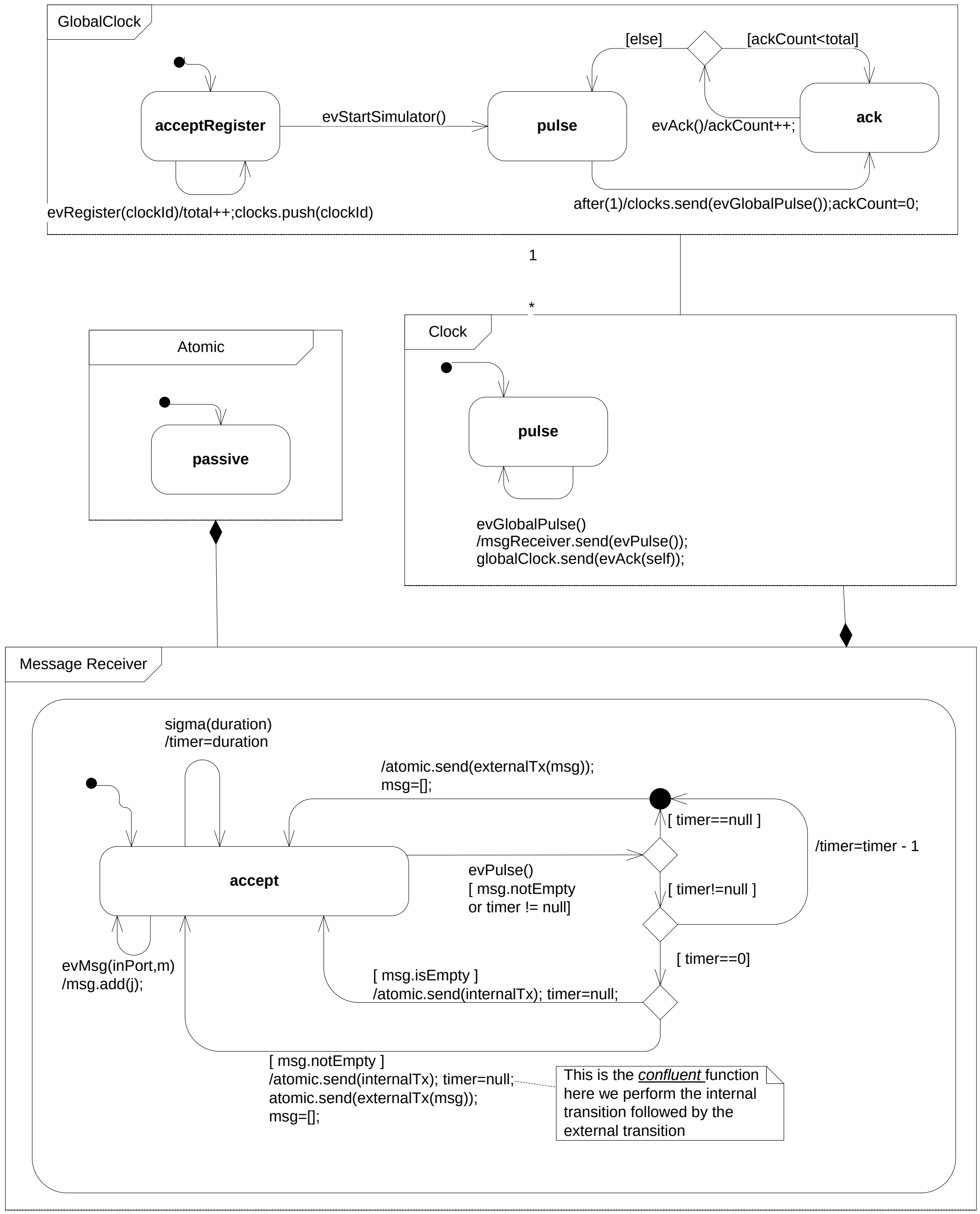

Figure 14 Hummingbird Feeder – Domain Independent

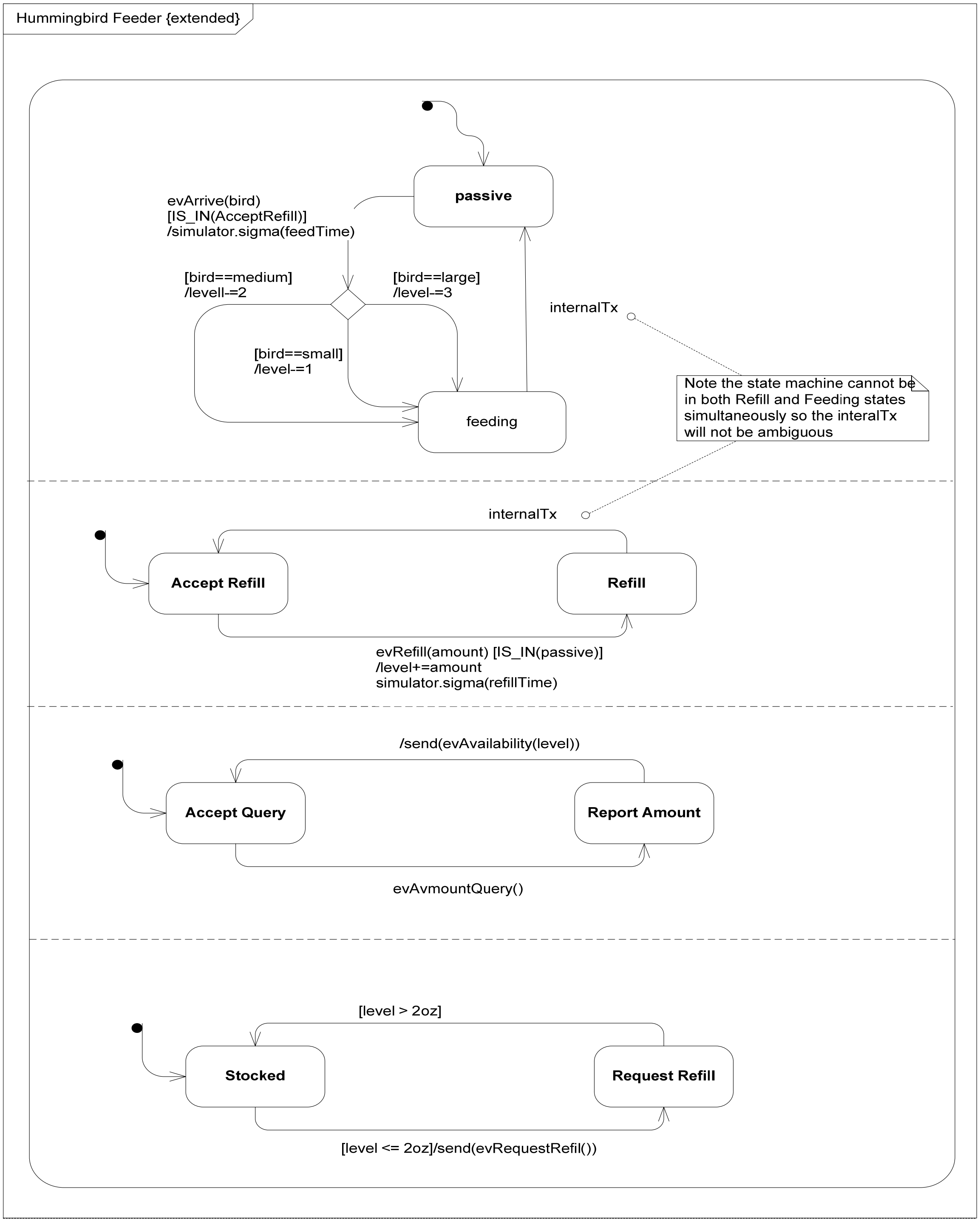

Figure 15 Hummingbird Feeder – Domain Specific

We now consider the advanced Hummingbird feeder introduced in (ACIMS, 66). We define the hummingbird state machine as an *extension* of the Atomic Model state machine. UML provides a clear specification for how state machines may be extended and elaborated upon. Care must be taken to ensure that the internal transitions are unambiguous. Observe that we cannot be in the *Refill* and the *Feeding* states at the same time.

At any point there should only be one state in which active processing is ongoing as expressed via an *evSigma* signal generation. This signal expresses the fact that there is processing that takes a certain amount of time, possibly zero, before it completes. Such states may, however, be *interruptible*. Note, in UML it is possible to have states in orthogonal regions within a state machine that may be performing actions simultaneously – for DEVS/UML, the restriction is that only one state responds to an internal transition event at any time.

At this point we have separated much of the simulation related infrastructure from the specification of the state machine for the particular model of interest. This is important since the objective of DEVS/UML is to allow architects to focus on the problem domain with little consideration for the issues involved in enabling simulation. We have the outline of a contract that the architect may follow which will enable models duly specified to be executable under a suitable DEVS framework.

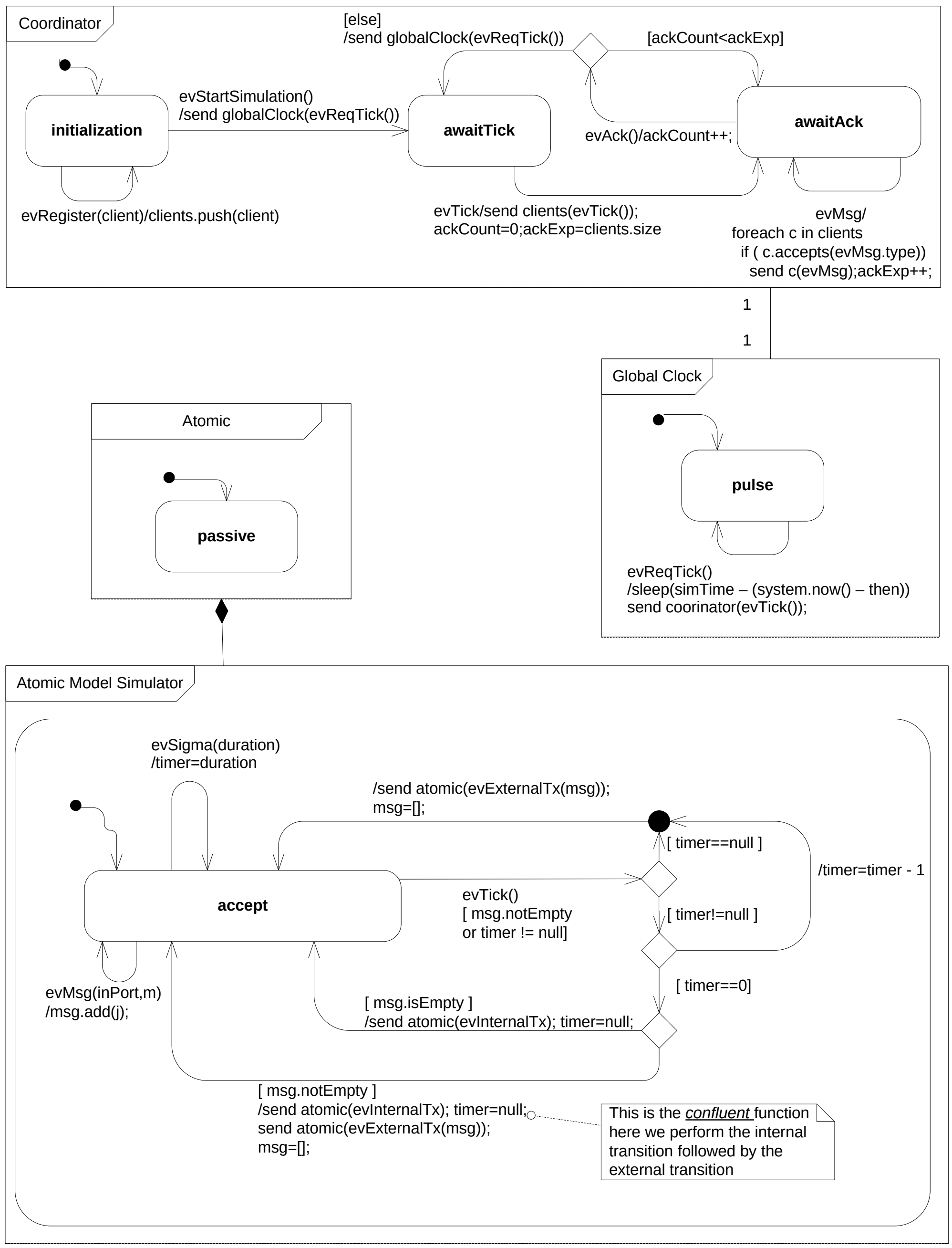

Figure 16 DEVS/UML Simulation Template I

In *Figure 16 DEVS/UML Simulation Template I,* we employ some of the terminology that we elected to use for the prototype system. Now, instead of each atomic model having its own clock, we have one clock and allow time to be globally controlled and disseminated with each tick. There is a contrivance in the diagram in which we try to account for the amount of time spent processing events between each tick that took zero time from a simulation perspective – the use of *system.now()* is <u>not</u> referencing a defined UML clock construct. Note that the role of coordinating time throughout the system is assumed by the *coordinator*. The coordinator is really just a *coupled model* – the outermost coupled model for the entire system.

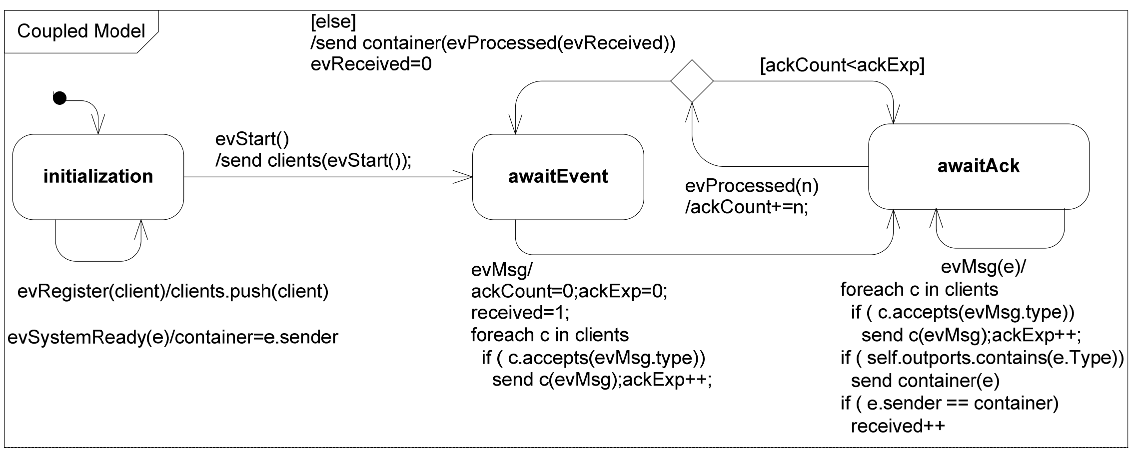

Figure 17 DEVS/UML Coupled Model

This is a simplified state machine for a *coupled model*. The coupled model is essentially the same as the coordinator except that it is completely reactive with regards to message arrival in the main event loop, whereas the coordinator prompts the *global clock* for a tick event; all other coupled models simply wait for a tick event and then pass that along to the models they contain. In this state machine, we allude to an initialization phase during which we accept registrations from contained sub-models, and then we await an event indicating that the system is ready to enter into the event loop to start processing all the events for a simulation execution cycle. With this model, the coupled model has no conception of time and its only function is to relay events down to its contained sub-models or back up to its own container. Note also that peer models contained within a coupled model do not communicate directly with one another and instead they go through the coupled models to send events to each other *regardless* of whether the ports between these models appear to be directly connected. This simplifies the specification of the atomic model simulator since all communication is restricted to its container or its atomic model.

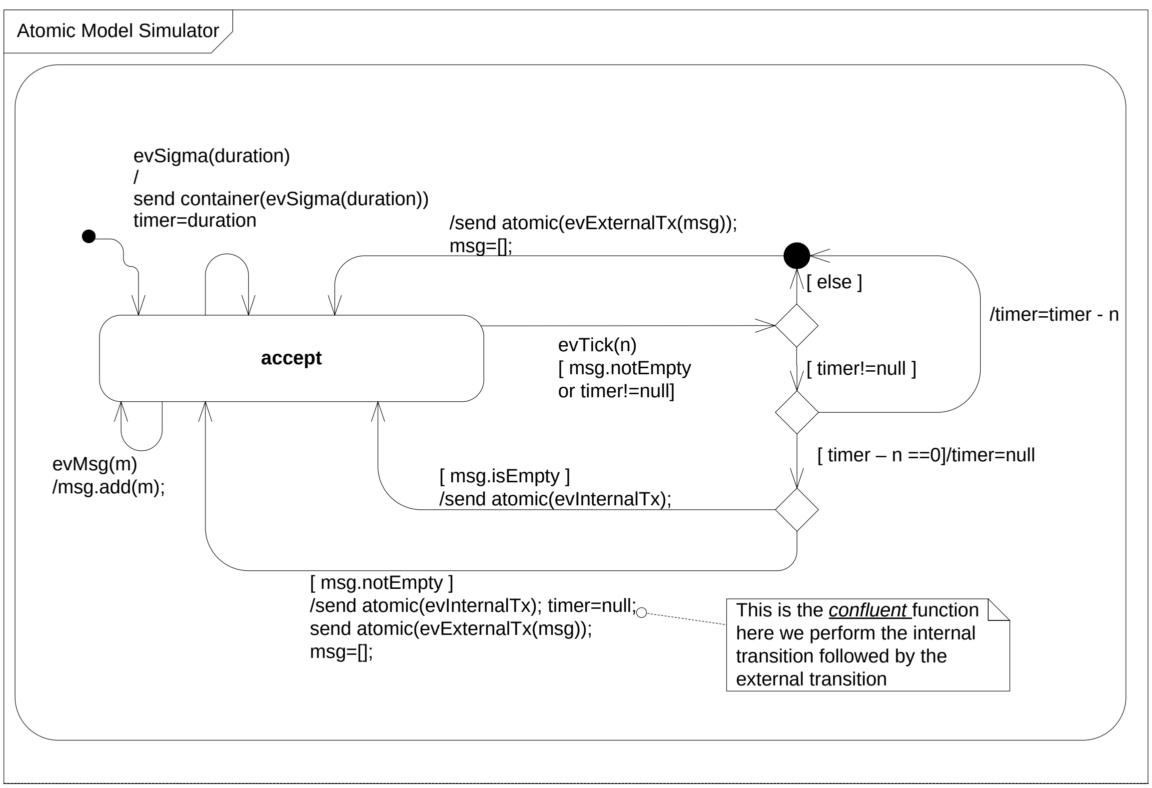

Figure 18 DEVS/UML Atomic Model Simulator

In order to allow for a more efficient simulation execution we can have the *evSigma* events issued by the atomic models percolate up through the atomic model simulator to the coupled model and eventually up to the *coordinator*. This way atomic model simulators do not need to receive every tick but only when the *evSigma* value expires or when they have external transition messages. Each *tick* event now has an *n* parameter that specifies the amount of time that has elapsed since the last tick. Within the same clock cycle, there may be a need for the message bag to be relayed to the atomic model multiple times (each time with a new bag). The fact of its being in the same clock cycle should be transparent to the atomic model simulator with a new tick event having a value of zero for *n*. And note that the structure that holds the messages is not an array or other ordered structure but rather an *unordered bag*. Also note that if the atomic model wishes to indicate a *passive* state where *evSigma* has a value of infinity, it can pass a value of -1 as the duration of the *evSigma*. In this way, the atomic model simulator can essentially remove any existing timer and the atomic model will remain dormant until it receives another external event. When an external event is received the $timer - n == 0$ condition will evaluate to false since $timer - n$ will be less than zero. Thus that the *confluent* function will not be entered. One thing that has not been addressed thus far is passing the elapsed time since the last internal or external transition to the atomic model itself. This is easily accommodated by adding this time as an argument to the messages sent to the atomic model from the atomic model simulator. We now have covered most of the fundamental issues necessary to present the DEVS/UML prototype simulation engine and to articulate a contract for DEVS/UML compliant state machines.

Representing Coupled Models in UML

The representation of coupled models in UML is made significantly easier in UML 2.*x* through the provision of new compositional constructs. Components may contain sub-components in a hierarchical fashion similar to coupled models containing models. Components may be connected to one another and attached via ports again similar to DEVS. In DEVS connections between ports are unidirectional, whereas, this is not necessarily the case in UML. Therefore, in DEVS/UML, all ports should be unidirectional. In UML, ports may have required or provided interfaces. In DEVS/UML a port may either provide an interface or require an interface but not both since this implies bi-directionality. These interfaces may be represented diagrammatically via UML lollipop notation or in DEVS/UML via the convention of placing input ports on the left hand side of a component and output ports on the right hand side of a component. In UML, ports may have a multiplicity greater than one; this is not the case in DEVS (and hence not allowed in DEVS/UML). In UML, port may be unnamed; they must be named in DEVS and thus in DEVS/UML. In UML, connectors need not attach to components (more correctly *parts)* via ports; this is not an option in DEVS and hence not an option in DEVS/UML. If ports are specified to provide or require an interface, there should be only one such interface specified in DEVS/UML. In (Huang and Sarjoughian 2004) there is a mapping for coupled models into UML-RT structure diagrams, but the use of the UML Profile for Schedulability, Performance and Time Specification (OMG 2005) is unnecessary when mapping from DEVS to UML.

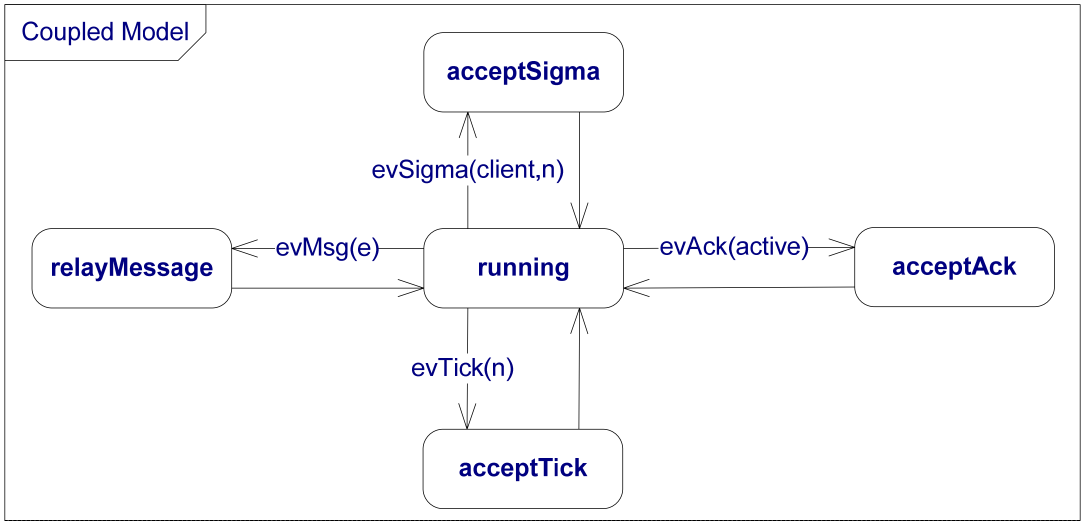

Figure 19 DEVS/UML Coupled Model Simulator

Here we see the events that a coupled model processes during its main event loop. The signal *evTick* comes from the containing model and is relayed to any *active* model and to any model with an expiring *timeNext*. The *evAck* signal is generated by a coupled model when it has received an *evAck* from each sub-model to which it sent an *evTick* signal. The *evAck* signal has one parameter, *active*, which indicates whether any sub-model is active. Note, an active model is any model with messages pending delivery. In this way, the coordinator knows whether there are any active models in the system, and if so, whether another simulation cycle is necessary. An atomic model must generate an *evAck* in response to an *evInternalTx* event after it has generated any external output messages – that is, at the end of its output function.

Clock Cycle/Simulation Cycles

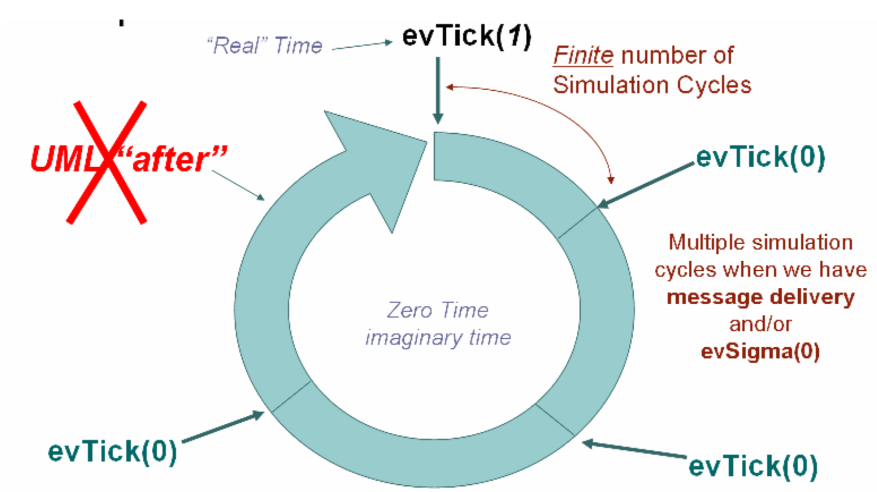

Figure 20 Clock/Simulation Cycle

As seen, a clock cycle may have multiple simulation cycles. During the first simulation cycle, a model receives an *evTick* event with a parameter indicating the amount of time that has elapsed since the last *evTick* message received. A coupled model will only receive an *evTick* message in the event that it has a *timeNext* of zero. Since atomic models may generate outputs in response to an *evInternalTx* signal those messages must be delivered during this clock cycle. However, it is preferred that these messages be delivered as a bag of messages and not delivered individually. To facilitate delivering bags of messages, a coupled model marks as active any model to which it sends an *evMsg* message. Atomic model simulators do not pass messages directly to atomic models upon receipt but rather wait for an *evTick(0)* message to arrive. An *evTick(0)* will never be the first *evTick* message in the clock cycle since there will never be undelivered messages from a previous clock cycle. Thus, the first *evTick* message in a clock cycle will always have a non-zero amount of time elapsed. An *evTick(0)* may also be triggered by the generation of an *evSigma(0)* message by an atomic model.

```
acceptTick

entry/
  ackCount=0;ackExp=0;isActive=false;
  foreach m in models
    m.timeNext-=curr_event.n
    m.timeElapsed+=curr_event.n
    if ( m.timeNext == 0 ){
       send m(evTick(m.timeElapsed));
       ackExp++;
       if activeModels.contains(m);
         activeModels.delete(t.client)
       timeNext.delete(t);
    }
  }
  foreach m in activeModels
       activeModels.delete(m);
       send m(evTick(0));
       ackExp++;
  }
```

Figure 21 EvTick

The algorithm for accepting *evTick* events in a coupled model simulator is presented above. For each model contained within the coupled model, a *timeNext* is maintained containing the amount of time remaining until the next scheduled *evInternalTx*. Also, a *timeElapsed* is maintained containing the amount of time elapsed since the last *evTick* was sent to that model. If there is no scheduled *evInternalTx,* then the time remaining will be a negative number. Atomic models send an *evSigma(-1)* to indicate a passive state where the sigma value should be considered as infinity. A coupled model is considered active if any of its sub-models are active.

Once a coupled model sends an *evTick* to a sub-model, that sub-model is no longer considered active.

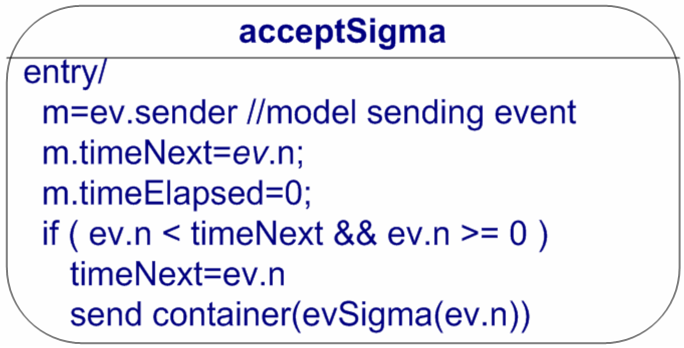

Figure 22 EvSigma

The *evSigma* signal is initially generated by the atomic model and relayed through the atomic model simulator to the coupled model. The coupled model stores the time contained in the *evSigma* event as the *timeNext* for the model. As part of the event signal, the sender is also identified. A coupled model also contains a *timeNext* for itself representing the earliest *timeNext* of all its sub-models. If the arriving *timeNext* is sooner than the coupled model's earliest *timeNext,* then it communicates this new *timeNext* to its own container.

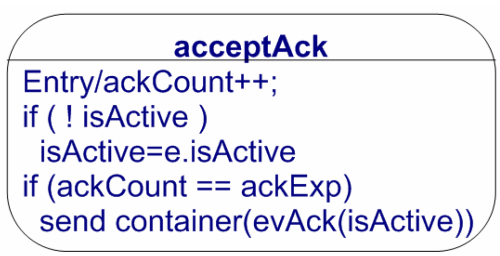

Figure 23 EvAck

The *evAck* signal is generated by the atomic model when it receives an *evInternalTx* signal and after it has completed generating any external *evMsg* messages. The *evAck* message has one parameter, called *active*, which is always set to false in the case of an atomic model. For coupled models, the *evAck* signal is generated upon receipt of the final *evAck* from each of its sub-models and the *active* parameter is set to true depending upon whether there are any active sub-models.

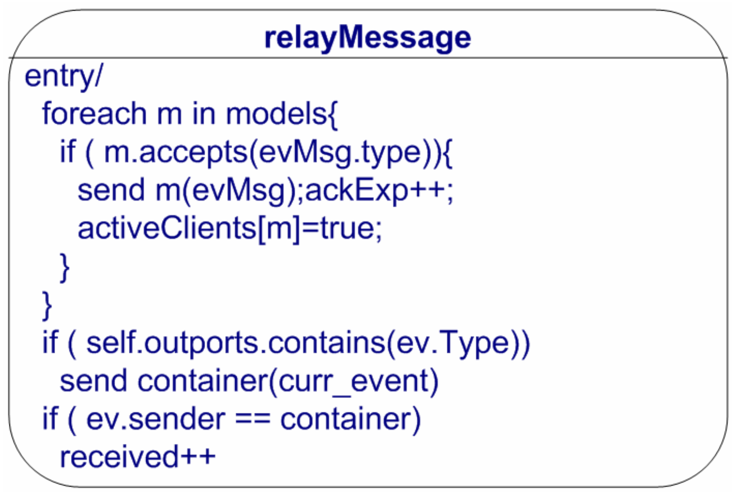

Figure 24 EvMsg

The *evMsg* signal is generated by an atomic model as part of its output function. The output function is the logic performed upon receipt of an *evInternalTx* signal and before the *evAck* signal is generated. The output function is the only time during which external *evMsg* messages may be generated. The *evMsg* type is itself an abstract message type. The atomic model must send a concrete sub-class of this message type. For a coupled model, when an *evMsg* message is received, it is relayed to any sub-models that have the corresponding concrete message type as an input. The *evMsg* is also relayed to the containing model if the coupled model has itself the corresponding concrete message type as an output.

C H A P T E R 4

DEVS/UML PROTOTYPE

A prototype DEVS/UML Simulator has been written to support this paper and is now presented. Initially, the system was developed using the JRuby language which is a pure Java implementation of the Ruby language executing in a Java Virtual Machine. This language has the benefit of having access to all java libraries such as Swing or SWT for graphical user presentation but also it is a highly dynamic language allowing a developer to express concepts in fewer lines of code than the equivalent Java code. It has the drawback of poor performance which for prototypical situations is not a significant impediment. However, JRuby itself proved to be an unsuitable language for this task since the multi-threaded implementation of the simulation engine exposed a number of bugs that required fixes in the JRuby interpreter. The prototype code was then completely rewritten in Java and the JavaDoc documentation is available as an accompaniment to this thesis.

Interestingly, another dynamic language, Python, was used to develop another lightweight DEVS simulation engine (Borland 2003).

Prototype Architecture

The important classes that participate in prototype beyond the model objects are a graphical user interface (GUI), a thread management object (DevsRunner), a messaging infrastructure (MessageSpace), and a global clock.

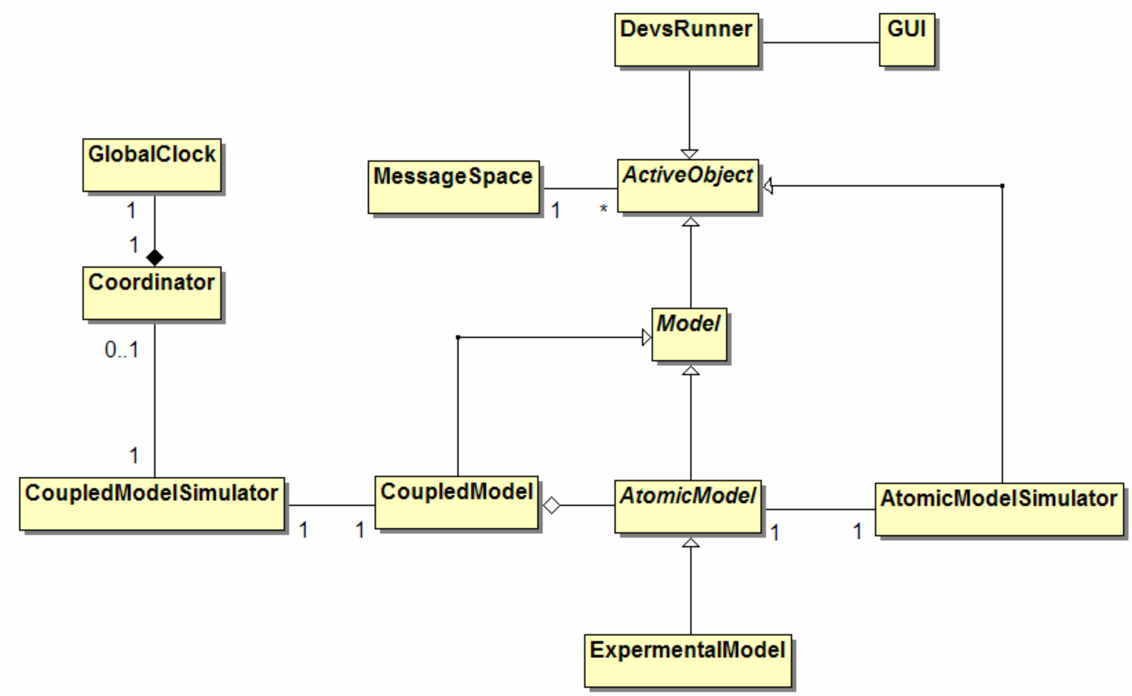

Figure 25 DEVS/UML Prototype Simulator

Event Types

The modeler will primarily be sending events based on the EvMsg event type. This is the event type from which all other event types that are application specific ought to be derived. In addition, the EvInternalTx and EvSigma will be specified at the atomic model level. The atomic model simulator and coupled model simulator are primarily interested in EvTick, EvAck, and EvTick.

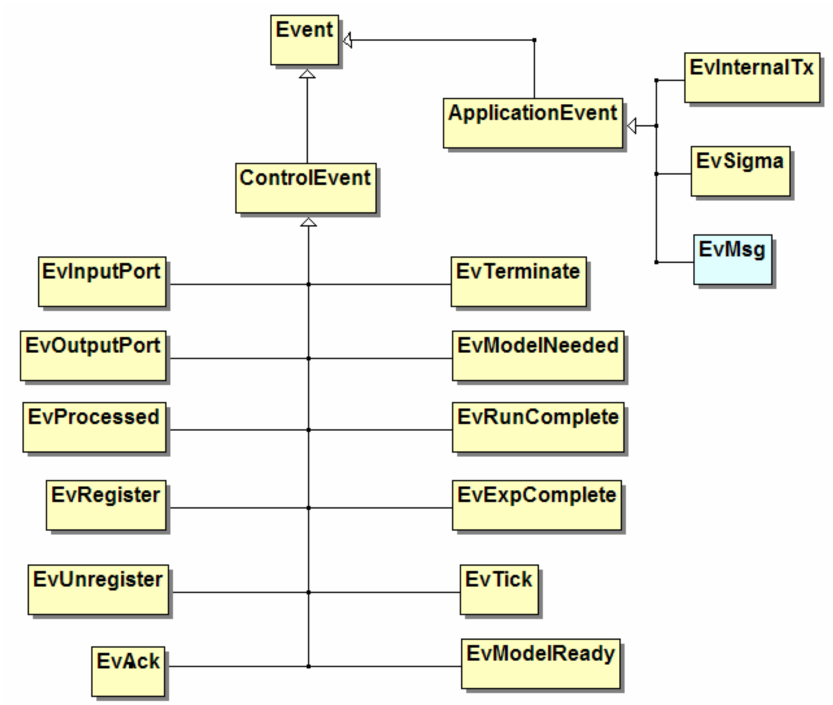

Figure 26 DEVS/UML Event Types

The additional event messages allow coupled models to accept registration of contained sub-models along with specification information. It is intended that the coupling of models support a maximum degree of runtime dynamics.

Graphical User Interface

For the prototype this is a somewhat trivial interface allowing a user to load a model and to execute an experiment. The execution speed can be controlled so that it is faster or slower than real time, and the user can step through each clock cycle or let the simulation run through completion and examine the results at the end. The output approximates a sequence diagram insofar as time descends vertically and messages pass horizontally between objects. In order to facilitate debugging, message dialog boxes can be displayed during execution via the inclusion of debug messages in the Java source code itself.

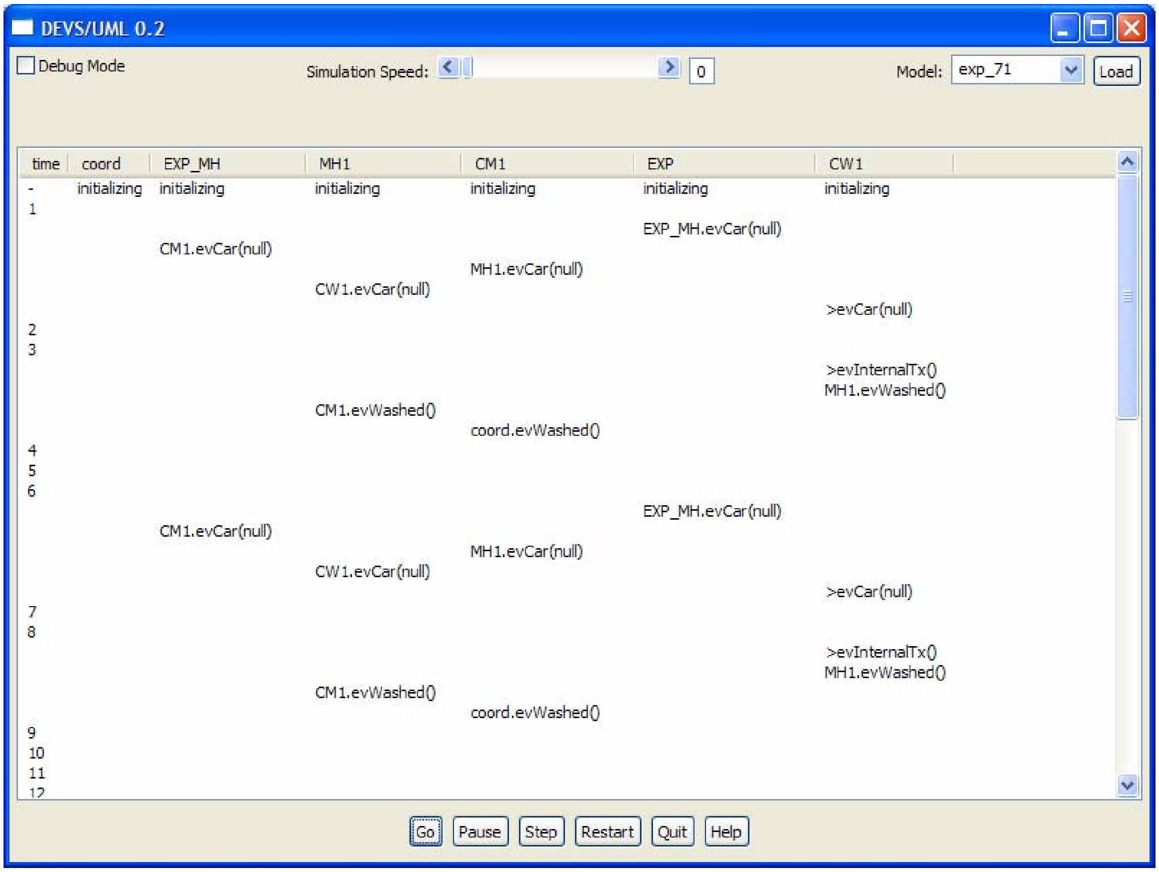

Figure 27 DEVS/UML Prototype Simulator

By default we do not show the message space object, nor do we show the control messages such as registration since these are of little concern to the modeler. Although trivially simple from a user's perspective, the challenges involved in implementing the underlying simulation engine were far greater than imagined with about eleven thousand lines of Java code necessary for the prototype.

Models

Both atomic and coupled models are sub-classes of the Model class. The Model class has some logic and attributes common to both coupled and atomic models, and it is in turn a sub-class of ActiveObject, which contains all the logic necessary to communicate with other objects, specifically it has the logic to interface with the message space. In the prototype developed in support of this thesis, this is a sub-class of the Java Thread class. A Coordinator class is a special instance of a coupled model – it is the root model for an execution.

Global Clock

The global clock is relatively trivial; it simply issues a tick to the root coupled model (coordinator) and awaits an acknowledgement. Depending on the simulation speed, the delay between each tick is adjusted to run faster or slower than wall clock time. Since the execution of the simulation itself, such as sending and receiving messages, takes some time, this time is subtracted from the amount of time to sleep between clock ticks. Thus, by specifying a simulation time of zero, thereby indicating that we should not sleep between ticks, the simulation speed is dictated by the speed of the computer and its resources.

Processes & Threads

The DevsRunner singleton class is responsible for the creation of all of the objects and associated threads necessary for an execution run. Each coupled model executes in its own thread. Each atomic model had *two* associated threads – one for the atomic model and one for its atomic model simulator. In DEVSJAVA (ACIMS) this was found to create scalability issues, and instead of creating separate threads for each model, just two threads were created and the independent execution of the models was simulated by these threads. All communication between the threads is handled by the singleton MessageSpace object.

Model Communication

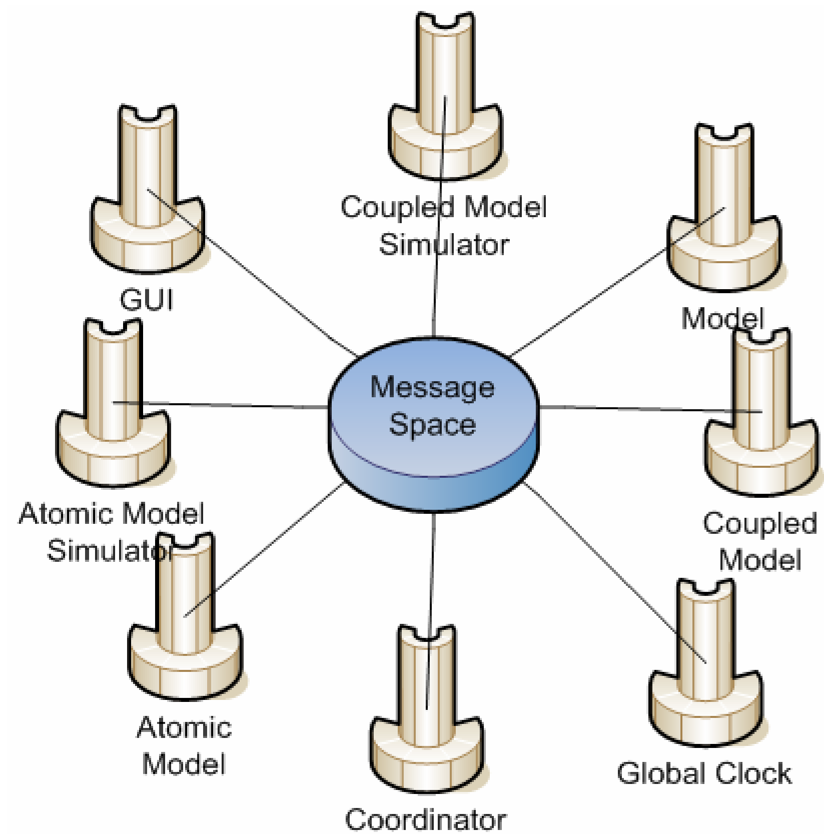

Figure 28 DEVS/UML Message Space

The messaging architecture used to develop the DEVS/UML prototype involved a custom written implementation of a tuple space (Carriero 1992). Although orthogonal to the issues discussed in this thesis it is worthwhile providing a brief discussion concerning the implementation and the possibilities that a tuple space provide for an implementation of a simulation engine. The tuple space implemented is intended to simplify communication between the models by eliminating direct model-to-model communication and by employing a backboard type system whereby all messages are written to the tuple space and consumers of these messages can register to listen for messages that match a certain pattern. For example, a coupled model can simply request the next tuple written to the tuplespace that is addressed to it. Such calls are blocking calls. Alternatively, a tuple space client can check to see if such a message exists without being blocked. This messaging architecture is based on the work of David Gelernter of Yale and has been widely adopted. It has been incorporated into Java's *Jini* where it is called *JavaSpaces* (Sun Microsystems). Although the prototype has a primitive implementation of this mechanism, it can easily be replaced with a commercial strength implementation such as *JavaSpaces*. The beauty of this approach is that all of the inter model communication is decoupled and replaced with a very simple protocol that allows models to potentially execute across different processes and hosts.

Tuple Spaces & Multi-Processor Architectures

A potential benefit of using tuple spaces as the messaging architecture for a DEVS simulator is the possibility of simplifying the specification of multi-server architecture wherein a farm of identical processors is available to handle requests and a coordinator distributes requests to these processors depending on load or some such criterion. In DEVS if each processor is identified with a unique input port then the multi-server coordinator must have a matching port for each processor in the farm; otherwise, if the processors are all identical instances with the same input port names then the requests will be broadcast to all processors with an embedded address indicating which processor is responsible for handling the request. By using a tuple space as the messaging mechanism, the first processor ready for a request can simply remove the message request from the tuple space and the specification of the multi-server coordinator can have one output port mapped to the all the processors but only one processor (the first to grab it) will receive the message. Employing a tuple space in such a manner is not without theoretical and modeling consequences since the tuple space becomes, in effect, part of the formalism – an examination of the full consequences of which is beyond the purview this thesis.

Objects in Tuple Spaces

In the prototype developed for this thesis only messages were placed in the tuple space. In this primitive implementation the tuple space only served as a data store. However, in theory in tuple spaces, and in practice in the JavaSpaces implementation, we are not restricted to data but entire objects can be placed in the tuple space providing access to all the methods or services provided by the object. This opens up the possibility for a very different simulation engine than that developed in support of this thesis. For example, all atomic model instances could be placed in the tuple space, essentially registering each as a service. A range of architectural possibilities would then become available.

Model Initialization

Each model and the atomic model simulators go through the same initialization process. First wait for the message space to be initialized. Second, await availability of its container. Third, register with the container itself.

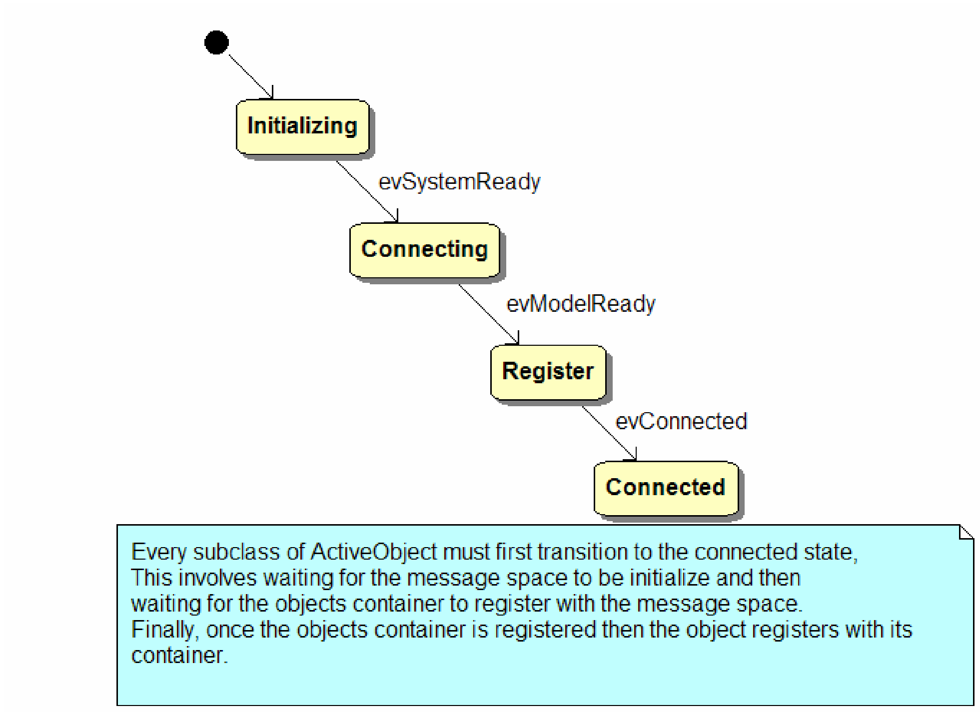

Figure 29Atomic Model Specification - Carwash

Atomic Model Specification

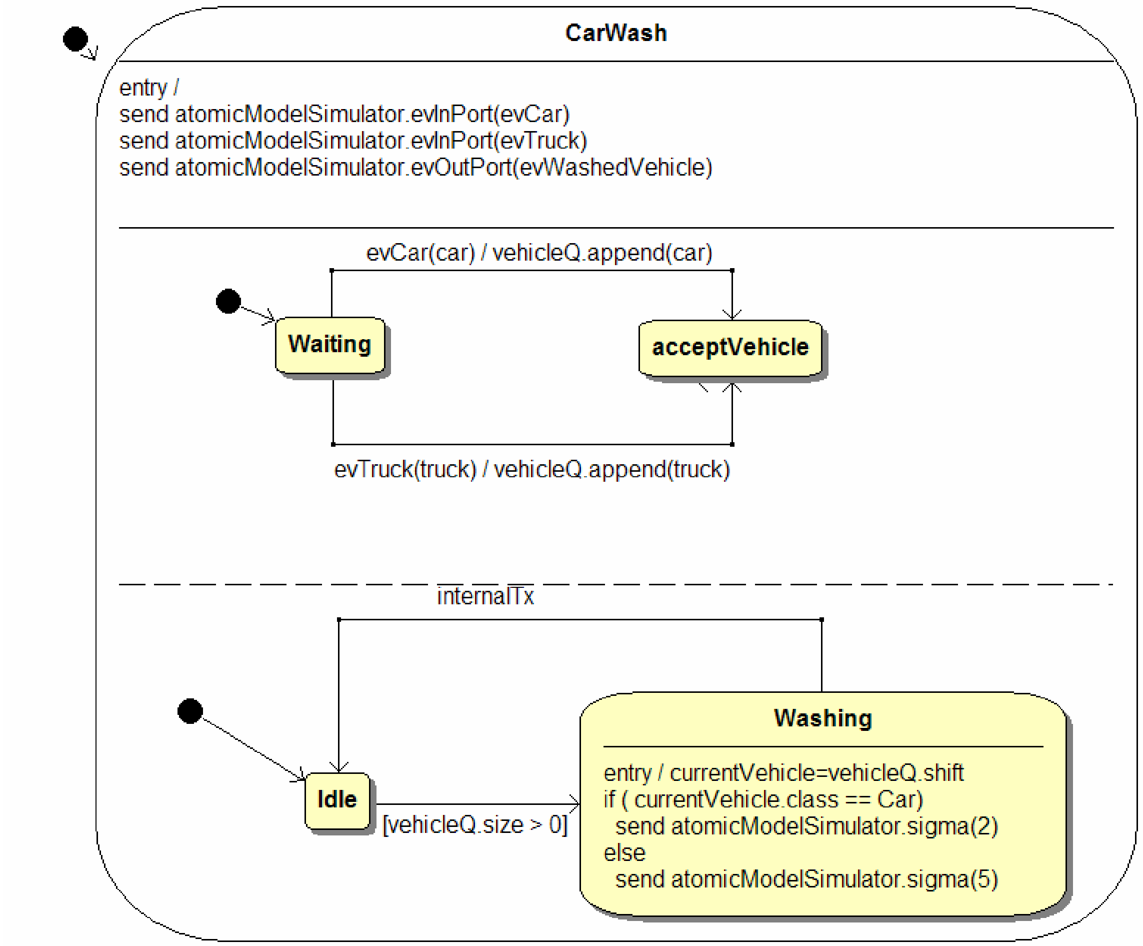

Figure 30 Atomic Model Specification - Carwash

Here we show a compliant DEVS/UML state machine. Note the *evSigma* signal that is generated in the *Washing* state and sent to the *atomic model simulator*, and the *evInternalTx* trigger event which in turn gets generated by the *atomic model simulator* upon the elapsing of the sigma timer. We do not use *after* since this would compromise our ability to execute a valid simulation. All events generated by the atomic model must be generated only upon receipt of an *evInternalTx* event.

Experiment Specification

In the prototype developed, the experiments are very simple. Upon startup, a schedule of events is specified. During execution, an initial *evSigma* signal is generated for the first scheduled event. Upon receipt of the *evInternalTx* event the first output event messages are generated and sent to the coupled model in which the experiment model is contained. Then the next *evSigma* signal is generated, and so on. When all outputs for the experiment have been generated, an *evExperimentComplete* message is generated. When any other experiments have also completed, the execution cycle is terminated. During initialization, the coordinator is informed of all experiments. The design also allows for experiments to be activated midstream.

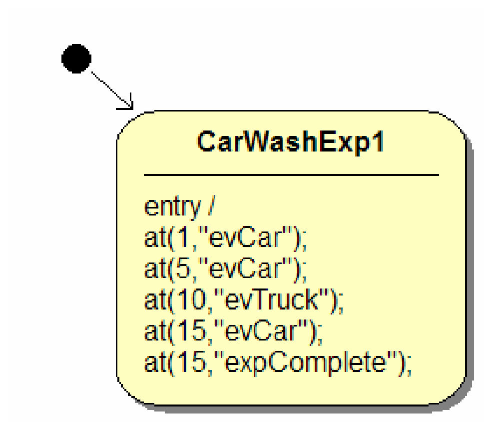

Figure 31 Experiment Model Specification – Carwash

An experiment is the vehicle for injecting events into a system in order to perform a simulation. The experiment is a sub class of an atomic model and behaves in much the same way except that it has additional logic to support the generation of events according to some pre-defined specification. For each experiment there should be a corresponding concrete class where the setup method is implemented much the same as an Atomic Model. This involves the definition of the input and output ports of the experimental frame model and may optionally include a list of other models upon which the experiment depends. The concrete experiment implementation may include multiple scenarios, one or more of which may be executed during a single simulation execution.

Atomic Model Simulator

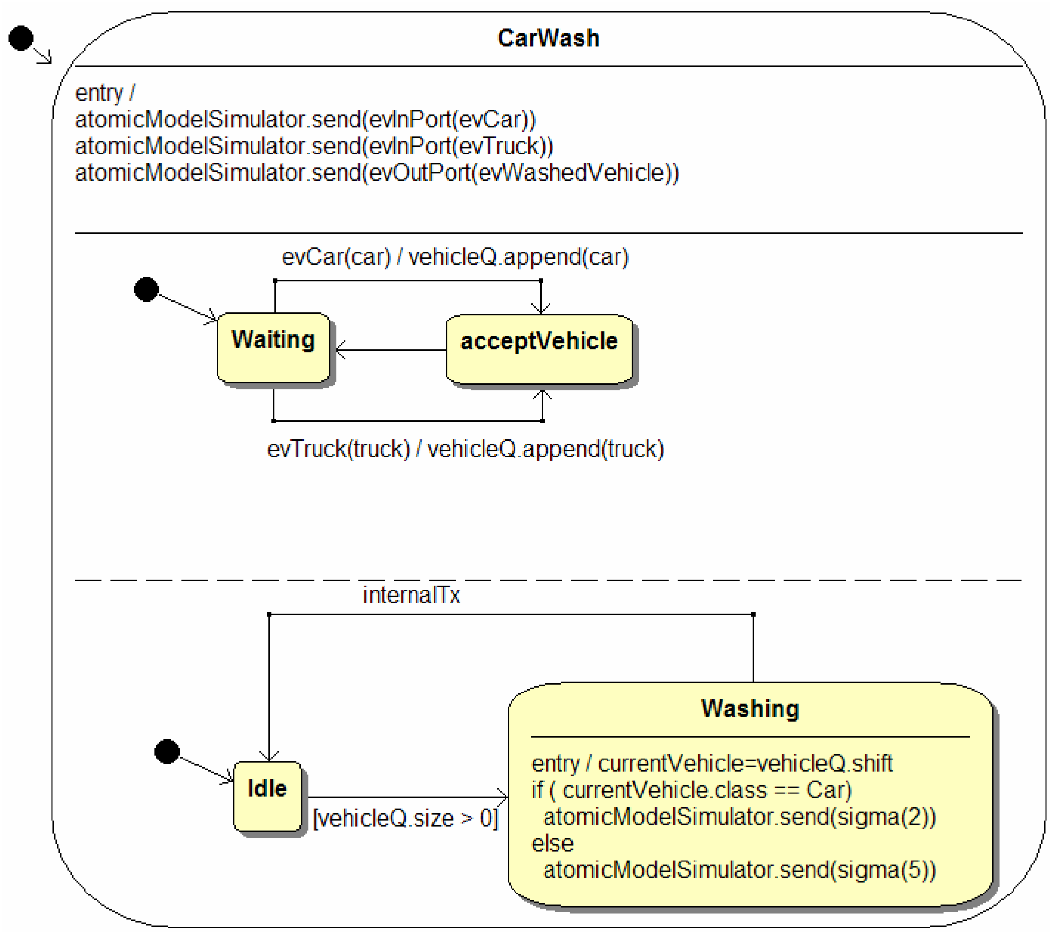

Figure 32 Atomic Model Specification - Carwash

Here we show a compliant DEVS/UML state machine. Note the sigma signal that is generated in the Washing state and sent to the atomic model simulator.

Coupled Model Specification

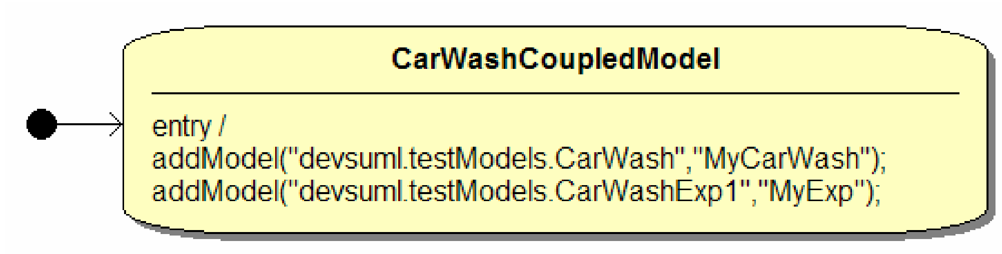

Figure 33 Coupled Model Specification - Carwash

Within the prototype, events are either application or control events. Application events are those used for passing the application messages between models during execution. Control events are those used to control the execution of the simulation itself, such as the registration of models with their respective containers. Since most of the classes used in the simulation engine are themselves fundamentally state machines, they receive messages such as the declaration of the input and output ports of the contained models within their event loops via event signals from the atomic or coupled model instances that they contain.

The signal events that an application generates corresponding to the outputs it generates do not need to be subclasses of ApplicationEvent but it is useful to have these events partitioned into a separate hierarchy from the control events. In this way the specification and implementation of a coupled model and atomic model simulator can be generic in nature and not have specific reference to the event types that they relay.

Time Coordination

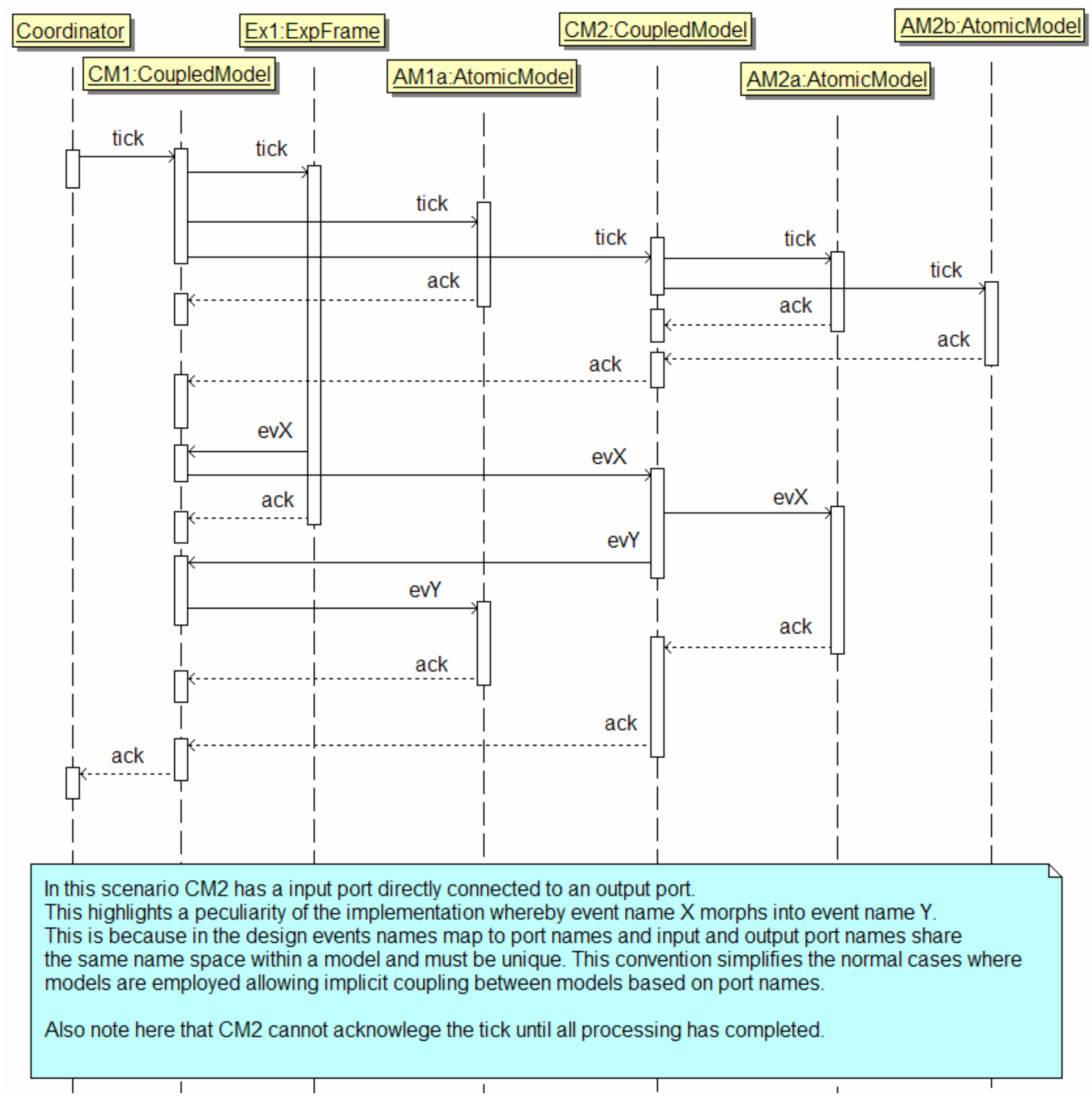

Figure 34 Clock Cycle ~ Scenario I

In *Clock Cycle ~ Scenario I,* we show how the simulator handles advancing the clock for one tick or cycle. In this example we have a coupled model receiving a tick and then relaying it to its component models *Ex1*, *AM1a* and *CM2*. *CM2* in turn relays the tick to its atomic models *AM2a* and *AM2b*. We then go through a cycle whereby outputs are generated by atomic models and routed to their destinations, and once all this message flow completes, the tick can be acknowledged. In this scenario we presume that all models receive a tick message regardless of whether they have a scheduled internal transition. In practice we would not send tick messages where there is no scheduled internal transition. Furthermore, we would skip all ticks until there was an internal transition scheduled somewhere within the system. This requires relaying the time of next internal transition from each atomic model to its coupled model on up to the outermost coupled model also called the *coordinator*. In a distributed environment it may be less desirable to rely upon a single coordinator if it is reasonable for elements of the simulation to become disconnected, in which case certain *rendezvous* points may be established whereat communication between different substructures (coupled models) becomes possible. For example, it may be that top level coupled models *A* and *B* only ever communicate upon a periodic schedule. In this case it may be appropriate for *A* and *B* to coordinate their own events and not rely upon a global coordinator except for the periodic communications. Thus, it can be seen that several different forms of time and event coordination are possible within a simulation environment depending on scale and distribution. Further, many issues common to transactional databases, such as optimistic versus pessimistic locking, also have applicability during simulation. Different sub-models may be allowed to become temporarily out of synchronization with respect to time, and then latter need to communicate with one another. In an optimistic scenario, we would allow separate sub-models to operate independently, so long as communication between them is not required, and then block, waiting for both sub-models to comes back in sync. In *Clock Cycle ~ Scenario II*, we present a more involved scenario whereby a coupled model may acknowledge a tick and still continue to receive events before the arrival of the next tick. This is due to the fact that all the events that occur *at the same time* are not always available from a simulation perspective at the same time. That is, from a simulation perspective, there may be multiple iterations of message passing during a single clock cycle, because any time taken to deliver a message must be accounted for explicitly within the model and <u>not</u> in the simulation engine – within the simulation engine all message-passing takes *zero time*.

Another point to note is that although a coupled model may acknowledge the tick event, it may still receive events from its container in the same clock cycle – it should be quiescent with regards to any activity within the models contained within it. The algorithm is quite simple and does not require differentiating between time events, *ticks*, or non-time events, *application events*. Each model must acknowledge the receipt of a message regardless of type, but it can only acknowledge receipt of the message when all its sub-component models have acknowledged receipt of events passed to them. In the event that a sub-model generates additional events that are relayed by the coupled model to other peer sub-models, these events must also be acknowledged before the original initial event is acknowledged. All the while, the coupled model maintains a counter of events that it generated and sent to its container. When the acknowledgement is finally sent, this count of events signaled is also forwarded, and serves as a checksum for the parent coupled model ensuring that it has indeed received and processed all the events generated by the child model.

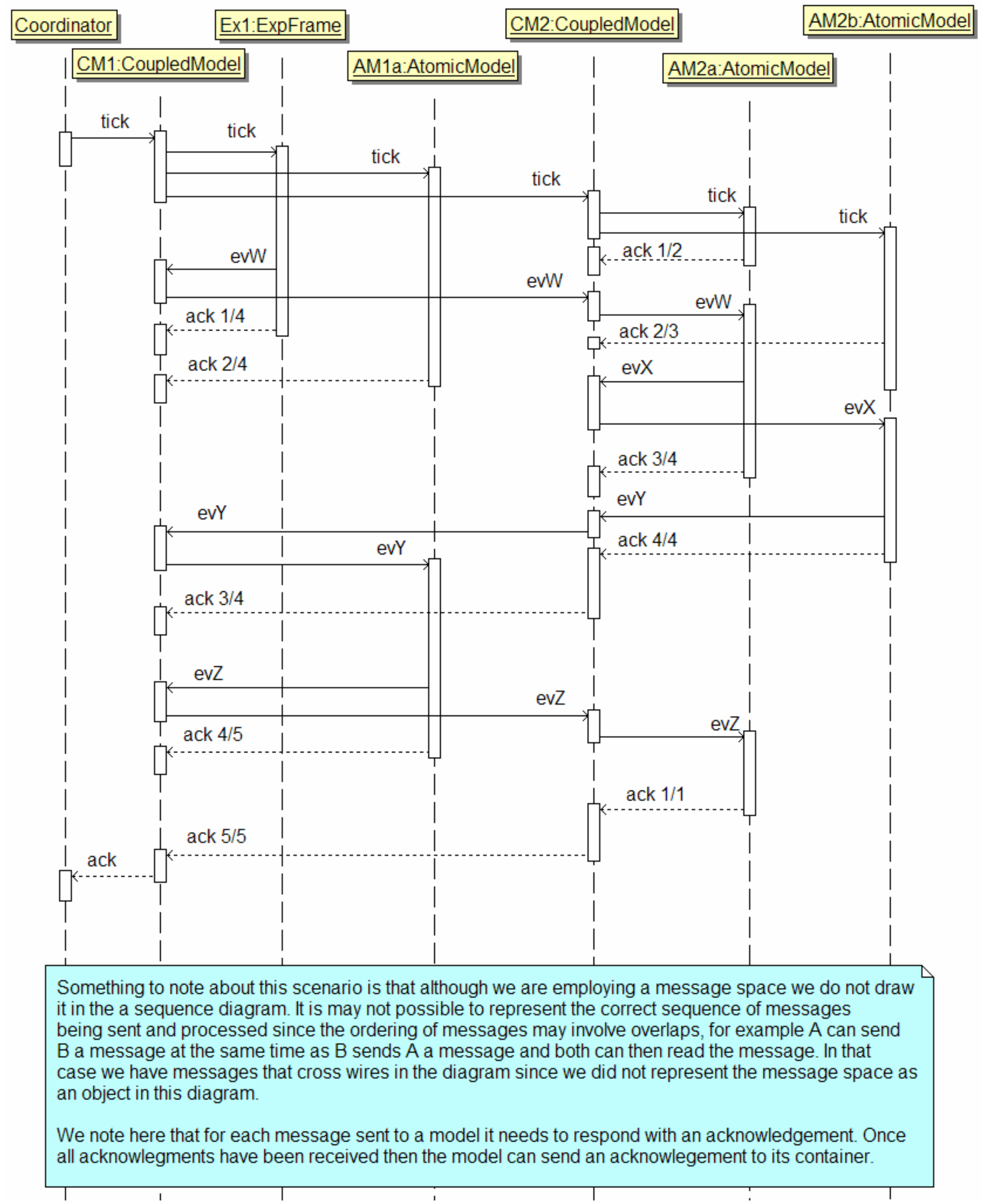

Figure 35 Clock Cycle ~ Scenario II

In both scenarios presented thus far we have not dealt with the bundling of messages so that all messages that occur at a given time are presented together and not as separate events.

A model never knows when it has received its last message within a given clock cycle. Not until it receives the next *evTick* event signaling the beginning of the next clock cycle do we know the previous clock cycle has terminated. It would be nice if the model knew that it would receive no more messages during a clock cycle, prior to its *output function.* If this were possible, the next state would be a simple determination of the current contents of the message bag. However, this is not possible since a model has no control over its inputs, and as such, other messages can arrive during the same clock cycle (but in a later simulation cycle). Therefore, the internal transition scheduled as a result of the messages received is subject to change, since other messages may yet arrive during the same clock cycle. Imagine that if a prime number of messages is received, then we would transition to state s_1, and for a non-prime number of messages, we transition to state s_2. To model this we would require a counter that is initialized to zero during simulation set-up and reset to zero upon each *evInternalTx* event. For each external message bag receipt, we would increment the count by the number of messages in the bag, provisionally set the next state depending on the current count, and signal an *evSigma* of 'one'. We say *provisionally* since out intention is to transition to this state unless other messages arrive; if another message arrives, we recalculate the next state. Upon *evInternalTx,* the output function would then generate an output based on the current count value, and reset the count back to zero. The confluent function would specify that the *evInternalTx* takes place before processing external events, ensuring *evSigma* expiration would not interfere with the count.

It would be nice to deliver all messages that occur in a given clock cycle all at once and to the extent that there are no *evSigma(0)* signals <u>anywhere</u> in the system during the clock cycle then this is possible. Where this condition hold true all outputs are generated in the first simulation cycle of the clock cycle and all these messages are delivered during the second simulation cycle of the clock cycle.

By default, any atomic models that have an expiring sigma, and thus an imminent internal transition, receive that expiration notification simultaneously (during the first simulation cycle of the clock cycle). Since outputs are delivered as part of the next simulation cycle, the notion of a *confluent function* becomes a non-issue, except where a model issues an *evSigma(0)* during a simulation cycle, and that model is also sent messages during that same simulation cycle by another model. In such a case, there are confluent internal and external events. Confluent events are simultaneous events that occur during the same simulation cycle. Confluent events are not simultaneous events occurring during *different* simulation cycles during the same clock cycle. Where models are specified such that external events are given precedence over internal transitions, it may be appropriate to trigger any internal transition after any scheduled internal transitions for other models, that is, during the second simulation cycle. This becomes problematic when there are multiple such models since there is no way to resolve certain situations. Consider model M_1 and M_2 that are bi-directionally connected. Both could have an internal transition at time t_n that results in a message being sent to the other model – for both models these are confluent events: the expiration of *evSigma,* and the receipt of the message from the other model, occur during the same simulation cycle. There is no way to solve this chicken or egg problem other than to have an internal transition during the first simulation cycle and to have the message from the other model arrive during the next simulation cycle – in which case neither model has any confluent events.

Registration

Before any simulation can commence all the various models must register. This is a bottom-up procedure whereby the atomic models register with their coupled model and once all atomic models have registered for a particular coupled model then it can proceed to register with its containing coupled model and so on until registration is complete for the outermost coupled model (a.k.a *coordinator*). At this point, the coordinator can communicate to all of its coupled models that they can now start, meaning that they can prepare to start accepting messages as part of a simulation execution.

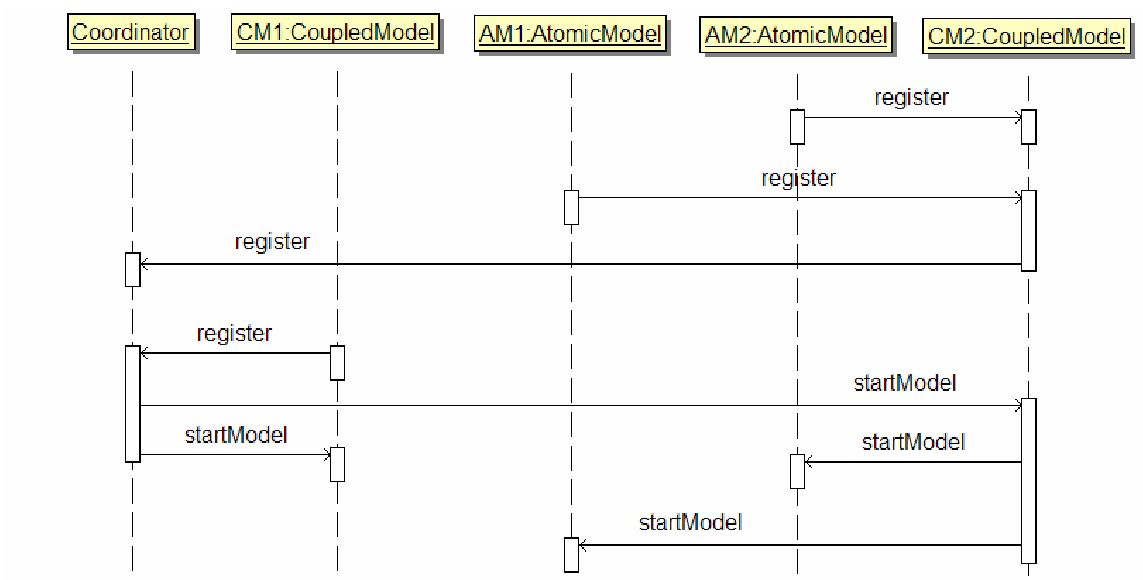

Figure 36 Static Registration

Dynamic Registration

In the prototype developed for this thesis, an alternative form of dynamic registration was available called *dynamic registration*. Within this registration protocol the coupled models were essentially unaware of the structure of the models that they contained and during the registration phase the models informed their respective coupled model as to the input and output ports they contain (and potentially other information that may be of interest such as timing and possible messaging filtering).

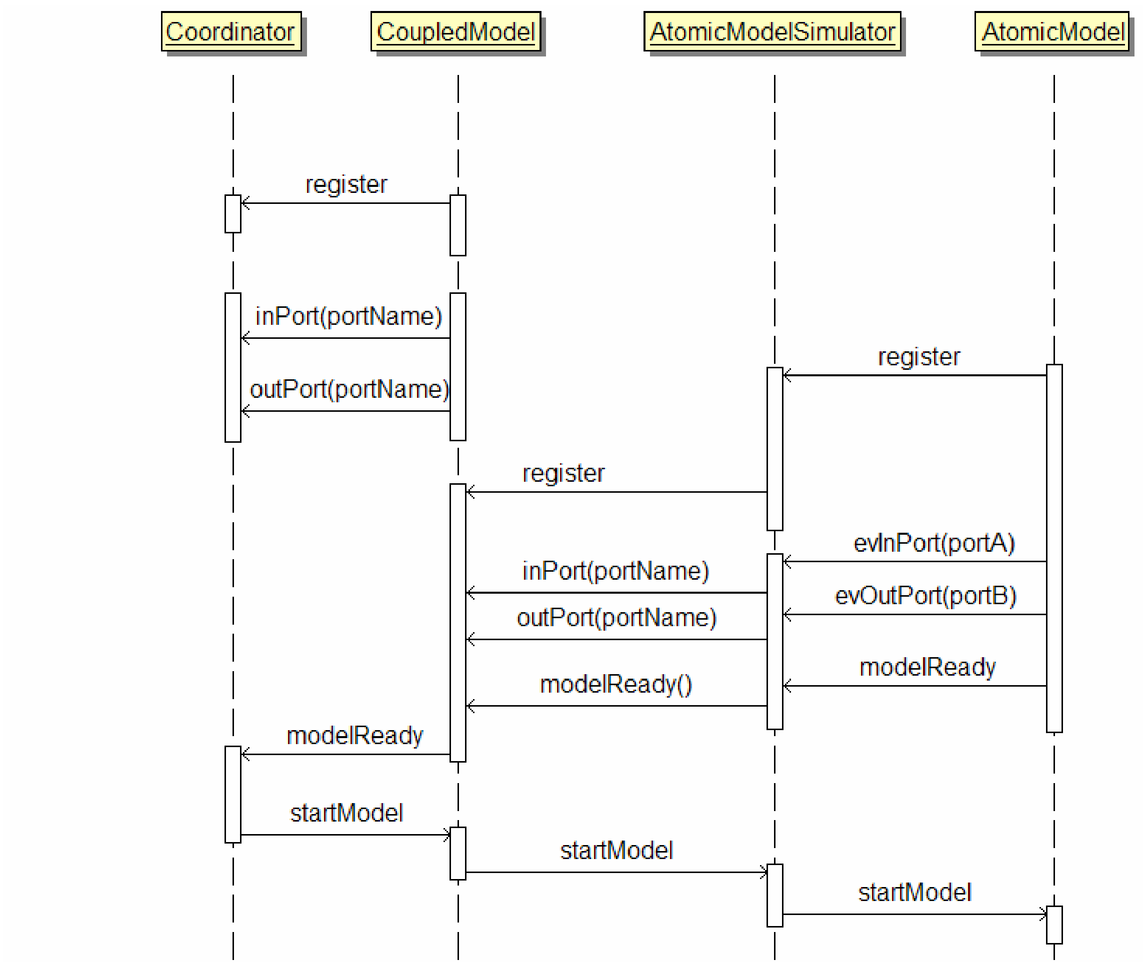

Figure 37 Dynamic Registration

Connecting Model Ports

Hitherto, we have glossed over how outputs from these state machines are made available. In DEVS components (models) have input and output ports. Output ports from one component can be connected to the inputs of another peer component or to the output ports of a containing component. Likewise input ports of a containing component connect to the input ports of sub-components or input ports can connect to output ports of a peer component. In DEVS, any component that contains other components is called a coupled model and all behavior is derived from these sub-components – there is no behavior specified at the container/coupled model level. It should be noted that coupled models blur the line between class and run-time object since sub-components are essentially instances of their respective model. The coupled model is simply a runtime container of its sub-components. In UML, a Structured Class is a rough analogue to a DEVS coupled model though it has the capability to have its own responsibilities beyond being a simple container and its ports are bi-directional. If we use a Structured Class to represent a DEVS coupled model, ports must be uni-directional, and there can be no connections from a part back to itself. Additionally, corresponding to each port is an event signal type.

For state machines, we map DEVS input ports to events and output ports to event signal generation. It is envisioned that atomic (including experiments) and coupled models can join a system dynamically at runtime. Further, in order to simplify the specification of how models are connected to each other, instead of specifying this at the level of the coupled model, a set of conventions are employed so that in the event of dynamic registration a complete mapping of the ports of the model being registered need not be provided to the coupled model. During registration the input/output port names defined in the model's specification can be mapped to different port names, that is, the actual port names for an atomic model may not make sense for the particular coupled model and a mapping is provided during registration.

By convention, where port names match between a coupled model and a sub-model, a sub-model may be registered without supplying a port-to-port name mapping. Where port names do not match, or conflict, we must provide a mapping from atomic model port names to the port names that are currently defined in coupled model instance. In the event of output ports remaining unconnected at the end of initial registration, they are connected to a null output port for the coupled model, meaning that their outputs are discarded.

C H A P T E R 5

DEVS/UML CONTRACT

Under DEVS/UML, an instance of a class may participate in a model, if and only if, it is contained within an instance of *simulatable object* or is itself an instance of a *simulatable object*.

A *simulatable object* is one that has its behavior defined via a *compliant state machine*. This implies that all communication is essentially asynchronous insofar as replies to messages require an internal transition before a response is available. However, since *evSigma* can be specified with zero time delay, the distinction is moot.

A *compliant state machine* has the following characteristics:

- All time dependent behavior is expressed via *evSigma* signals and *evInternalTx* events. There shall be no *after* or other UML time related references in the state machine specification. The *evInternalTx* event may only be generated by the *atomic model simulator*.
- Each atomic model state machine has an associated *atomic model simulator*.
- All communication (signal generation) beyond the boundary of the *simulatable object* occurs only <u>upon</u> receipt of an *evInternalTx* event and occurs before the *evInternalTx* transition completes – this is the *output phase*. Such activity is defined in the action part of the specification of the transition triggered by the *evInternalTx* event.
- An *evAck* signal must be generated and sent to the atomic model simulator upon completion of processing of an external event. All processing from receipt of the external event through the generation of the *evAck* signal must be atomic – it must run to completion.

- All internal communication (signal generation/method calls) occurs only in response to an external event (including *evInternalTx*). The state machine must be dormant (*quiescent*) after the *evAck* signal is issued.
- An *evSigma* event signal is generated and sent to the *atomic model simulator* whenever the state machine wishes to simulate the amount of time required to complete some hypothetical processing, transmission time, or other such delay. This event is sent to the associated *atomic simulator* object. The *atomic simulator* will send an *evInternalTx* event back to the state machine upon expiration of this time. The state machine remains dormant (*quiescent*) during this period or until it receives another non-*evInternalTx* external event.
- If an *evSigma* event signal is generated, it <u>must immediately precede</u> the *evAck* signal generation.
- Upon receipt of an external event, an atomic model may issue a new *evSigma* signal which supersedes any previously generated *evSigma* signal event. Otherwise, any existing *evSigma* will remain in effect. A duration of *-1* in an *evSigma* signal indicates infinity or *passive* state. As such no *evInternalTx* event will occur and the state machine will be dormant until the next external event.
- All processing time involved in handling events during simulation is performed in *zero time* unless <u>explicitly accounted</u> for via *evSigma* signals.
- All messages (event signals) are transmitted and received in *zero time* – the simulation clock is stopped. Likewise, all logic performed in the state machine is performed in

zero time. Any requirement to model this time must be explicitly accounted for in the model via *evSigma* signals.

Caveats

As with any modeling there is an art to the choices one makes and as such no set of rules will guide a modeler to a wise solution. Only through experience will one become adept at making the right choices with regards to the appropriate level of abstraction of a given model, the naming of ports or events, and the types of experimentation necessary to demonstrate and validate the design. However, the beauty of modeling at the level defined by a DEVS model is that the temptation to delve into implementation decisions and create an elaborate model is tempered by the usefulness of simulation as a tool to aid all stakeholders as to the problems at hand and the proposed solutions intended. Since such simulations are possible with relatively trivial models, this encourages an iterative approach to model development and stakeholder validation.

CHAPTER 6

RELATED WORK

Mappings between UML and DEVS

Several others have approached this subject from various perspectives. Similar observations are presented with regard to the desirability of performing modeling and simulation early in system design and that a combination of DEVS and UML represent an appropriate means achieving this goal (Risco-Martin et al. 2007). The authors outline an approach whereby a UML state machine is translated into an equivalent State Chart XML (SCXML) representation. The SCXML is, in turn, converted into a finite deterministic DEVS state machine model. The DEVS state machine model gets translated into an equivalent FD-DEVS XML model which can then be executed by a DEVS simulation engine. The authors do not address the specifics of how issues that are the domain of a simulation engine in the DEVS world, such as dealing with simultaneous events, confluent events, and transmission of events to multiple destinations are handled. This thesis diverges from their approach insofar as it is recommended that UML state machine models employ DEVS specific events (*evInternalTx*) and event signals (*evSigma*) in preference to using a UML *after time-spec/action-spec*. This thesis recommends that the state machine for each atomic model sends and receives messages via a corresponding *atomic model simulator* state machine. Alternatively, it may be possible to use a single state machine, partitioning the simulation specific logic into separate orthogonal region within the state machine. Each type of atomic model should extend the base atomic model state machine. This thesis argues that in a UML world issues that are properly the domain of a simulation engine are best segmented out into separate state machines (*atomic model simulators*) so as to ensure that the specification of state machines for atomic models remain as simple as possible. This allows for their easy deployment as reusable components within a given system architecture. As a practical example, a state machine, thus defined, is completely reusable regardless of the coupled models in which it is are employed. By following a simple set of rules, we can be assured that verification and validation of the model is not compromised. Where and how instances of an atomic model are employed is of little concern during its design. There are no references to the external environment in an atomic model specification except the external events it receives. An atomic model is agnostic with respect to the specific source of such events.

A formal mapping from DEVS to UML is presented (Zinoviev 2005). Within this mapping input and output ports are mapped to UML events as is the case with this thesis. Also, input and output ports are likewise assumed to be disjoint. DEVS state variables that are non-continuous are mapped to UML states, and DEVS state variables that are continuous are mapped to attributes of UML states. At a formal level, this makes sense for mapping between both specifications but in practical terms it may be inappropriate to represent every non-continuous variable with its own state. Zinoviev (2005) also employs a combination of a special timeout event and use *after* events for handling internal transitions. The special timeout event has a guard condition introduced to protect against timeout events that have been made obsolete by an intervening external transition. Overall, the mapping presented by Zinoviev (2005) is elegant and avoids the infrastructural complexity of the global clock, global coordinator, and atomic model simulator presented in this thesis. However, from a simulation perspective there is a significant problem whenever the UML *after* event is employed: the act of simulating a model and the associated processing time is not accounted for and will interfere with simulation itself; this time must never be commingled. In short, we cannot use *after* if the referenced time is coincident with the time involved in enabling the simulation which will inevitably be the case unless this time is centrally coordinated and accounted for. Further, when using after, synchronized behavior among models cannot be guaranteed because the time cost of simulation is left unaccounted. Remember, within a simulation, time passes only as accounted for by the *evSigma*. The act of setting state variables, performing transitions, generating output etc. all occurs in *zero time*. This simulation specific overhead cannot be reliably accounted for via the *after* function.

An informal mapping from DEVS to an equivalent STATETMATE Statechart is presented (Schulz et al. 2000). In a somewhat similar fashion they account for time in an orthogonal region of the state machine. However, the proposal does not introduce a global clock which is necessary for synchronicity among models. Schulz et al note that DEVS has greater expressive capabilities than state charts (Harel and Naamad 1996) and that any DEVS model can be represented via STATEMATE Activity Charts and an appropriate naming convention for events.

UML Diagrammatic Representations of DEVS Models

We are presented with the specifics of how one may represent DEVS models diagrammatically using UML-RT (Huang and Sarjoughian 2004). However, the authors conclude that UML-RT is not suitable for a simulation environment, and they assert that the design of software and simulation is inherently distinct. As such, both modeling frameworks should co-exist without the need to extend one or the other and thereby compromise the respective formalisms and applicability. In a sense, a somewhat similar conclusion was arrived at during the research for this thesis: UML-RT is not an especially suitable framework for the expression of DEVS models specifically because it represents an over-engineered solution for what in DEVS is an elegant expression of a simulation specification.

Model-driven Architecture

Model-driven architecture is very much a buzzword at the moment. Certainly, the notion of models driving architecture is central to DEVS and DEVS/UML. Ideally, work that is today spent writing implementation code in Java or C++ could be eliminated if models are specified to a sufficient level of detail to allow complete code generation. Generally, code generation alone is insufficient and much code must still be handwritten. Quiet often the model becomes out of sync with respect to the code.

DEVS/UML can be considered a *model compiler* that produces an executable model. In many ways DEVS/UML represents a more attainable form of MDA since the scope of the problem is vastly reduced. Both platform specific and platform independent models can be created so that DEVS models demonstrate what is theoretically possible versus what is practically possible. Modeling in a DEVS/UML environment can mirror that of the transformations central to OMG MDA (OMG 2003). Douglass (1999, 2004) has been actively involved in advancing model driven architecture specifically in the area of real time systems. It is worth noting his observation that "simulation has its place, but the purpose of building and testing executable models is to quickly develop defect-free applications…" Douglass (2004, 35). This thinking reveals a lack of appreciation of simulation as a worthy element of systems development.

Dynamic Languages/Domain Specific Languages

Despite the difficulties that working with JRuby presented during the construction of the prototype for this thesis, dynamic languages offer several significant advantages over traditional compiled languages such as Java and C++. First, the ability to execute code on the fly is very useful in a prototype or simulation environment. Dynamic languages such as Smalltalk, Ruby, Python, and Groovy, through the availability of the interpreter, are more readily capable of defining domain specific languages wherein custom languages created using the verbs and nouns common to the domain being modeled can be defined with relative ease. Such domain specific languages may be then used for model specification. Java and C++ are generally not suitably for the creation of lightweight domain specific languages.

Validation & Verification

In their paper, *A Compositional Approach to State Machines Semantics* (Luttgen et al. 2000), the authors observe that classical state charts system behavior cannot be deduced from the behavior of their subsystems, and this is a serious weakness in state charts that impede employing them in a compositional fashion. They introduce a global clock and tick events in order to bound macro steps in a similar manner to how they are presented in this thesis. They contend that such modifications to a state machine enable validation and verification which is not otherwise possible. Each macro step is defined as a subset of transitions that are causally well-founded such that each of the respective transitions is enabled by the state machine at the point of that transition. They refer to the necessity of synchrony wherein events generated in one macro step are consumed by the same step and not the next step. They point out that STATEMATE (Harel and Naamad 1996) does not uphold synchrony.

Future Scope

One area not covered in the thesis is the generation and/or validation of other UML artifacts such as sequence diagrams during simulation. Another area is the incorporation of different design patterns into a design and then using simulation to determine their relative suitability. With regards to simulation engine implementation, Tuple Spaces represent an exciting area in which potential performance improvements and other distributed topologies may be possible. Some research has already taken place in this area (Teo 2003).

Another area of interest is the use of design patterns during the creation of simulatable models. Design patterns have already been shown to have applicability at the domain level when creating DEVS models (Ferayorni and Sarjoughian 2007). This thesis suggests that there is additional scope for employing design patterns at the UML level and in a more non-domain or generic manner when creating simulatable UML models.

CHAPTER 7

CONCLUSIONS

This thesis proposes a new approach to architecting software systems using the UML wherein formal simulatable models are the first executable artifacts. A set of rules is specified for a UML practitioner to follow to ensure that the models generated are simulatable based on the DEVS formalism. This approach should help mitigate some of the objections to employing simulation during the early development of a software system. Such simulatable models implicitly adhere to the DEVS formalism. The objective is that those unfamiliar with DEVS and more comfortable with the UML have a convenient and relatively straightforward mechanism by which their UML models can be executed and verified at an early stage of design. Since other formalisms such as those employing discrete time and differential equations can be represented using the DEVS formalism, it means that this approach has a wide generality in terms of the types of systems and problems to which it can be applied. Also, since the models produced are component-based, and given the closure property of DEVS, any component can be replaced by a different component with a greater degree of decomposition, and the resulting system will have an equivalent behavior. This modeling technique fits neatly within the modern iterative approach used to develop software systems. Finally, it is anticipated that this, almost seamless, integration of simulation during the architecture of a system will help broaden and deepen the appreciation and application of simulation as a discipline within the field of software architecture.

REFERENCES

ACIMS. 2007. *DEVSJAVA*. http://www.acims.arizona.edu/SOFTWARE/software.shtml.

Borland, S. 2003. Transforming statechart models to DEVS, McGill University, Montreal.

Choi, K., D. Bae, and T. Kim. 2006. An approach to a hybrid software process simulation using the DEVS formalism. *Software Process: Improvement and Practice* 11 (4):373-383.

Chow, A. 1996. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International* 13 (2):55-67.

Crane, M., and J. Dingel. 2005. UML vs. classical vs. rhapsody statecharts: Not all models are created equal. *Software and Systems Modeling* 6 (4):415-435.

Douglass, Bruce P. 1999. *Doing hard time: Developing real-time systems with UML, objects, frameworks, and patterns*. Boston: Addison-Wesley Professional.

———. 2004. *Real-time UML: Developing efficient objects for embedded systems*. 3rd ed. Boston: Addison-Wesley Longman Publishing Co., Inc.

El Sheik, A., A. Al Ajeeli, and E. Abu-Taieh. 2008. *Simulation and modeling: Current technologies and applications*. New York: IGI Publishing.

Ferayorni, A., and H. Sarjoughian. 2007. Domain driven modeling for simulation of software architectures. In *Summer Computer Simulation Conference*. San Diego, CA.

Gelernter, D., and N. Carriero. 1992. Coordination languages and their significance. *Communications of the ACM* 35 (2):97-107.

Harel, D., and A. Naamad. 1996. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology* 5 (4):296-333.

Huang, D., and H. Sarjoughian. 2004. Software and simulation modeling for real-time software-intensive systems. Paper read at Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications, at Washington, DC.

Huang, Y., M. Jeng, and C. Hsu. 2004. Modeling a discrete event system using statecharts. *IEEE International Conference on Networking, Sensing and Control* 2:1093-1098.

JRuby. 2007. *JRuby Programming Language*. http://jruby.codehaus.org/

Kofman, E., M. Lapadula, and E. Pagliero. 2003. PowerDEVS: A DEVS–based environment for hybrid system modeling and simulation. *Technical Report LSD0306*.

Luttgen, G., M. von der Beeck, and R. Cleaveland. 2000. A compositional approach to statecharts semantics. *ACM SIGSOFT Software Engineering Notes* 25 (6):120-129.

Mellor, S. J., and M. J. Balcer. 2002. *Executable UML - A foundation for model-driven architecture*. Boston: Addison-Wesley Longman Publishing Co., Inc.

OMG. 2003. *MDA Guide*. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf.

OMG. 2005. *UML 2.0 Superstructure Specification*. http://www.omg.org/cgi-bin/doc?formal/2005-01-02

OMG. 2007. *UML Profile for Schedulability, Performance, and Time*. http://www.omg.org/cgi-bin/doc?formal/07-02-03.

PAVG. 1998. *Rapide*. http://pavg.stanford.edu/rapide/.

Risco-Martin, J. L., S. Mittal, B. Zeigler, and J. de la Cruz. 2007. From UML state charts to DEVS state machines using XML. In *IEEE/ACM International Conference on Model-Driven Engineering Languages and Systems*. Nashville, TN.

Ruby. 2007. *Ruby Programming Language*. http://ruby-lang.org.

Samek, M. 2002. *Practical statecharts in C/C++: Quantum programming for embedded systems*. Manhasset, NY: CMP Publications, Inc.

Schulz, S., T. C. Ewing, and J. W. Rozenblit. 2000. Discrete event system specification (DEVS) and statemate statecharts equivalence for embedded systems modeling In *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*. Edinburgh.

Sun Microsystems. 2007. *Jini Network Technology*. http://www.sun.com/software/jini.

Teo, Y. M., and Y. K. Ng. 2002. SPaDES/Java: Object-oriented parallel discrete-event simulation. *Proceedings on the 35th Annual Simulation Symposium*:245-252.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems*. 2nd ed. San Diego: Academic Press.

Zinoviev, D. 2005. Mapping DEVS models onto UML models. *Proceedings of the 2005 DEVS Integrative M&S Symposium*:101-106.