Transforming Statechart Models to DEVS

Spencer Borland
Supervisor: Hans Vangheluwe

August, 2003

School of Computer Science
McGill University, Montréal, Canada

A thesis submitted to McGill University in partial fulfilment
of the requirements of the degree of Master of Science.

Abstract - Rèsume

The Statechart formalism is explored. The meta-modeling environment, AToM^3, is examined. Statecharts are meta-modelled using the AToM^3 tool. The classical, hierarchical DEVS formalism is also explained in detail. A simulation package, `pythonDEVS` is used to simulate DEVS models. Small modifications to the `pythonDEVS` package yields a real-time DEVS executor, `pythonDEVS-RT`. From the Statechart meta-model, a Statechart modeling environment is generated. The mapping of Statechart models onto behaviourally equivalent DEVS models is presented. Statecharts are mapped onto DEVS. A code generator is built which automates the Statecharts to DEVS translation. The resulting models can be executed in real-time using the `pythonDEVS-RT` package.

Le formalisme de Statechart est exploré. L'environnement de méta-modellisation, AToM^3, est examiné. Statecharts sont méta-modelés l'aide de l'outil d'AToM^3. Le formalisme classique et hiérarchique de DEVS est également expliqué en détail. Un paquet de simulation, `pythonDEVS` est employé pour simuler des modéles de DEVS. Les petites modifications au paquet de `pythonDEVS` rapporte un exécuteur en temps réel de DEVS, `pythonDEVS-RT`. Les crée de méta-modéle de Statechart un Statechart modelant l'environnement. Statecharts sont tracés sur DEVS. On présente un générateur de code qui automatise le Statecharts la traduction de DEVS. Les modles résultants peuvent être exécutés en temps réel en utilisant `pythonDEVS-RT` le paquet.

Acknowledgements

I would like to thank Professor Hans Vangheluwe for all his ideas and vast knowledge. Thanks to Ernesto for always getting me out of a pickle and for knowing something about everything. Thanks to Jean-Sébastien for his work on DEVS. Thanks to all those in the MSDL who, at one point, helped with something or another.

And a special thanks to George W. Bush (a.k.a., ”W”, pronounced ”double-ya”) who is quoted as saying,

> ”We ought to make the pie higher.”
> -South Carolina Republican Debate, Feb. 15, 2000

Contents

List of Figures

List of Tables

1 The Statechart Formalism

1.1 Introduction

Statecharts have become a popular visual formalism in recent years. They are challenging finite state automata as the most widely used language for describing reactive systems. In fact, statecharts can represent complex behaviour much more compactly than their predecessor, finite state automata. Statecharts present an elegant solution to the problem of state explosion when flattening concurrent states. They also model the hierarchy of information in their specifications, in an intuitive way. Statecharts have been used for a wide range of fields from graphical user interface (GUI) design to embedded systems.

Statecharts were built for specification of systems under the event oriented paradigm. Event-driven systems are everywhere and it is these systems such as telephones, stock exchanges and computer networks to name a few, that statecharts elegantly represent.

When linked with code generation, statecharts can be very useful in the software engineering world as applications can be developed rapidly, with a high level of modularity. Code generation from a statechart specification is a rich subject as there are many implementation details to be decided upon.

In order to understand the basics of statecharts, it is useful to first look at higraphs. Higraphs, while normally used for formal mathematical specification, are also used as a basis for statecharts.

1.2 Higraph Basics

Visualizing information concisely is the motivation behind higraphs. It is important to note that the style of information that is to be represented is relational or topological (as opposed to geometrical). The shapes of the visual objects used are not important, rather, it is the relationships between objects that is important. These constructs are termed *topiovisual* [33].

Euler was the first to create such a topiovisual construct. He developed the well known visual formalism, *graphs*. Graphs are capable of defining elements of a set, S with a binary relation, R using vertices (small circles) connected by edges (lines) [28]. The field of graphs is huge and there are many different types of graphs created by different types of relations. It seems this would be a useful feature for describing information relationships.

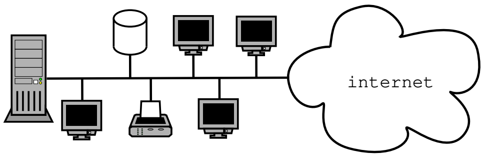

Figure 1.1: A hypergraph.

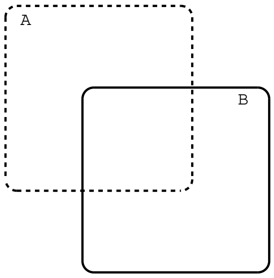

Figure 1.2: A higraph.

Another topiovisual formalism is Venn diagrams (an extension of *Euler circles*), developed by John Venn (1834-1923). Venn diagrams are a very intuitive way of describing the binary set operators, union and intersection [28]. This also seems like a useful notion for representing information.

If the goal is to create a topological representation of a microcircuit where the edges are the *wires* and the vertices are *modules* using a graph, there exists a problem. The problem is that the relation, R, is binary while microcircuits do not necessarily have edges which simply connect two modules. What is needed is a type of graph which utilizes a relation where the cardinality of their operands may be greater than one. This is accounted for in the *hypergraph* construct. An example hypergraph can be seen in Figure 1.1.

Now the previous ideas can be extended to create the Higraph [28] construct. First, Venn Diagrams are used to account for set inclusion and exclusion. Each set is called a *blob* and is denoted using a rectangle with rounded edges. For the purposes of labeling, it is required that each intersection explicitly contains another blob. Looking at Figure 1.2 it is evident that having such a requirement is necessary. Since all the labels for a set are contained with a blob, it is unclear whether A denotes the entire blob with the dotted edge or (the blob with the dashed edge) - (the blob with the solid edge). In Figure 1.3, A and B and C are uniquely defined where $C = A - B, D = A \cap B, E = B - A$. Using this method the entire topography is flattened so an *atomic* blob (a blob that has no sub blobs) is obtained for every set or subset to be referenced. Another possibility is to draw overlapping

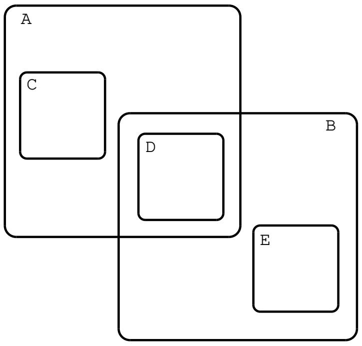

Figure 1.3: A higraph with hierarchy.

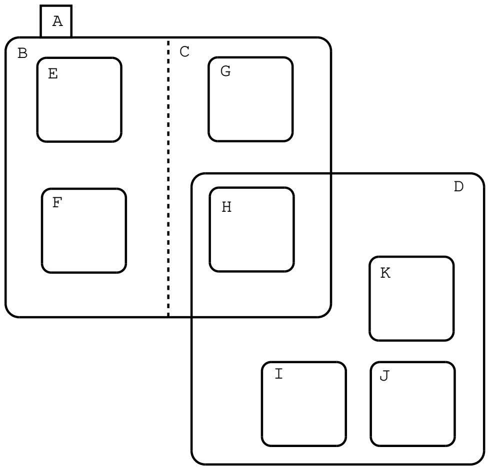

Figure 1.4: A higraph with orthogonal components.

blobs, A and B, without explicitly adding a blob in the intersection region. Such a construct is fairly useless since no inference can be made about the regions $A - B$, $A \cap B$ and $B - A$.

Next, the notion of the Cartesian product is used to deal with concurrent parts of the system being designed. The ability to split any blob into sections using a dotted line is added. These sections, or *orthogonal components* represent the Cartesian product of the blob containing them. For example, in Figure 1.4, $A = B \times C$. Since $B = E \cup F$ and $C = G \cup H$, then $A = (E \cup F) \times (G \cup H)$. B and C are called the *orthogonal components* of A. Note that saying $A = B \times C$ is the same as saying $A = C \times B$. Thus, the Cartesian product operator $\times$ is not entirely correct since A is really an unordered set. Thus, it should be written using the unordered Cartisian product operator as $A = B \otimes C$, but $\times$ will be notationally equivalent.

In fact, higraphs are defined by Harel as a 4-tuple as follows:

$$H = (B, \sigma, \pi, E)$$

B is the set of unique blobs. E is the set of all hyper edges. Note that E can also be defined as:

$$E \subseteq 2^B \times 2^B$$

σ is the sub-blob or hierarchy function. It takes a blob, X and returns the set of sub-blobs within X:

$$\sigma : B \to 2^B$$

Also, σ should never define a cycle. That is, if x is a descendant of X, then $x \notin \sigma_+(x)$ with,

$$\sigma_0(x) = x$$

$$\sigma_{i+1}(x) = \cup_{y \in \sigma_i(x)} \sigma(y)$$

$$\sigma_+(x) = \cup_{i=1}^{\infty} \sigma_i(x)$$

The partitioning function (the function which specifies how a blob is broken up into its orthogonal components) is defined as:

$$\pi : B \rightarrow 2^{B \times B}$$

Note that π is an equivalence relation and it specifies partitions of a blob, each of which are equivalence classes.

This concludes a definition of higraph syntax. To know more about the semantics of the higraph construct, some definitions are given. The set of atomic blobs, A is defined as:

$$A = \{x \in B \mid \sigma(x) = \emptyset\}$$

Note that since there exists a criterion on σ which ensures that cycles cannot occur, A will not be empty. The unordered Cartesian product of two sets is written as:

$$S \otimes T = \{s, t \mid s \in S, t \in T\}$$

Now the model of a higraph is defined. It is another way of writing the higraph formally for purposes of understanding its structure. The model, M, is a 2-tuple:

$$M = (D, \mu)$$

D is called the domain of the model, and it is simply a set of unstructured elements. It is defined to provide a mapping between the atomic blobs and D's power set. This mapping is defined by μ as follows:

$$\mu : A \rightarrow 2^{D}$$

Note that $\mu(x) \cap \mu(y) = \emptyset (if x \neq y)$. Next the unordered Cartesian product is used to define a more complex construct containing orthogonal components. For each $x \in B$,

$$\mu(x) = \otimes_{i=1}^{k_x} (\cup_{y \in \pi_i(x)} \mu(y))$$

The semantics of the edges must also be defined. A relation called the E_M is used as follows:

$$(\mu(x), \mu(y)) \in E_M \Longleftrightarrow (x, y) \in E$$

1.3 Higraph Applications

Higraphs are now a well-defined visual formalism and can be used for many methods of formal specification. To see the power of Higraphs, look at Figure 1.5. The constraint of specifying each edge in a 5-clique is relaxed. The Higraph representation is much more compact.

Higraphs can also be used to replace entity-relationship diagrams. Figure 1.6 shows explicitly an entity relationship diagram describing various real world objects and their relation to one another.

In Figure 1.7, the entity relationship diagram is represented as a higraph. It is important to note that sub-blobs of the statechart version have an implicit *relationship* with their parent contour. This implicit relationship often takes on meanings such as *is-contained-in* or *is-type-of*.

Higraphs are also a useful basis for statecharts. Next, the statecharts formalism is introduced.

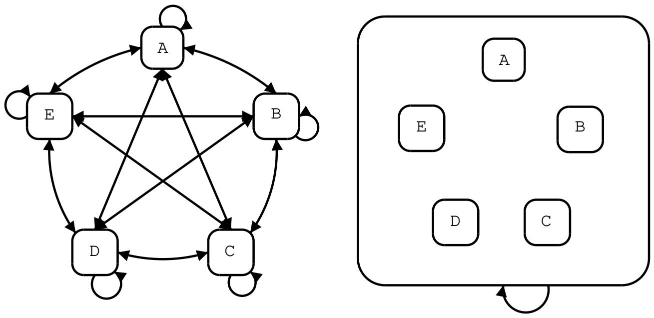

Figure 1.5: The clique, fully connected semantics.

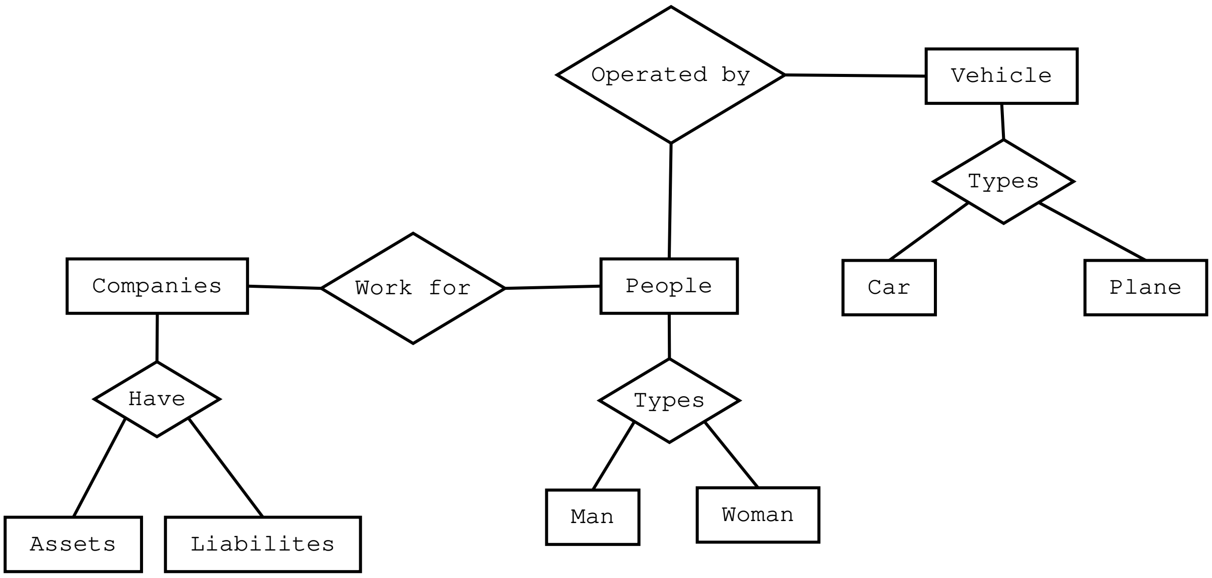

Figure 1.6: A standard ER diagram.

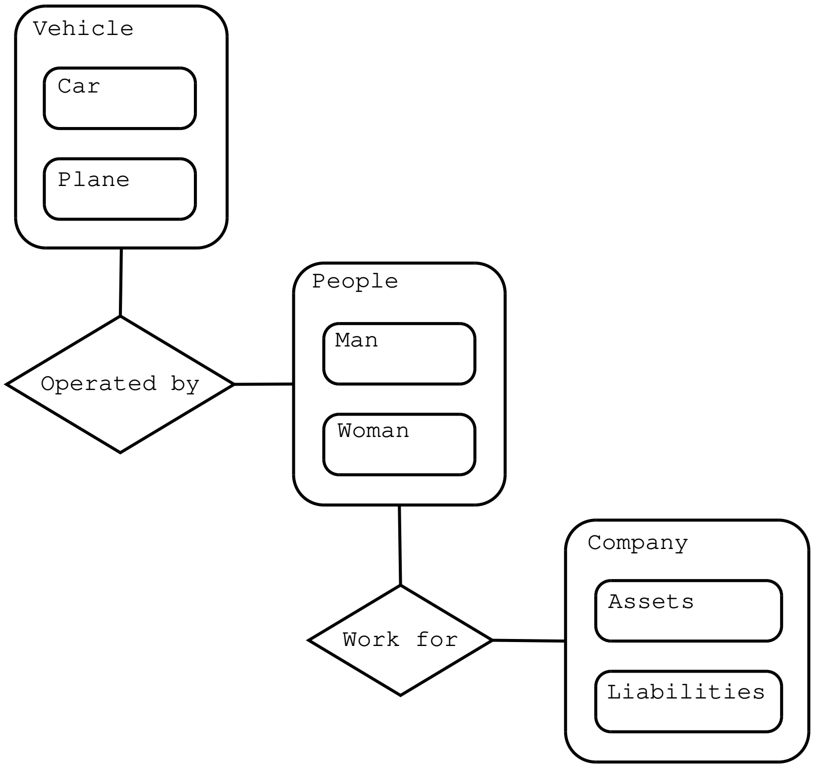

Figure 1.7: A higraph as a statechart.

1.4 The Basics of Statecharts

The most basic elements in statecharts are *blobs*. Statecharts are a visual formalism and a blob is usually represented by a variable sized rectangle with rounded edges. Blobs have a similar meaning to *states* in finite state automata. However, there are different kinds of blobs as well. This differs from the FSA specification in which there is essentially only one kind of *state*.

These different kinds of states are called OR-blobs, AND-blobs and BASIC-blobs. These different types of blobs give rise to hierarchy in statecharts. It makes it easy to see that when a certain state is reached, some details may be left out. However, these details can be specified using OR-blobs.

More specifically, a BASIC-blob is a state such that it has no sub-OR-blobs, sub-AND-blobs nor any sub-BASIC-blobs. OR-blobs are blobs that have one or more sub-OR-blobs. AND-blobs have, what are called orthogonal components. The orthogonal components of a blob are normally drawn with a dashed line and denote concurrent operations. These definitions are easily seen with a picture in Figure 1.8.

Blobs E, H, I, K, L are of type BASIC, blobs B, C, D, F, G are of type OR and blobs A and J are of type AND. It is also useful to note that OR-blobs may contain AND-blobs and vice-versa. Also, BASIC-blobs may be contained within any type of blob except BASIC-blobs.

This definition so far has much expressive power on its own. An example of a basic class diagram can be seen on the left side of Figure 1.9. An example which gives a static description of the components of an automobile, can be seen on the right side of Figure 1.9.

Next, *transitions* are used in order to specify system dynamics in the diagrams. A transition is simply an arrow which indicates the system can change its current state from the source state of the transition to the destination state of the transition. Transitions are normally labeled with a construct of the form $e[c]/a$ [32]. This syntax denotes that the transition *fires* when event e occurs

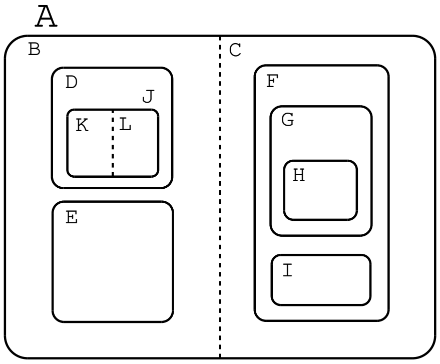

Figure 1.8: A first look at the basics.

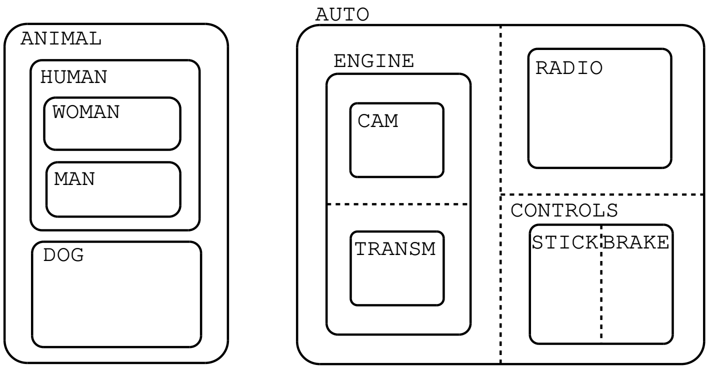

Figure 1.9: An automobile specification.

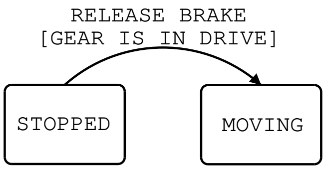

Figure 1.10: The transition from STOPPED to MOVING.

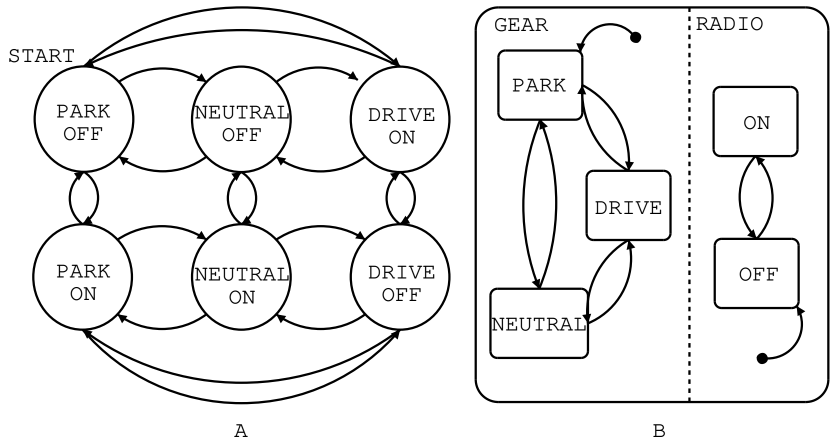

Figure 1.11: Statecharts are compact.

and condition c is true. The 'slash a' usually means one of two different things, depending on the variant of statechart. In some definitions, '/a' denotes that upon firing the transition, the a event is broadcast throughout the entire statechart, which may trigger more transitions. In the UML definition, '/a' usually denotes a series of actions (such as computer code) to execute, upon firing the transition. The UML suggests the potential use of a function $GEN(e)$, which can be one the many steps in a. This function is used to broadcast the event e. Most statechart variants combine these two aspects. An intuitive example of this (without actions or broadcasted events) is given in Figure 1.10.

Moreover, there are different kinds of transitions. First, there is the *initial transition*. It is the default transitions that fires upon entering an OR-blob, AND-blob or the root. They force the diagram to be deterministic by denoting which is the current state upon entering an OR-blob. They are usually drawn as a small filled-in circle with the connected transition pointing towards a blob and may only be labeled with an action (no trigger event).

1.4.1 State, Transition Relaxation

The orthogonal component property of statecharts solves the problem of state explosion in a FSA when trying to denote synchronization between concurrent systems. For example, in Figure 1.11a, there is a FSA which has two independent components: the radio and the gear shift of an automobile. In fact, it is not immediately noticeable that the FSA has 2 independent components. But with some inspection it is seen that all the states actually consist of a 2-tuple indicating that there are probably two separate processes occurring simultaneously.

To keep the example simple (because the number of states really does increase rapidly) the gear shift will be denoted as being in one of three possible states: PARK, DRIVE and NEUTRAL. Switching gears is the event that will change the state of the shifter from one gear to another. Also, it helps to abstract the radio behaviour by allowing only two states, ON or OFF.

Notice that in the FSA specification there are 6 states to describe the system fully. There are also 18 transitions to move between the various states. A possible statechart description (Figure 1.11B) reduces the number of states (blobs) to 5 and the number of transitions to 8.

When design modifications are needed, the statechart formalism becomes a great help. What if another state must be added to the radio system? The number of states in the FSA grows to 9, while the number states (blobs) in the statechart only increases by one. In fact, the number of

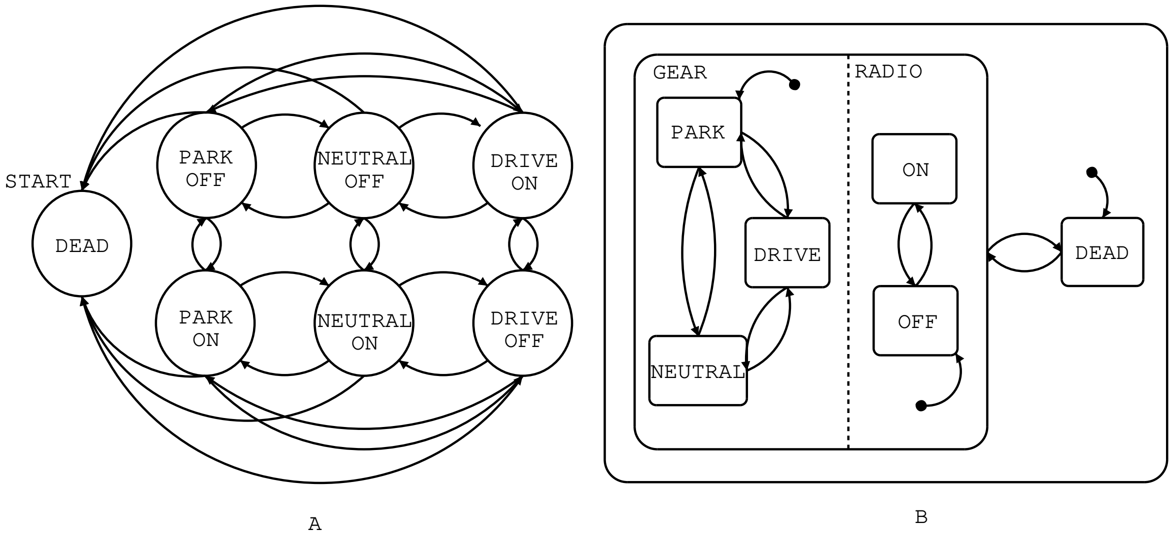

Figure 1.12: It is easy to modify statecharts.

states in the FSA specification is the product of the number states in the orthogonal components of the statechart. Therefore, the number of states in the FSA grows exponentially faster than the number of states (blobs) in the statechart as they are added to the orthogonal components.

Suppose more changes are needed. An important part of a car is the *DEAD* state which indicates that the automobile has not been started via the key ignition. In this DEAD state all the other information about the gear and the radio becomes unimportant. The car is DEAD, so the gear of the car and the state of the radio does not matter (this is obviously simplified - most cars today allow you to play the radio while it is not running). To make this modification of the FSA, an extra state must be added, and a transition must be drawn from every previously existing state to this new state in the event that the key is turned and the car stops running. Moreover, an extra transition is needed to indicate that when the car starts, the radio will be off and the gear will be park (this is also not exactly true since the most cars today can be started in DRIVE or NEUTRAL, but this point will be touched upon later). This is exemplified in Figure 1.12A.

In the statechart specification, making this change is much simpler. Add the blob outside of the CAR RUNNING blob and connect the two with transitions indicating that when the car starts, the GEAR and RADIO information come into scope. This is seen in Figure 1.12b.

Notice how complex the FSA specification has become. On the other hand, the statechart diagram immediately displays several properties of the system: the two concurrent components GEAR and RADIO and well as the fact that the DEAD state is reachable at any time.

1.4.2 More Statechart Elements

The previous example will be built upon to make it slightly more realistic. Consider a car whose gear is DRIVE and who's radio is turned ON. Also, the car is running. If the car suddenly stops running via the key ignition or perhaps stalling, the car's state will be DEAD. Once the car starts up again, the state of the gear and radio should really be DRIVE and ON respectively. However, the specification denotes that the state of the gear and radio will be reset to their default values. This is not the desired effect and there exists some mechanics in statecharts to solve this problem: the history blob.

The history blob lies inside of a blob B and indicates that upon entering B, the current state should be updated to the state which was visited last before leaving B. So, in the previous example, this

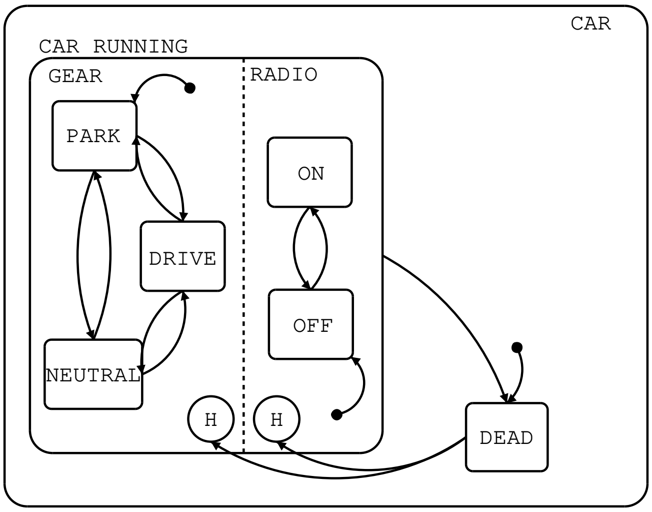

Figure 1.13: A look at history.

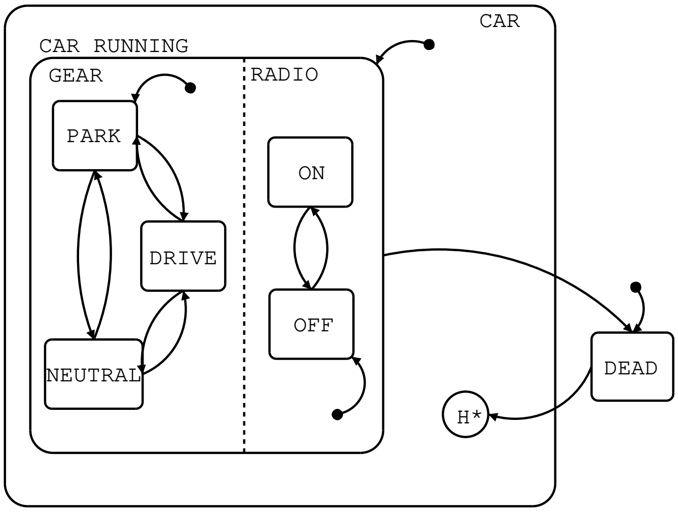

Figure 1.14: H* retrieves the deep history in the hierarchy.

problem is solved by adding a history blob to the CAR RUNNING blob and a transition from the DEAD state to the new history blob. The current state, upon re-entry, will be updated to whatever it was when the car became DEAD.

It is important to discuss the *scope* of the history state. The history state only applies to the blobs which are contained in the same blob as the history state itself. Since there are two orthogonal components in the CAR RUNNING blob, how is it possible to denote that the history is to be updated in each of GEAR and RADIO?

In order to explicitly state that the history must be updated in each of the two orthogonal components, GEAR and RADIO, use some notation from higraphs. Higraphs use special edges called hyperedges to denote moving from one state set to another, each of which may have a different cardinality. This hyperedge will simply be an edge which starts at the DEAD blob and splits off like a snake‘s tongue to both of the history states in each of the orthogonal components. This is drawn in Figure 1.13.

What about specifying the history for the BASIC-blobs that are nested quite deep within an OR-

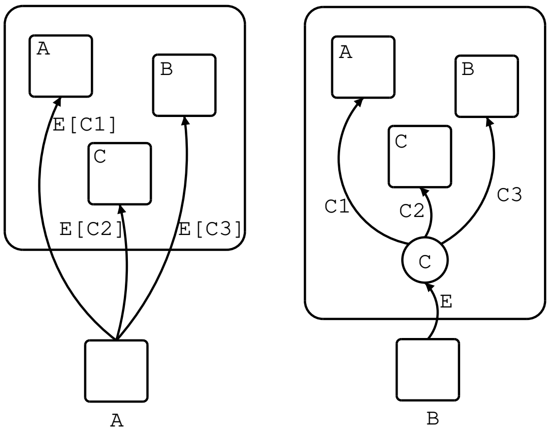

Figure 1.15: The *condition* construct.

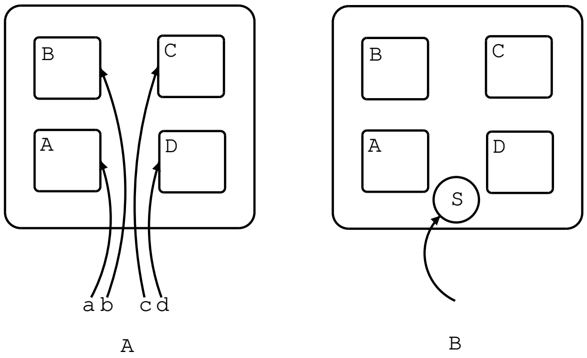

Figure 1.16: The *select* construct.

blob? Is it necessary to draw hyperedges to all the nested BASIC-blobs, if the history is to be applied as such? In fact, it is not required to draw all these hyperedges. The H* deep history pseudo-state is another special state which sets the history to apply to the deepest nested BASIC-blobs within the state at the level of the H* state. Knowing this, the diagram in Figure 1.13 can be revised to an alternative notation seen in Figure 1.14.

Another feature of statecharts which can help simplify the visual intuitiveness of the diagrams is the *conditional*. Normally denoted as a *C*, the conditional gets rid of edges which are enabled on the same event but fire on different conditions.

This is most easily seen in Figure 1.15a. Note that Figure 1.15A can be represented as 1.15B.

The selection feature of statecharts is similar to the C/C++ select statement. The selection element simplifies diagrams where several transitions enter a blob, without any condition requirements, and each is in a one-to-one correspondence with the inner blobs. That is, each transition enters blob B and each links to some inner blob. An example of this can be seen in Figure 1.16A. This can be further simplified by using the selection feature, usually denoted with an *S*. The transitions must be specified explicitly, but can be omitted for simplicity such as in Figure 1.16B.

1.5 How do Statecharts Work?

Until now, this discussion has been focused on the syntax of statecharts. Proper syntax basically means to draw a legal statechart. But how do statecharts really work? What are the their underlying mechanics? In this section the *operational* semantics of statecharts will be outlined. This will be the basis for building a Statechart simulator.

1.5.1 Operational Semantics

Much work has been done on describing the semantics of statecharts. The Statecharts formalism was first introduced in [27] and its semantics were discussed, though not in detail. The first rigorous definition was given in [33]. Many papers have proposed variations on statecharts as well as describe some semantics [68].

A loose definition equates Statecharts as follows.

Statecharts = State Automata + Hierarchy + Orthogonal Components + Broadcasting

Many approaches map statecharts onto another known formalism such as ESTEREL [59], etc. Having understood the ESTEREL construct, some inference can be made about statecharts upon performing this mapping. One of the most widely referenced papers on statechart semantics is [32]. This paper describes in detail the semantics of statecharts implemented in the STATEMATE software tool developed by the iLogix corporation. The paper's description takes the form of plain English and pseudo code. Thus, the same approach is taken in this chapter.

Many subtleties arise upon specifying exactly what happens when a statechart is put in motion. For example, there must be a carefully defined algorithm which deals with external events generated at any time. Once an event is generated, when does the system stop following transitions? What if a loop occurs? What if transitions cross state boundaries into different levels of hierarchy, which states are entered and exited? It is clear there are many problems to be classified and solved. This is exactly the purpose of having an operational semantics.

1.5.2 The *STATEMATE* Semantics of Statecharts

The *STATEMATE* semantics of statecharts, [32], is probably the most popular and most cited semantics for statecharts today. It is the semantics used to implement iLogix's statechart tool, *STATEMATE*. This tool can be used for building valid statecharts, simulating these statecharts with random or user specified events and generating portable code from the statechart mainly used for running embedded hardware.

First, consider what is classified as a single *step* using the STATEMATE semantics. A step takes zero simulation time. That is, the clock which tracks the statechart's notion of time is not incremented during the execution of a step. Once an external event is generated, transitions which respond to this event become *enabled*. Recall that transitions are labeled $e[c]/a$, meaning the transition fires and action a is executed if event e occurs while condition c is true. So, if event e occurs but condition c is not true, this transition cannot be followed nor will the *current state* be updated. The current state refers to the state of the statechart model. If a state is current, then so is its containing state. This means the current state is actually a *tree structure*, extending from the root, which is always active, down to the basic states at the leaves of the tree.

If the condition c is true, the system proceeds to execute action a and update the current state to that of the transition's destination state. If the action generates another event which in turn triggers another transition to become enabled and fire, then this is also carried out within the very same step. This process is repeated every time the execution of a transition's action causes another event to fire. When no more internal broadcasted events trigger more transitions to fire, the step is considered to be completed. The series of steps which leads to another stable configuration is called a *macrostep* while the smaller series of steps are *microsteps*.

Another note about *time*. The fact that execution of a step takes zero time helps us realize an important distinction between simulating a statechart and generating executable code from a stat-

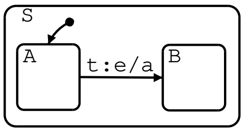

Figure 1.17: A look at semantics.

echart. When simulating or animating a statechart the user may choose to speed up or slow down simulation time as the statechart knows it. The user can define the time model used upon running the simulation. Once executable code is generated, however, the time model is inherent to the statechart design and the system which runs the code. Moreover, a step clearly does not take zero time.

Some more important rules which will be discussed in more detail follow. Every operation that is carried out during the execution of a step solely depends on the status of the system at the *beginning of the step*. This become an important point, later, for certain special cases. Reactions or updates from external and internal events can only be *realized* until *after* the step is completed. A maximal set of non-conflicting transitions is always executed. This means that if a transition is enabled, it will be executed as long as there is no non-determinism. It may happen that when several transitions are enabled, some may be conflicting and unable to be executed in the same step. A priority scheme is one way to resolve conflicting transitions.

There are special events which are generated to mark the entering and exiting of a Statechart, A. These events are $entered(A)$ and $exited(A)$ for entering and exiting a Statechart, respectively.

Some examples follow which start very simple and become progressively more complicated. A very simple example can be seen in Figure 1.17 [68]. Suppose that the system is currently in state A when it suddenly receives the external event e. The following operations are carried out:

1. Transition t becomes enabled and fires (there is no conditional constraint on this transition). The system will exit state A and will enter state B (the current state will be B after the step has completed).

2. Action a is executed.

3. More events are generated: $entered(B)$ as well as $exited(A)$.

4. The condition $in(A)$ becomes false while $in(B)$ becomes true.

5. The static reactions which were executing while the system was in state A continue to execute in state B since state S is never exited. So, the static reaction for state S is executed.

6. *Static reactions* are actions associated with a state that are continuously executed upon entering. The *static reactions* for A are disabled and stop running while the static reactions for state B become enabled and start running.

Think, now, about what happens during actions. In the most intuitive implementations, actions are simply computer code. Since code can have scope, what happens when the value of a single global value is changed from two different actions, executed in the same step? In Figure 1.18 an example of this situation is given. If the state of this system is (A,C) and event e is generated, both actions a_1 and a_2 will be executed in the same step. The action are defined as follows:

- $a_1 : x \leftarrow 5$
- $a_2 : x \leftarrow x + 1$

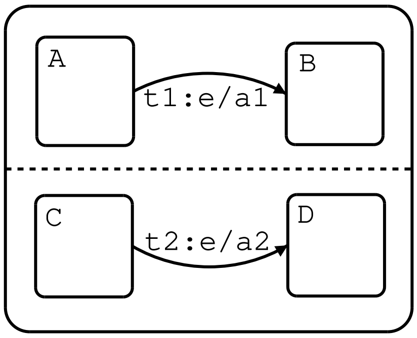

Figure 1.18: More semantics.

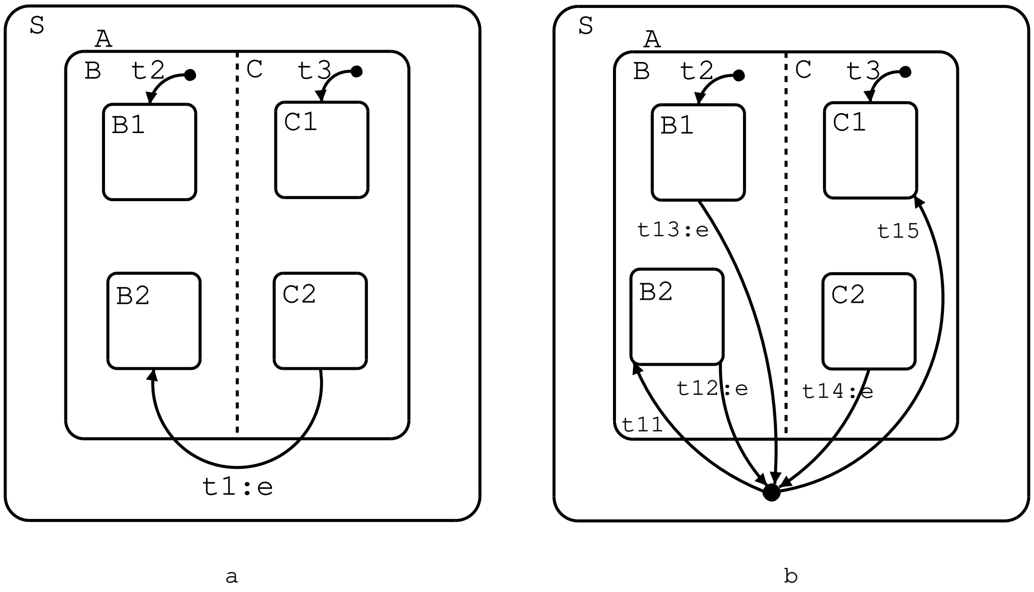

Figure 1.19: A look at scope.

a_1 defines an action where the value 5 is moved into data buffer x. Action a_2 takes the value in data buffer x, adds 1 to this value, then moves this new value back into buffer x. What is the value of x after this step is completed? It is difficult to say since the order in which the actions are executed is unknown. It appears that there exists a *race condition* and the outcome of this step is unknown. A statechart tool should have the capabilities of detecting and reporting such a race condition. Chances are that a race condition is an undesirable behaviour so the current statechart model must be refined to avoid these types of problems.

1.5.3 Scope

The scope that transitions maintain is another important factor to keep in mind when designing statecharts. The scope of a transition refers to the lowest OR-state in the hierarchy which is not exited when traversing a transition. This also means that the scope state is a proper common ancestor of all the source and target states of the transition. For example, in Figure 1.19a, taking transition t_1 causes the current state to change from $(B2, C2)$ (or $(B1, C2)$) to $(B2, C1)$. This means exiting $C2, B1(orB2), B, C$ then A and entering $A, B, C, B2$ then $C1$. State S was not exited, thus S is the scope of transition t_1. Figure 1.19b helps to see the actual transition represented by t_1 in Figure 1.19a. Note that transition t_1 consists of transition segments t_{11}, t_{12}, t_{13} and t_{14}.

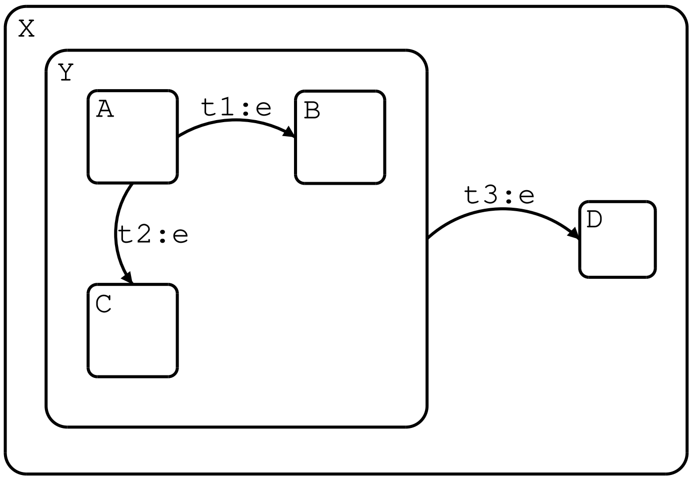

Figure 1.20: A look at non-determinism.

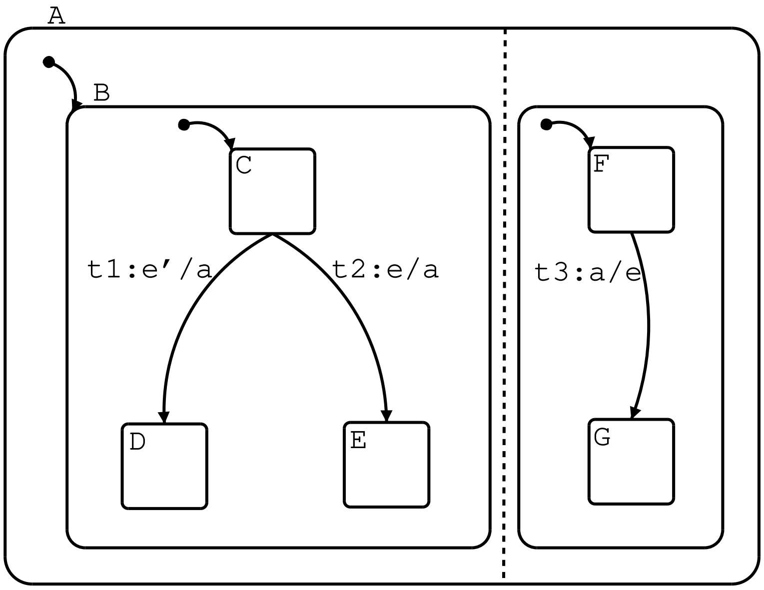

Figure 1.21: A look at compositionality.

1.5.4 Non-determinism

It is important to note how the STATEMATE semantics treats non-determinism in various cases. There is a *top-down priority* when it comes to conflicting transitions which gets rid of non-determinism in many, but not all, cases. First, look inside state Y in Figure 1.20. It is evident that there is non-determinism if the system is in state A and event e is generated. Transitions t_1 and t_2 are said to be in *conflict*. The system wants to update the current state to both B and C, which is of course, meaningless and incorrect. This configuration is illegal and should generate an error. However, is it apparent that within the entire system there is still a conflict between t_1 and t_2. But, since the scope of t_3 is higher in the hierarchy than t_1 and t_2, it has priority and can fire without any confusion.

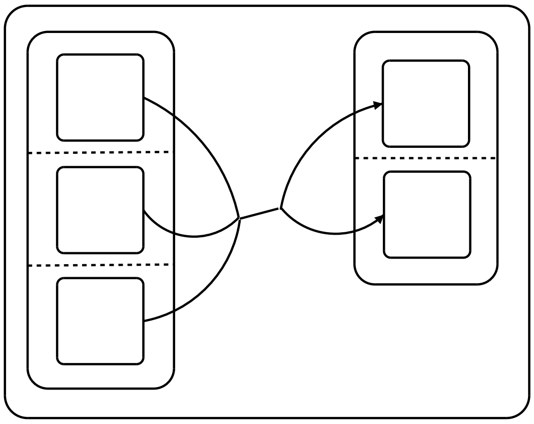

Figure 1.22: A look at hyperedges.

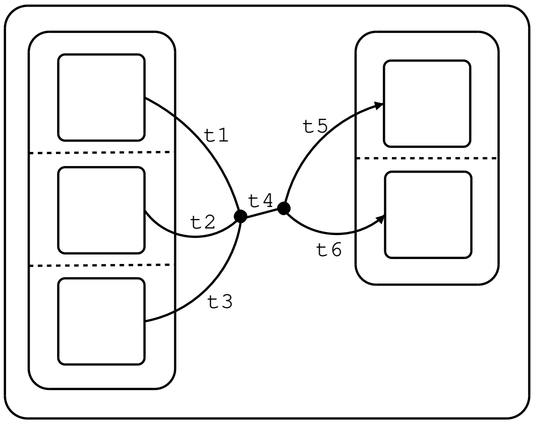

Figure 1.23: A look at joints and forks.

1.5.5 More about time

The STATEMATE semantics of statecharts includes two models of time: *synchronous* and *asynchronous*. The synchronous time model uses an internal clock to decide when to execute another external event. This means, at every clock tick, the system can execute external events. These external events must be generated in the previous step, when the system was not processing anything, or they will go unnoticed. The asynchronous time model handles external transitions as they are generated. Thus, several steps can take place within a single point in time, which differs from the synchronous time model. However, in both methods, a step still takes zero time (that is, the clock stops) and the statechart still does not react to external events when it is processing events.

In the synchronous sense, a problem exists with Figure 1.21. First note that e' refers to the absence of event e. If this statechart is in state (C,F) and the e' event is generated, an internal event, a, is generated. This, in turn, causes t_3 to fire in the other orthogonal component, which in turn causes an e internal event to be generated. Upon generating an e, transition t_2 should fire since status of the system at the *beginning* of the step is all that is taken into consideration. However, this point arose because there was an absence of event e. This problem is called a lack of *compositionality* and is discussed in more detail in [45].

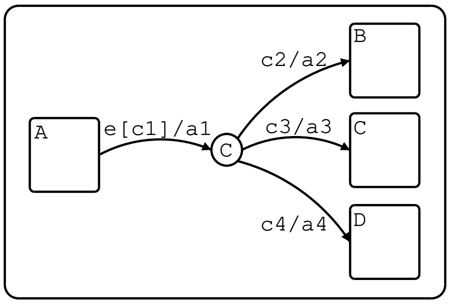

Figure 1.24: What happens when e occurs, c_1 is true but none of c_2, c_3 or c_4 are true?

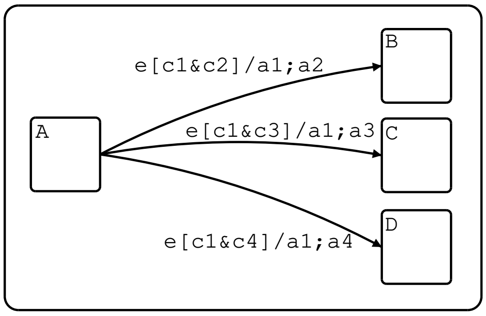

Figure 1.25: Another way of looking at fig. 1.24.

1.5.6 More about transitions

In most of the examples so far, every transition has been a regular graph edge. That is, the cardinality of the source and destination sets in each case has been one, so far. Now consider how to represent hyperedges, where the cardinality of the source and destination sets can be more than one, such as in Figure 1.22.

In addition to being able to represent hyperedges, the statechart can never end a step by stopping at a conditional, select or history construct. For example, in Figure 1.24, if event e occurs and c_1 is true but none of c_2, c_3 or c_4 are true, then does the system stop at the condition, or does it roll back to state A? It would seem logical to simply stay at state A in this case.

In order to check that one of these *entire transitions* fires it must be broken up into *transition segments*. Since after each transition fires, the system should be in a stable configuration, the transition segments are joined to ensure that all conditions are satisfied before letting the entire transition fire. This conjunction of transitions is called a *basic compound transition* (CT) [32]. For example, Figure 1.24 becomes Figure 1.25. Note that in order to ensure that they are all true for the basic CT to fire, simply take the logical AND of the conditions. Also, note that two actions in each basic CT must be executed, each corresponding to the transition segment actions.

There are two types of *connectors* which allow grouping of transition segments. The first is the OR-connector. This has already been seen in Figure 1.24, where the OR-connector is actually the condition entity. The OR-connector is named this because only one *path* traveling through the connector needs to be executable. That is, if T_1 is the set of transition segments going into an OR-connector C, and T_2 is the set of transition segments going out, then a transition moving through T_1 and T_2 will have one segment from T_1 and one from T_2. This differs from an AND-connector which

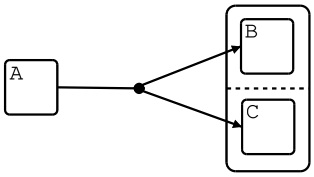

Figure 1.26: An AND-connector.

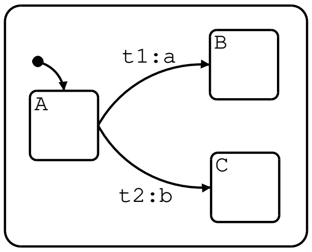

Figure 1.27: What if events *a* and *b* are generated at the same time?

usually involves orthogonal components and can be seen in Figure 1.26. In this case, if T_1 is the set of transition segments going into an AND-connector, C, and T_2 is the set of transition segments going out, then the transition moving through T_1 and T_2 contains all the transition segments in $T_1 \cup T_2$.

So, a hyperedge is simply an edge with AND-connectors. The hyperedge in Figure 1.22 is drawn with AND-connectors in Figure 1.23.

1.6 More Statechart Efforts

1.6.1 Statecharts and the Object Oriented Paradigm

One recent and popular research effort is to combine statecharts with class diagram-like formalisms. Class diagrams are static in the sense that they represent the relationship of many complex entities but do nothing to specify any semantics of execution. Specifying execution semantics is the main purpose of this fusion between statecharts and class diagrams. The goal is to bring to life the high-level designs created using class diagrams by linking them with a full statechart specification.

One example of object execution modeling can be found in [29]. The class diagram-like formalism used in this paper is called O-charts. O-charts are similar to class diagrams because they contain information about class objects, multiplicities, inter-associations and of course inheritance. Once a class is specified using the O-chart formalism, its behavior is specified using statecharts. Many issues must be addressed such as initialization, references to instances, creation and deletion of instances, inheritance and even threading. These issues are left open-ended in [29].

1.6.2 Subtle Semantics

A well-defined summary of some important and less obvious issues regarding statechart semantics is given in [68]. A few will be discussed in this section.

It is important to allow negated triggers. Negated triggers can help reduce non-determinism in several cases. For example, consider Figure 1.27. This appears to be a simple, deterministic statechart. However, if events *a* and *b* are generated simultaneously, a non-deterministic case arises. Luckily, negated trigger events can be used to work around this problem. All that is needed is to change one

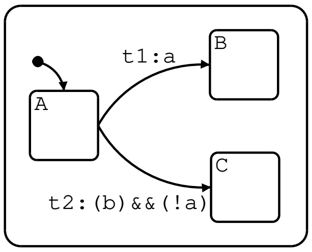

Figure 1.28: A solution to 1.27 using negated trigger events.

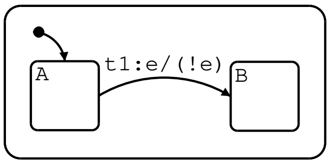

Figure 1.29: The opposite event is generated from the trigger event.

of the transitions' triggers to ensure exclusivity. An example of this can be seen in Figure 1.28. In order for t2 to fire, event *b* must occur along with the explicit non-occurrence of event *a*. This small modification ensures the statechart will be deterministic in the case that events *a* and *b* are generated at the same time.

By allowing negated trigger events a new problem arises. Transitions fire because of a certain event being generated. This can be thought of as simply doing something because of an event *e* being generated. So, in a situation such as Figure 1.29, action !*e* is executed when this is triggered by *e*. This behaviour is confusing and inconsistent.

Different semantics handle this situation in different ways. Most semantics dictate that if a transition fires and generates an event *a*, then the event !*a* cannot be a valid trigger in any future micro steps. Only some semantics say that if event *a* is generated at transition *t*, in a series of micro steps, then !*a* cannot be a valid trigger for a previous transition in this series. The preferred semantics is to be only concerned with future micro steps.

Another subtle problem exists in statecharts like that of Figure 1.30. If state B is entered and exited in the same instance of time, does the macro step halt at state B? Most semantics prohibit such a construct. However, there are advantages as well as disadvantages for allowing such instantaneous states. One desirable property of allowing instantaneous states is that only a finite number of micro steps can occur in one macro step.

This problem can be simplified to the example in Figure 1.31 in order to study the effects of allowing such a semantics. If state A is the current state, how many times must event *a* occur so that state *C* is reached? If the answer is that only 1 occurrence of event *a* is sufficient to get to state *C*, then it is possible for an infinite sequence of micro steps to be executed in a macro step. Conversely, two occurrences of event *a* will work just as well. Perhaps the best statechart

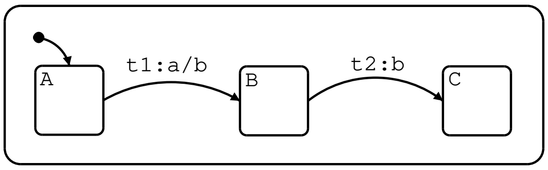

Figure 1.30: Is state B instantaneous?

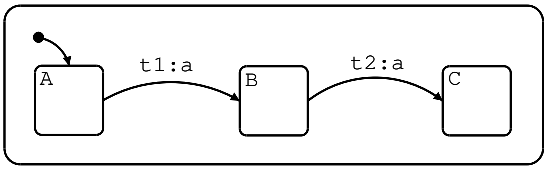

Figure 1.31: How many times must event *a* occur in order to reach state *C*?

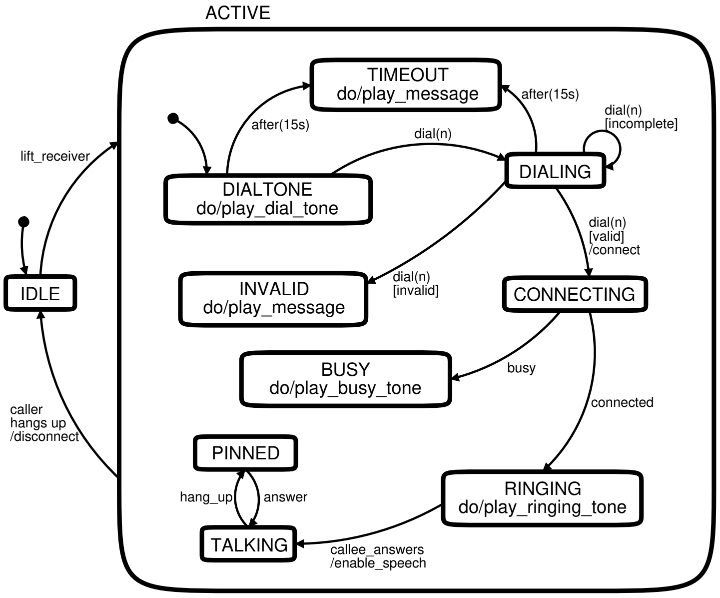

Figure 1.32: An example UML statechart - A simple telephone.

environment would allow for both types of semantics.

1.7 Other formalisms of interest

1.7.1 UML Statecharts

The Object Management Group (OMG) is the organization responsible for defining the Unified Modeling Language (UML) among several other standards such as CORBA. Included in the UML are the specifications of *State Diagrams* which closely resemble STATEMATE statecharts in their syntax and semantics. The OMG have developed their version of statecharts to be geared towards designing application level software, where more classical definitions have been mainly used for lower level software. That is, their statechart definition has been modified to be more compatible with the object-oriented paradigm. Firstly, the UML adds a few syntactic components which do not appear in most statechart specifications.

For the most part, UML statecharts look identical to most other types of statecharts. They have all the core features of statecharts that were major developments from flat state machines like FSA. Such features include hierarchy and orthogonal components. An example of a UML statechart is given in Figure 1.32.

Like most statechart variants, states have the following special actions:

- Entry: The action to be executed upon entry into this state.
- Do: The action to be executed while this state is current.
- Exit: The action to be executed upon leaving this state.

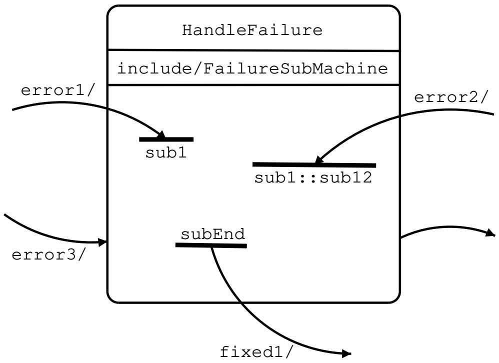

Figure 1.33: Using stub states.

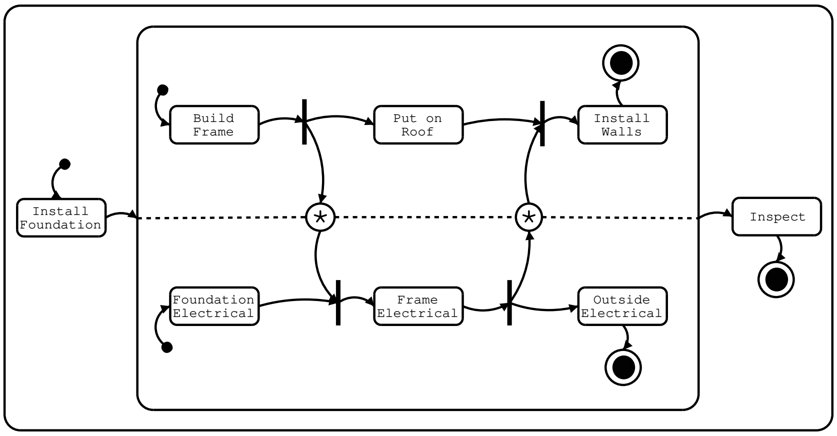

Figure 1.34: Using synch states.

- Include: Used to link other statechart specifications to be referenced within this state.

If *include* is used, one then has the ability to reference the included statechart. Since the specification of these statecharts remains hidden, a special notational component called a *stub state* is used to denote which state a transition terminates on within the included sub machine. An example of including a sub machine can be seen in Figure 1.33 [51]. Upon generating event *error*1, the corresponding transition fires and terminates on state *sub*1 within Handlefailure. Generating *error*3 causes the system to enter the default state of FailureSubmachine. If the *fixed1* event is generated, the system will leave HandleFailure granted the current state is *subEnd*. Referencing a sub-sub state of the included statechart is written *sub*1::*sub*2::…::*subn*. In Figure 1.33, *error*2 causes the system to terminate on state *sub12* which is a substate within *sub1* of FailureSubmachine.

Stub states make it possible to define a statechart, put it in a library and reference it later when needed. This is exactly the same as the object-oriented world. In fact, the UML specification states that these statecharts can be used to define class instances.

Another interesting notational component introduced in the OMG UML specification is the *synch state*. Synch states are used in conjunction with hyperedges to synchronize activities between orthogonal components. Visually, they are circles with a number inside. The number indicates an upper bound on how many times a transition may fire and may also be unlimited (*). Synch states usually appear on the boundaries of concurrent regions. An example of synch state usage is given in Figure 1.34 [51]. Notice that, before the electrical parts can be installed in the frame, both the frame and the electrical parts of the foundation must be installed. The synch state ensures these

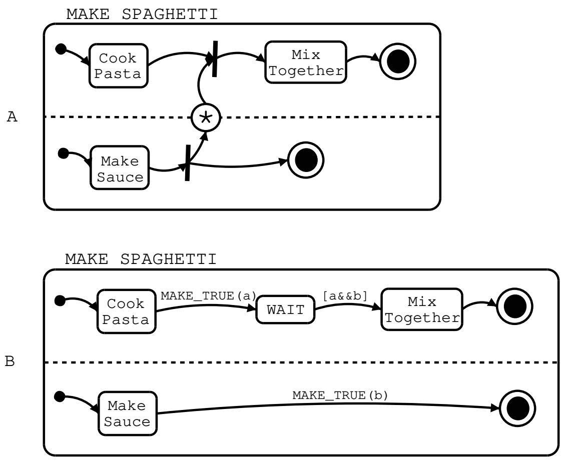

Figure 1.35: Converting synch states into *core* elements.

two activities are completed before electrical parts are installed in the frame. This also denotes there is a type of dependency between these activities.

Synch states can be modeled using only the ”core” elements from the usual statechart definitions. One possible method of converting a synch state into only *core* elements is to use the MAKE_TRUE with boolean operators and guards. An example of such a conversion can be seen in Figure 1.35. The process of preparing spaghetti requires cooking the pasta and making the sauce before mixing the two together. The synch state in Figure 1.35A ensures this constraint holds. Notice that in Figure 1.35B the *cook pasta* is followed by a *wait* state. This is done to allow a boolean variable a to be set to true. The bottom orthogonal component sets a boolean variable b to true as well. Once both activities are completed the boolean expression $a \wedge b$ will be true and the *mix together* activity can begin.

Refining UML Statecharts

The OMG have decided to go one step further and define rules for refining statechart models in the event that their statechart formalism is used to specify class behaviour. In this framework, each statechart diagram is a class instance in the object oriented paradigm (OOP). Since in the OOP classes can be extended in various ways, the same should be allowed with statechart models.

There are three basic methods of refinement that the OMG suggests may be useful: sub-typing, strict inheritance and general refinement.

Subtyping It is very important to preserve pre/post conditions of the class specification when subtyping. This preservation is what supports to substitutability. In the subtyping framework, states and transitions can only be added, not removed. The set of outgoing transitions must remain intact, but more may be added. There are no constraints on modifying the set of incoming transitions. More substates may be added and may even be divided into orthogonal regions. A refined transition may end on a substate of the ending state in the super class. Moreover, disjunctions may be added to refined guards (this means extra OR clauses) since this only relaxes the boolean expression. Procedures executed on transitions must maintain the same actions in the same order. More actions may be added granted they do not alter the course of the macrostep.

Strict Inheritance Here, the internals of a state machine are of no concern. The goal is to simply allow implementation reuse. Like subtyping, a refined state can change its set of incoming transitions, add outgoing transitions, add substates and may split into concurrent regions. A refined transition can terminate on a different target state but must maintain the source state. Guards can

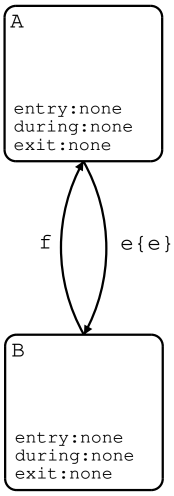

Figure 1.36: Stateflow loops.

be completely modified. Procedural actions can be removed and added, but must remain in the same order.

General Refinement This is the most relaxed refinement policy. Refined states may have different incoming and outgoing transitions. Refined transitions may have a different source and destination. Also, guards and procedures can change altogether.

1.7.2 Notes on Semantics

The OMG outlines several semantic details which are tightly intertwined with implementation issues, since their documents are supposed to aid a developer in building a statechart environment, not how to build statechart models.

The following is a list of a few subtle semantic points of interest taken from [51].

- The OMG statechart specification clearly leaves details regarding event durability open-ended. They want to be able to allow several different semantic models such as zero-time semantics.
- Instantaneous states are allowed.
- UML statecharts employ one transition priority rule which helps extinguish non-determinism. This rule states that a transition who's source state is deeper in the state hierarchy has a higher firing priority. Note that this is the opposite of the STATEMATE semantics.

1.7.3 Stateflow

The creators of MATLAB (The Mathworks) have built a development environment using a formalism similar to statecharts called stateflow. The package is designed to work with The Mathworks' flagship products Simulink and MATLAB for designing and simulating complex, event-driven systems.

Stateflow is largely based on the statechart formalism, containing most features present in [32] such as hierarchy and orthogonal states. It lacks certain constructs such as the select and history entities. Another small difference is the representation of transition actions as well as broadcasted events. Transition actions are represented as the part after the slash on a transition, while broadcasted events are written within curly brackets.

The operational semantics of a stateflow diagram is also similar to the STATEMATE semantics

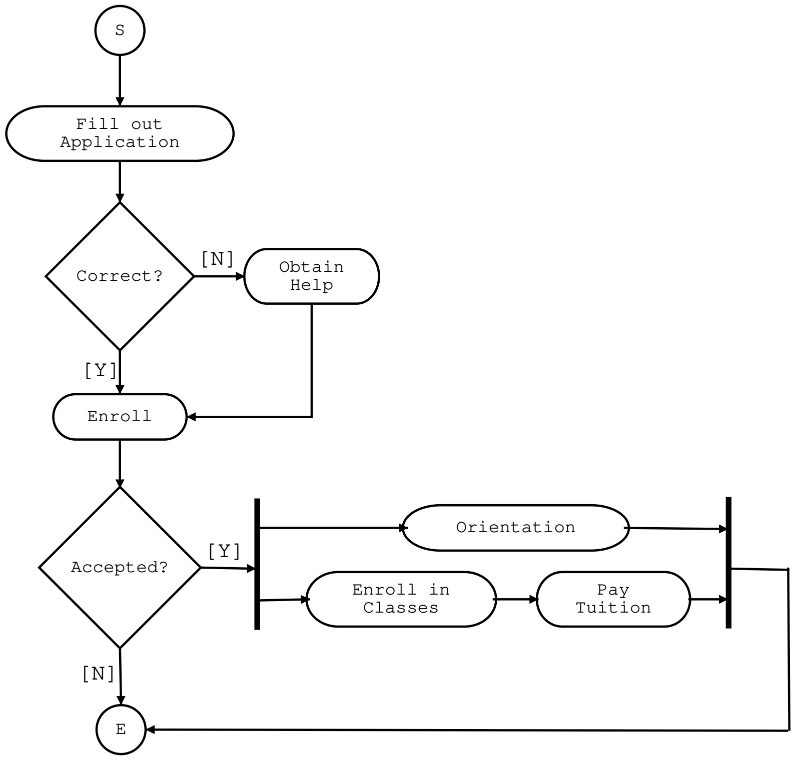

Figure 1.37: An activity diagram: The college application process.

of statecharts. One disagreeable semantic result is the possibility of loops within stateflow. For example, in Figure 1.36 if the system is in state A and event *e* occurs, the system should move to state B and broadcast event *e*. However, the formal semantics of a stateflow diagram states that internally broadcasted events are incepted before the current state is updated. Thus, in Figure 1.36 upon receiving an *e* event the system starts to move from state A to B. Another *e* event is broadcast. Since the current state is still A, the same transition fires! This results in an unwanted infinite loop and is a condition of which a developer using stateflow should be aware.

1.7.4 Activity Diagrams

The activity diagram is a visual formalism for describing work flow. This means they illustrate sequences of activities ordered such that dependencies are easily visible. Synchronization of activities is represented in the same way as petri nets. A bar is drawn and multiple arcs may connect to the bar to ensure that these activities are all completed before moving onto the next. Other information that can be obtained from activity diagrams at a glance is the concurrency of activities as well as conditional branching.

Activity diagrams are drawn using circles for the activities and directional arcs to denote the completion and beginning of activities. There are special start and end markers to denote the beginning and completion of the entire task. Conditional branches are drawn using a diamond, similar to the UML dataflow formalism. An example of an activity diagram is given in Figure 1.37 which is taken from [2].

One major difference between state diagrams like finite state automata and activity diagrams is that an FSA blob represents the static state of the system, while an activity diagram blob represents the system while performing an action. Statecharts, in fact, can represent both situations. Basic states (states with no children) are similar to FSA states, while non-basic states (states with children) are

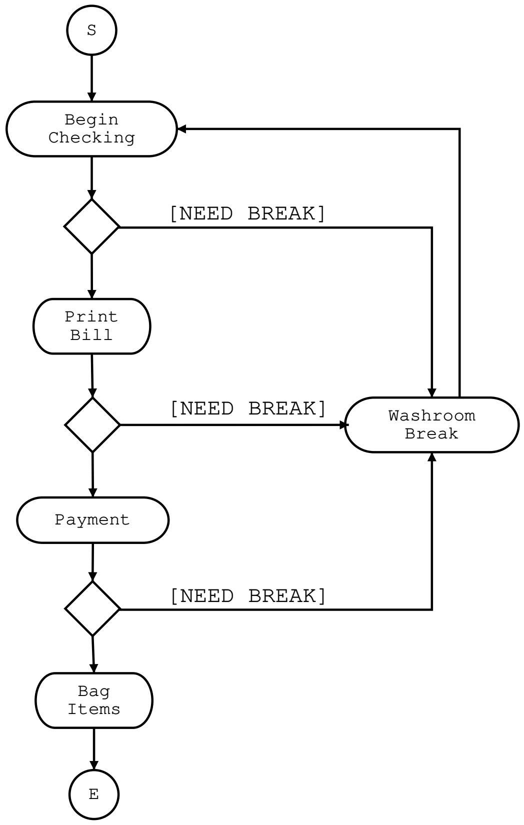

Figure 1.38: Activity diagrams lack compactness.

similar to activities, since an entire statechart may be inside.

Some describe activity diagrams as a similar formalism to statecharts, but they lack one of the most important features of statecharts: hierarchy. For example, in Figure 1.38 the grocery transaction process from the cashier's perspective is modeled using an activity chart. Note that in order to represent a washroom break at each step of the way, these arcs need to be explicitly modeled. Statecharts provide a more modular solution, allowing the user to group the main activities using hierarchy then only draw one arc to specify the washroom interrupt. Moreover, note that after the break, the cashier would normally return to the step which he or she was doing previously. This is easily modeled using the history construct in statecharts, but requires some bulky conditional branches in activity diagrams. To save space, simply indicate that after their washroom break, the cashier proceeds to *start checking items* again, which is probably not the case in reality.

1.8 Conclusion

In this chapter, the world of statecharts was explored. Like all visual formalisms statecharts have a syntax as well as a semantics. While it is relatively easy to specify a visual syntax, the semantics is quite another matter. One of the most important semantics definitions, STATEMATE, was examined as well as several semantic issues present in all statecharts specifications.

It is clear that more research must be done to standardize the formal semantics of statecharts as there are far to many variants. The bridge should be gapped between current statecharts research and standardization groups such as the OMG who defined the UML.

2

AToM^3: A Tool For Multi-Formalism Meta-Modeling

2.1 Introduction

Complex systems are characterized by components and aspects whose structure as well as behaviour cannot be described in a single formalism (due to their different nature). For example, if one wishes to model a temperature and level controlled vessel, the controller can be described with a discrete formalism (such as Petri-Nets or Statecharts [28]) whereas the behaviour of the liquid (which describes the variation in volume and temperature) should be described using a continuous formalism (such as Ordinary Differential Equations or CBDs).

One of the approaches to tackle complex systems is multi-formalism modelling. In this approach the different parts of the system are modelled using different formalisms. In order to analyze a multi-formalism system, it is not enough to look at each component in isolation. One must consider the whole system. For this reason, multi-formalism modelling attempts to convert all components into a common formalism which is closed under composition, so that the whole system can be properly analyzed or simulated.

In order to make the multi-formalism approach possible, the problem of dealing with a plethora of different formalisms must be solved. One would like to dedicated tools to model in each one of these formalisms, but the cost of building such tools from scratch is prohibitive. Meta-Modelling is useful to deal with this problem, as it allows the (possibly graphical) modelling of the formalisms themselves. A model of a formalism should contain enough information to permit the automatic generation of a tool to check and build models subject to the described formalism's syntax. The advantage of this meta-modelling approach is clear: rather than building a whole application from scratch, it is only necessary to specify the kind of models dealt with. If this specification is done graphically, the time to develop a modelling tool can be drastically reduced to a few hours. Other benefits are the reduction of testing, ease of change and maintainability.

At the very least, the generated tool should be able to allow the construction of valid models and discover errors in their construction. If (meta-)models are stored as graphs, further manipulations of the models can be described as graph grammars [57]. In Multi-Paradigm Modelling and Simulation, model manipulations such as the following are of interest.

Table 2.1: Meta-Modelling Levels.

Level	**Description**	**Example**
Meta-Meta-Model	Model describes a formalism that will be used to describe other formalisms. Specified with a meta-formalism	Description of Entity-Relationship Diagrams, UML Class Diagrams
Meta-Model	Model describes a simulation formalism. Specified with a meta-formalism	Description of Deterministic Finite Automata, Ordinary Differential Equations (ODE)
Model	Description of an object. Specified with a formalism	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

- Model simulation or animation.
- Model optimization, for example, to reduce its complexity.
- Model transformation into another model (equivalent in behaviour), expressed in a different formalism.
- Generation of (textual) model representations for use by existing simulators or tools. The type of model transformation discussed in this chapter.

In this chapter, AToM^3 [4] [18] is presented. It is a tool which implements the ideas presented above. AToM^3 has a meta-modelling layer in which formalisms are modelled graphically. From the meta-specification (a model in the Entity Relationship formalism extended with constraints), AToM^3 generates a tool to process models described in the specified formalism. Models are represented internally using *Abstract Syntax Graphs* (ASGs), a generalization of the concept of *Abstract Syntax Trees* used by compilers. The ASG represents –in the form of a graph– the syntactic information of the model built by the user. As a consequence, model manipulation can be expressed as graph grammars.

2.2 Computer Aided Multi-Paradigm Modelling

Computer Aided Multi-Paradigm Modelling (CAMPaM) [65][48] is a research area which has the objective to simplify the modelling of complex systems by combining three different directions of research:

- *Meta-Modelling*, which is the process of modelling formalisms. Formalisms are described as models described in meta-formalisms. The latter are nothing but expressive enough formalisms, such as Entity Relationship diagrams (ER) or UML class diagrams. A model of a meta-formalism is called a meta-meta-model; a model of a formalism is called a meta-model. Table 2.2 depicts the levels considered in this meta-modelling approach. Note that only three levels are considered, although it can be the case that a meta-formalism mf_1 is powerful enough to describe the meta-meta-model of another meta-formalism mf_2. Both mf_1 and mf_2 as meta-formalisms are considered and are placed in the same meta-level. As will be seen later, in AToM^3 it is usually the case that meta-formalisms can describe meta-formalisms as well as formalisms.
  To be able to fully specify modelling formalisms, the meta-formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models).

For example, when modelling Deterministic Finite Automata, different transitions leaving a given state must have distinct labels. This cannot be expressed within ER diagrams alone. Expressing constraints is usually done by adding a constraint language to the meta-formalism. Whereas the meta-formalism frequently uses a graphical notation, constraints are usually given in textual form. For this purpose, some systems [36] (including ours) take advantage of the Object Constraint Language OCL [51] used in the UML. As AToM^3 [4] is implemented in the scripting language Python [55], arbitrary Python code can also be used. Another alternative to using constraints is to express as graph grammar rules, the kind of editing actions the user can perform at each moment in the modelling phase. This approach is called *syntax-directed* [6]. Other kinds of visual editors are called *free-hand* [47] and allow the user more flexibility in the model editing phase, but they have to check that the model the user is building is correct. In AToM^3, free-hand editing is the default approach, and model correctness is guaranteed by evaluating the constraints defined at the meta-level (and associated with events) when the user is building the model. In AToM^3, free-hand editing can be combined with the syntax-directed approach by building graph grammar rules for editing tasks.

- *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
- *Multi-Formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms. In Figure 2.1, a part of the "formalism space" is depicted in the form of a Formalism Transformation Graph [64]. The different formalisms are shown as nodes in the graph. The solid arrows between them denote a homomorphic relationship "can be mapped onto". The mapping consists of transforming a model in the source formalism into a behaviourally equivalent one in the target formalism. The dotted, vertical thick arrows denote the existence of a simulator for the formalism, which produces simulation traces. This iterative simulation can be seen as a special case of formalism transformation (into the "*traces*" formalism). The vertical dashed line separates continuous (left) and discrete (right) formalisms, whereas the horizontal dashed line below formalism *"DAE non-causal set"* separates causal (upper) and non-causal formalisms. It can be observed how DEVS [70] (Discrete EVent system Specification) can be a suitable target formalism when the purpose of the transformation is simulation, as DEVS can be simulated by parallel, highly efficient simulators based on the HLA architecture.
  In this approach, the specification of composite systems by coupling heterogeneous components expressed in different formalisms is allowed. For the analysis of its properties, the composite system must be assessed by looking at the *whole* multi-formalism system. That is, its components may have to be transformed to a common formalism, which can be found in the FTG. In this approach formalisms are meta-modelled and stored as graphs. Thus, the transformations denoted by the arrows (both for simulation and for formalism transformation) of the FTG can be modelled as graph grammars.

2.3 AToM^3: An Overview

AToM^3 [4] [18] is a tool which uses and implements the concepts presented above. As it has been implemented in Python, it is able to run (without any change) on all platforms for which an interpreter for Python is available: Linux, Windows and MacOS. The main idea of the tool is: *"everything is a model"*. During its implementation, the AToM^3 kernel has been bootstrapped from a small initial kernel. Models were defined for bootstrapped parts of it, code was generated and then later incorporated into itself. Also, for AToM^3 users, it is possible to modify some of these model-defined components, such as the (meta-)formalisms and the user interface.

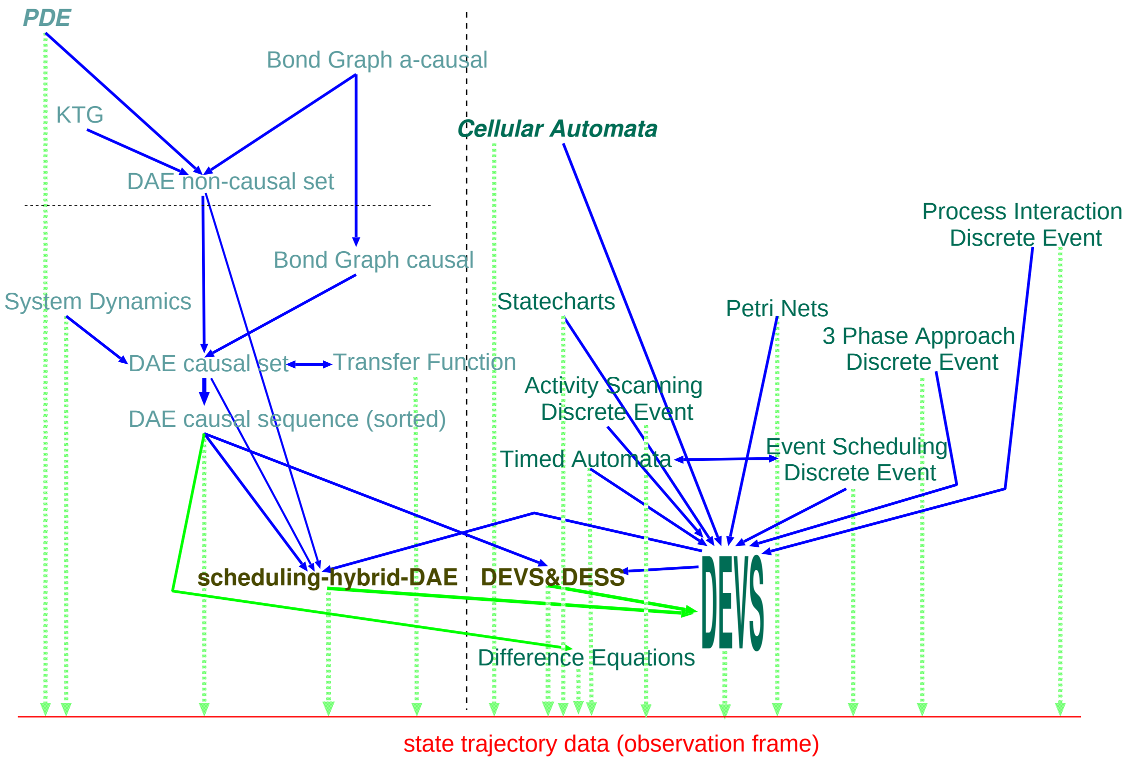

Figure 2.1: Formalism Transformation Graph (FTG).

The main component of AToM^3 is the Kernel, responsible for loading, saving, creating and manipulating models (at any meta-level, with the *Graph Rewriting Processor* and graph grammar models), as well as for generating code from the (meta-)$^+$models.

The ER formalism extended with constraints is available at the meta-meta-level. As stated before, it is perfectly possible to define other meta-formalisms using ER. Constraints can be specified as OCL or Python expressions, and are associated with events (similar to event-programming systems such as Visual Basic). The designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be related either to the *abstract syntax* (such as editing an attribute, connecting two entities, etc.) or *purely graphical* (such as dragging, dropping, etc.) If the constraint associated with an event is evaluated to *false*, the event is cancelled, or undone if the constraint was a post-condition. Constraints can be either associated with entities (*local constraints*) or with the whole model (*global constraints*). AToM^3, can also define *Actions*, similar to *Constraints* but with side-effects.

When modelling at the (meta-)$^+$level, the entities that may appear in a model must be specified together with their attributes (and constraints and actions as stated before). For example, to define the Petri Net Formalism, it is necessary to define both *Places* and *Transitions*. Furthermore, the attributes *name* and *number of tokens* must be added to *Places*. The *name* attribute must be added to *Transitions*. The (meta-)$^+$information is used by the AToM^3 Kernel to generate some Python files, which, when loaded by the Kernel, allow the processing of models in the defined formalism.

One of the components of the generated files is a model of a part of the AToM^3 user interface. This user interface model follows the *"Buttons"* formalism, and has its own meta-model. Initially, this model represents the necessary buttons to create the entities defined in the formalism's meta-model. It can be modified by the user to include, for example, buttons to execute graph grammars on the current model.

In AToM^3, entities may have two kinds of attributes: *regular* and *generative*. *Regular* attributes are used to identify characteristics of the current entity. *Generative* attributes are used to generate new attributes at a lower meta-level. The generated attributes may be generative in their own right. Both types of attributes may contain data or code for pre- and post-conditions.

Entities are connected by means of ports, which can be *named* or *unnamed*. An entity may have both types of ports. Unnamed ports are used when all the connections are semantically equal and there is no need to distinguish them. A typical example is Statecharts, in which *states* have unnamed ports to connect to other *states*. *Named* ports are used when there exists different meanings for the same types of connections. A typical example of this are the entities in the CBD formalism, where some entities represent functions to which other entities may be connected, representing the function's parameters. One needs to know exactly which parameter corresponds to each connection. For example, an *INTEGRAL* block has two parameters: the initial condition and the signal to be integrated. If a block is connected to an *INTEGRAL*, it must be known if this connection is to be interpreted as the initial condition or as the value to be integrated. This way, two named ports are needed for the *INTEGRAL* block to store the connections to each parameter.

In the meta-model, it is also possible to specify the graphical appearance of each entity of the defined formalism. This appearance is, in fact, a special kind of *generative* attribute. Objects' graphical appearance can be icon-like or arrow-like with optional icon decorations in the center, segments and extremes. For example, for Statecharts, *States* can be represented as ovals with the *name* inside the oval. *Transitions* are arrow-like drawings with the *events*, *conditions* and *actions* besides them. In general, semantic attributes are displayed graphically. Constraints and actions can also be associated with the graphical entities. Each graphical form, part of the graphical entity, can be referenced by an automatically generated name that has methods to change its graphical properties (colour, visibility, etc.) That is, in AToM^3, graphical manipulations must be explicitly specified by the user by means of constraints expressed in Python. This is in contrast with other approaches [6] in which constraint languages for graphical layout are used. Python constraints have the drawback of being at a lower abstraction level than a constraint language, but they are usually more efficient.

2.3.1 Graph Transformation in AToM^3

Graph grammars [57] are a generalization of Chomsky grammars, for graphs. They are composed of rules; each having graphs on their left and right hand sides (LHS and RHS). Rules are evaluated against an input graph (called host graph). If a matching is found between the LHS of a rule and a zone in the graph, then the rule can be applied. When a rule is applied, the matching subgraph of the host graph is replaced by the RHS of the rule. Rules can have applicability conditions, as well as actions to be performed when the rule is applied. Some graph rewriting systems have control mechanisms to decide which rule should be checked next. In AToM^3, rules are ordered according to a priority, and are checked from higher to lower priority. After the application of a matching rule, the system again tries to match, starting from the higher priority rule in the list. The graph grammar execution ends when no more applicable rules are found.

Model manipulations can be expressed in AToM^3, either as Python programs or as graph grammar models. The latter has the advantage of being a higher-level, natural, visual, declarative and formal notation. This makes computations become models, easier to specify, understand, and maintain and frees the user of knowing AToM^3 implementation details. The kind of model manipulations of interest include model execution, model optimization (for example, reducing its complexity), model transformation into another formalism, and code generation. The latter can be seen as a special case of formalism transformation. As a drawback, the use of graph grammars is constrained

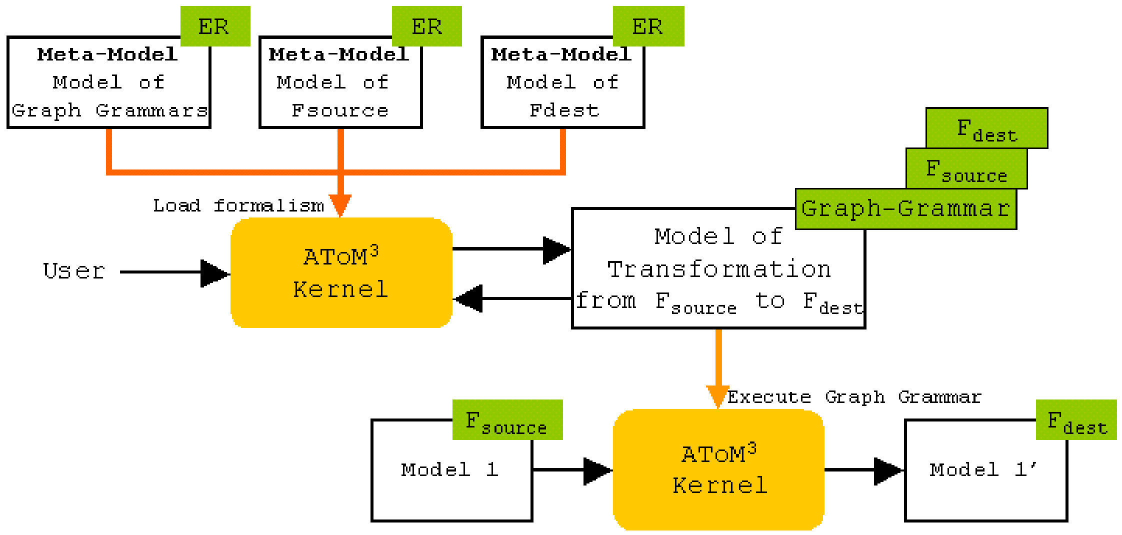

Figure 2.2: Model Transformation in AToM^3.

by efficiency as in the most general case, subgraph isomorphism testing is NP-complete. However, the use of small subgraphs on the LHS of graph grammar rules, as well as using node and edge types and attributes can greatly reduce the search space.

In Figure 2.2, a transformation of a model between two formalisms (F$_{source}$ and F$_{dest}$) has been depicted. To convert a model from formalism F_{source} to F_{dest} it is necessary to use the meta-models for both F_{source} and F_{dest}, together with the meta-model for graph grammars.

Graph grammars can be graphically edited (as any other model), as shown in Figure 2.3, in AToM^3. The image shows a moment in the editing of the RHS of a rule. The graph grammar is composed of three rules (see dialog window to the left). Note how in AToM^3 graph grammars can have actions to be executed before and after the graph grammar execution. The next dialog window to the right shows the information about the third rule, named "gen_OOCSMP_Outputs_CBD". In AToM^3 a rule has a *name*, a *priority*, a *time delay*, a flag for *subtype matching*, textual *conditions* and *actions* (expressed in Python) and LHS and RHS models. The *time delay* flag is used if the graph grammar is executed in *animation* mode. Note how this value can be changed by the actions of the rules. Other execution modes for graph grammars are *step-by-step* and *continuous*. The *subtype matching* flag is used in the matching process to specify either an exact type matching between the nodes of the LHS and the nodes in the host graph or a *"subtype matching"*. In the latter case, nodes (or connections) in the LHS and in the host graph do not need to have the same type. However, AToM^3 checks at run-time whether the node (or the connection) in the host graph has at least the same set of attributes as the node in the LHS. That is, if the node in the host graph is a structural subtype of the node in the LHS. Subtyping relationships in the meta-models do not need to be expressed. This relationship is found at run-time. This can be useful since one can write very general graph grammars, and reuse them for many formalisms.

The next dialog window to the right of the previous one shows the RHS of the rule being edited. Nodes and edges in LHS and RHS are provided with numbers (the entity displayed has been labelled "1") in such a way that if a number appears in both LHS and RHS, the node is not deleted. If it appears in the LHS, but not in the RHS, the node is deleted. Finally, if the number appears in the RHS but not in the LHS, the corresponding node is created. Nodes and connections in the LHS must be provided with the attribute values that will make a match with nodes and connections in the host graph. Any value of these attributes will make a match, or a specific value can be set.

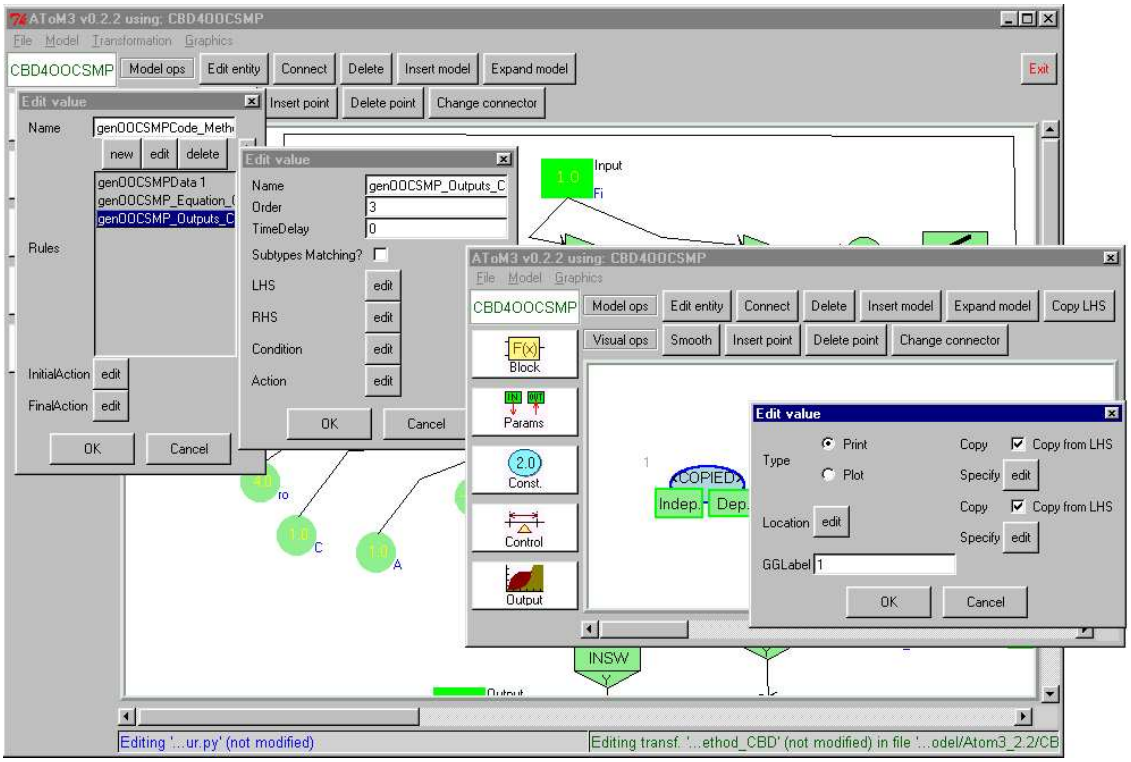

Figure 2.3: Editing a Graph Grammar in AToM^3.

Attributes of nodes and connections in the RHS of a rule, can either maintain the value (check-box labelled as “Copy from LHS” in the right-most dialog window in the figure), receive a specific value (specified in the different widgets in the dialog window, depending on the attribute’s type), or calculate a new one by means of Python code (button labelled as “Specified”). This code can use other node and connection attributes.

3

Modelling Statecharts with AToM^3

Modelling any formalism in AToM^3 first requires that a meta-model is created. Next, syntactic (or visual) constraints are added to the Statechart meta-model. These constraints fine tune the meta-model to only reflect valid Statechart models. Once the meta-model is complete, it is possible to automatically generate a Statechart model editor. There are several ways of specifying the operational semantics of the meta-modelled formalism. One way is to explicity model operational semantics by means of graph grammars. Another is to build a code generator. The latter method was used and is examined in a later chapter. This chapter only concerns itself with the Statechart environment and building Statechart models.

3.1 Statechart Meta-model

Figure 3.1 displays the Statechart meta-model in the `Entity Relationship` formalism created using AToM^3. From this diagram it is easy to see the hierarchical nature of Statecharts. A `Composite` state is related to itself via the `contains` relation. This means a `Composite` state may be contained within another `Composite` state.

Hyperedges connect all states to themselves. Thus, a hyperedge can connect `Basic` to `Basic`, `Basic` to `Composite`, `Composite` to `Composite`, etc. Recall that a `History` node is not actually a state. An execution environment cannot end a macro-step at a history node. A history node is a section of a transition. The fact that we have allowed transitions to emanate from a history is a short-hand notation. This notation allows the modeller to specify an alternate history state if no history information is available.

`Orthogonal` nodes may be related to a `Composite` via the `orthogonality` relation. This design decision makes it easier to add constraints which restrict only valid Statecharts. The `orthogonality` relation is essentially the same as the `contains` relation. Notice that an `orthogonal` may be contained within a `Composite` via `orthogonality` but can only contain a `Composite`, `Basic` or `History`. This formulation forces the strict AND-OR layering required by Statecharts.*

*Recall that the AND-OR requirement means an orthogonal cannot be contained within an orthogonal.

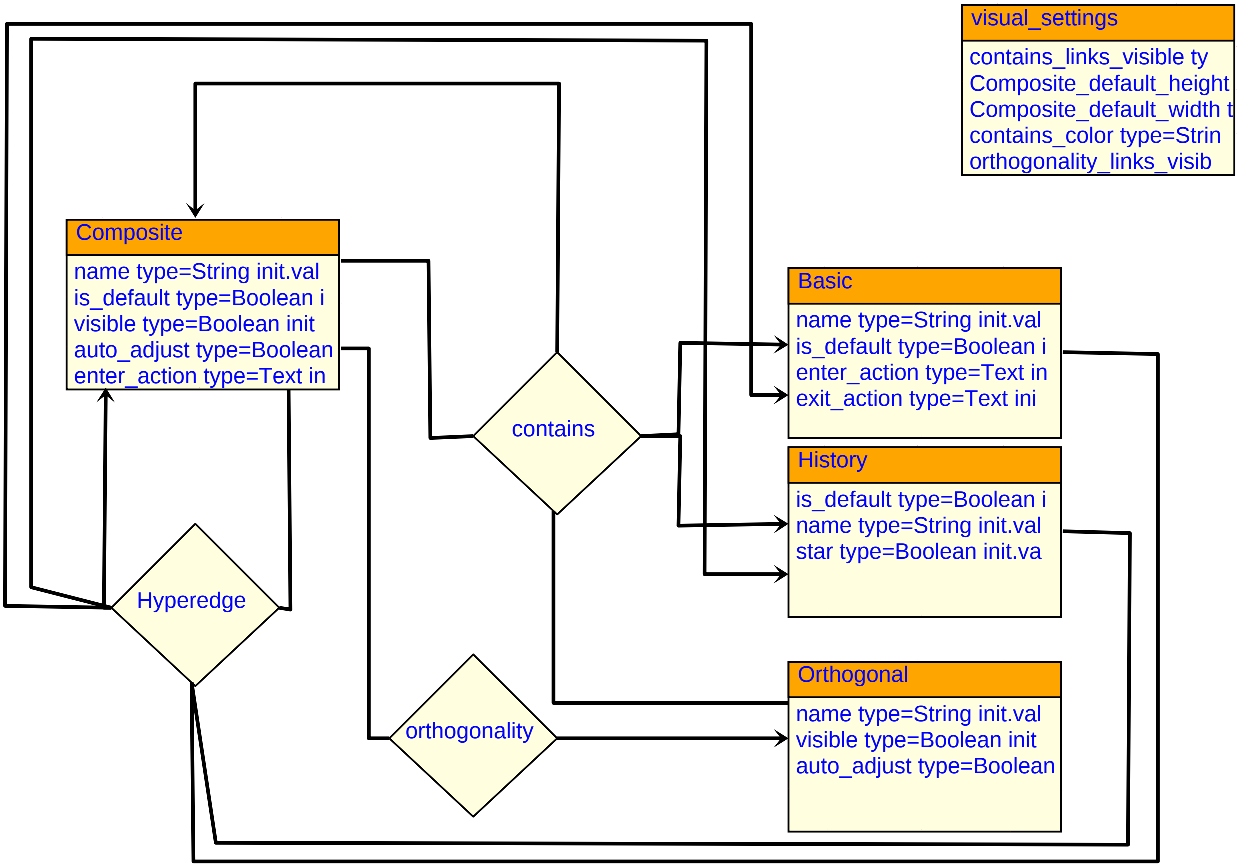

Figure 3.1: Statecharts Meta-Model in AToM^3.

Table 3.1: Statechart Meta-Model Cardinalities

Relation	Source, Cardinality	Destination, Cardinality
`contains`	`Composite`, 1	`Composite`, N
`contains`	`Composite`, 1	`Basic`, N
`contains`	`Composite`, 1	`History`, 1
`orthogonality`	`Orthogonal`, 1	`Composite`, N
`orthogonality`	`Orthogonal`, 1	`Basic`, N
`orthogonality`	`Orthogonal`, 1	`History`, 1
`Hyperedge`	`Composite`, N	`Basic`, N
`Hyperedge`	`Composite`, N	`Composite`, N
`Hyperedge`	`Composite`, N	`History`, N
`Hyperedge`	`Basic`, N	`Basic`, N
`Hyperedge`	`Basic`, N	`Composite`, N
`Hyperedge`	`Basic`, N	`History`, N
`Hyperedge`	`History`, 1	`Basic`, N
`Hyperedge`	`History`, 1	`Composite`, N
`Hyperedge`	`History`, 1	`History`, N

Table 3.2: Statechart Meta-Model Constraints

Entity/Relationship	Constraint
`Composite`	Must have exactly one default state.
`Composite,Orthogonal`	Can have a maximum of one `History` node.
`contains`	If A is a node instance, then $A \rightarrow A$ is not valid (where $\rightarrow$ is a `contains` relation and A is `Composite`).
`contains`	If $A \rightarrow B$ then $B \rightarrow A$ is not valid (where $\rightarrow$ is a `contains` relation and A & B are `Composite`).
`Hyperedge`	If a `Hyperedge` instance's source and/or destination cardinalities are greater than 1, the instance must have each respective source and destination node as a child of different orthogonal components.

3.1.1 Cardinalities

Table 3.1.1 indicates the allowed cardinalities between objects in the Meta-Model. The cardinalities are a way of restricting only the creation of valid Statechart models. Here, the cardinalities will not be specified as in AToM^3, but in a slightly more intuitive manner.

3.1.2 Constraints

Table 3.2 illustrates the constraints used to enforce the creation of valid Statecharts. The meanings of the constraints are described in plain English.

3.2 The Environment

Once the Statechart meta-model is complete, it becomes possible to automatically generate an environment which is capable of drawing Statecharts. This environment can be seen in figure 3.2.

Notice there are buttons for adding an object corresponding to each entity in the meta-model. Composite and Basic states can be added. In order to specify that one state is a sub-state of another,

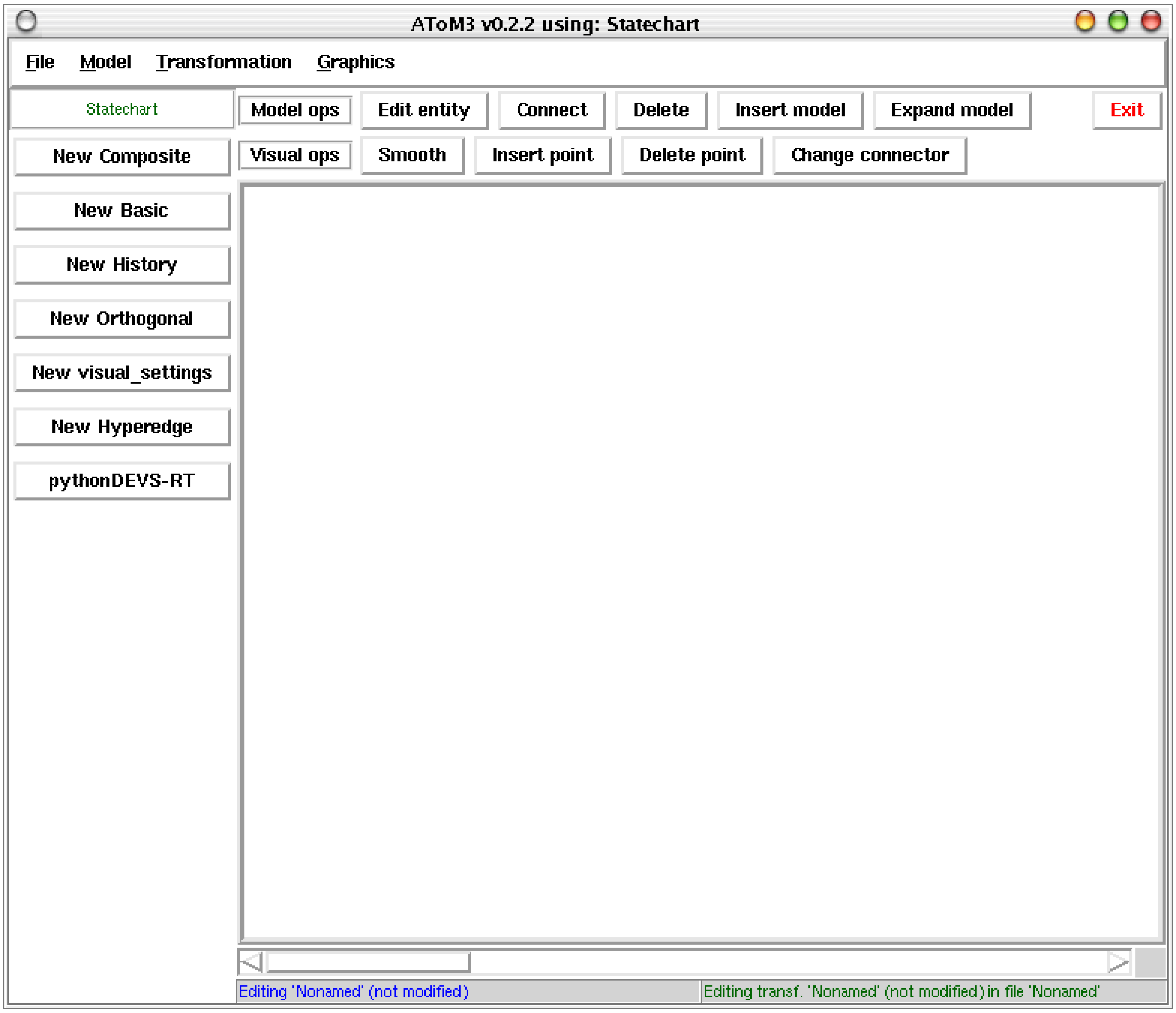

Figure 3.2: The Statechart environment in AToM3.

they must be connected by first clicking the connect button (on the top toolbar), then clicking on the parent and child states. This sequence of steps indicates that one wishes to connect these two entities using one of the relations in the meta-model. There are two ways to relate states in the meta-model: the contains relation and the Hyperedge relation. Thus, a dialog will present an option to the user to specify which relation to use.

Clicking the edit button and then clicking on an entity displays a dialog in which the user can modify the meta-model attributes for that entity. For a state, these attributes include the name of the state, enter action, exit action and a boolean value indicating if the state is default. A history node has a name attribute as well as a boolean flag indicating H or $H^\star$.

The user may also put `Orthogonal` regions on the canvas. `Orthogonal` regions may be connected via the `orthogonality` relation to a `Composite`. The `orthogonality` relation is basically the same a the `contains` relation. Two or more `Orthogonal` regions must be connected to a `Composite`, as indicated in the table of constraints. Having only one `Orthogonal` region within a `Composite` does not represent a valid Statechart.

Notice that there exists a button for creating a new `Hyperedge`. This is probably never needed while building Statechart models. However, it is included mainly for building graph grammars to convert Statechart models into DEVS models.

The final button is `PythonDEVS-RT`. This is used to perform the actual conversion from Statecharts to DEVS. This action will output a file which contains a valid DEVS model. The model can be executed in real-time using PythonDEVS-RT.

3.3 Statechart Attributes

In order to properly set the Statechart attributes, the user must know how the attributes are used. A description of the attributes used in the Statechart meta-model is given in the tables below as follows.

- `Basic` & `Composite`: Table 3.3
- `Orthogonal`: Table 3.4
- `History`: Table 3.5
- `visual_settings`: Table 3.6
- `Hyperedge`: Table 3.7

Table 3.3: Statechart meta-model attributes for the `Basic` and `Composite` entities.

Attribute Name	**Description**
`name`	The name of the state.
`is_default`	Boolean value indicates if state is default.
`enter_action`	Valid Python code.
`exit_action`	Valid Python code.

Table 3.4: Statechart meta-model attributes for the `Orthogonal` entity.

Attribute Name	**Description**
`name`	The name of the orthogonal state.

Table 3.5: Statechart meta-model attributes for the `History` node.

Attribute Name	Description
`name`	The name of the history node.
`star`	Boolean indicates if node is H (un-set) or $H^{\star}$ (set).

Table 3.6: Statechart meta-model attributes for the `visual_settings` entity.

Attribute Name	Description
`contains_links_visible`	Boolean which makes the `contains` relation visible (set) or invisible (unset).
`orthogonality_links_visible`	Boolean which makes the `orthogonality` relation visible (set) or invisible (unset).
`Hyperedge_links_visible`	Boolean which makes the `Hyperedge` relation visible (set) or invisible (unset).
`Composite_default_height`	The default height of a `Composite` node.
`Composite_default_width`	The default width of a `Composite` node.
`contains_color`	The color of the `contains` relation.
`orthogonality_color`	The color of the `orthogonality` relation.
`Hyperedge_color`	The color of the `Hyperedge` relation.
`Composite_color`	The color of the `Composite` entity.
`Orthogonal_color`	The color of the `Orthogonal` entity.

3.4 Statechart Operational Semantics

Graph Grammars were not used to model the operational semantics of the Statechart models. Instead a code generator, written in Python, converts the Statechart models into a DEVS model. The resulting DEVS model will represent the Statechart as well as the Statechart's operational semantics. This procedure is described in the coming chapters.

3.5 General Usage

This section outlines the general usage of the Statechart editor tool. It will illustrate the basics of building a Statechart model and important things to remember about scoping and broadcasting.

3.5.1 Adding Entities and Relations

To add an entity on the canvas, simply click the entity's button on the left tool bar, then click on the canvas. The entity will be created. There is no need to create hyperedges in this way. Hyperedge, contains and orthogonality relations are created using the "Connect" button on the top toolbar. Click the source entity then the destination entity and the relation will be created automatically. In some cases, the modeller will be presented with a dialog box as in figure 3.3. This allows to choose between a `contains` relation or a `Hyperedge` relation when the context of the connection is ambiguous. This is always the case when connecting `Composite` nodes since they may be joined by a hyperedge or one may be contained within the other.

Default Colors

The `visual_settings` entity can be used to modify the colors and visibility of certain entities and relations. However, the modeller should understand the default colors used since they denote

Table 3.7: Statechart meta-model attributes for the `Hyperedge` relation.

Attribute Name	Description
`name`	The name of the `Hyperedge` relation
`trigger`	A string containing the name of an event or a string of the form: AFTER(X) where X denotes a valid Python expression, which evaluates to a number.
`guard`	A string which is a valid Python boolean expression or a string of the form IN(X) where X is a string of the form A.B.C...Z. Each letter is the name of a state. Altogether, X is a path from root to destination. The destination (Z) is the referenced state.
`action`	Valid Python code to be executed when the transition fires.
`broadcast`	Python code to build an object to be broadcast. Use 'return X' as the last statement to indicate X as the broadcasted event. Should return `None` or an instance of `DEVSevent`
`display`	A string to be displayed on the canvas.

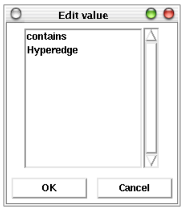

Figure 3.3: A dialog to choose between the `contains` and `Hyperedge` relations.

important information.

- Default state: Green
- Orthogonal region: Gray
- Non-default state: Blue

3.5.2 Editing and Deleting Entities and Relations

To edit the attributes of anything on the canvas, click the "Edit Entity" button on the top toolbar then on the entity or relation one wishes to edit. A dialog will be presented with all the attributes for that entity or relation as indicated in tables 3.3 to 3.7. To remove an entity or relation a similar action is taken. Click on the "Delete" button on the top toolbar, then click on the entity or relation to delete.

Following is a list of figures illustrating the edit dialogs for their respective entity or relation.

- State: Figure 3.4
- Hyperedge: Figure 3.5
- History: Figure 3.6
- Orthogonal: Figure 3.7

3.5.3 Broadcasting Events

The `broadcast` attribute of a `Hyperedge` relation is used for broadcasting or narrowcasting events. This attribute is a string value and represents a Python function. Thus, the modeller can use the

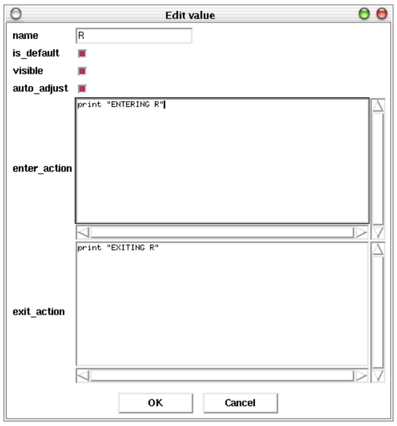

Figure 3.4: The state edit dialog.

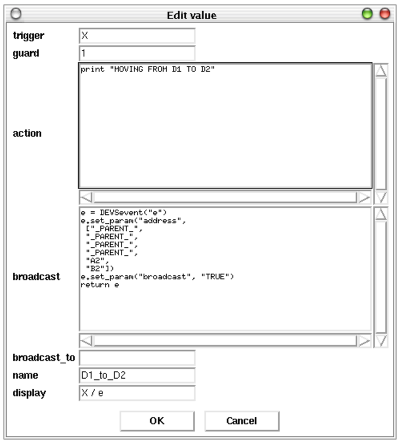

Figure 3.5: The hyperedge edit dialog.

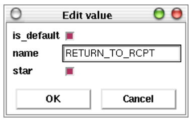

Figure 3.6: The history edit dialog.

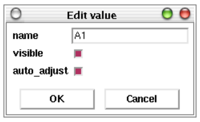

Figure 3.7: The orthogonal edit dialog.

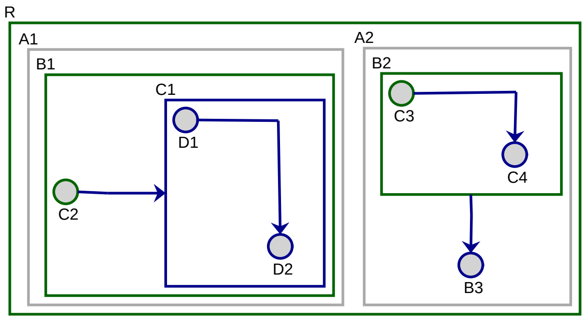

Figure 3.8: An example Statechart.

Python `return` statement to return the event to be broadcast. If no event is to be broadcast, then `None` should be returned. Otherwise, an instance of class `DEVSevent` must be returned.

The modeller must use the special attributes of the `DEVSevent` to indicate how the event is to be narrowcast and where this narrowcasting will occur. The `address` parameter is used to indicate the final destination of the event. The address parameter is a list of strings which are names of DEVS ports. One can use the names of children as well as the special name `_PARENT_` (outputs to the $\perp$ or parent port).

If the modeller wishes this event to be broadcast from its final destination point, then they should set the `broadcast` parameter to the string value `TRUE`.

For example, consider the Statechart in figure 3.8. The following specification of the `broadcast` parameter for the hyperedge going from $D1$ to $D2$ would cause the event e to be narrowcasted within $B2$.

```
e = DEVSevent("e")
e.set_param("address",
  ["\_PARENT\_",
  "\_PARENT\_",
  "\_PARENT\_",
  "\_PARENT\_",
  "A2",
  "B2"])
e.set_param("broadcast", "TRUE")
return e
```

Notice that the `address` parameter is a virtual path, not a path from the root. That is, the path starts from the source state of the transition.

3.5.4 Scoping

The scope of any action can be considered as local. That is, the modeller will not know any names defined outside of its own local scope. Thus, if persistent variables are needed, they should be defined using dynamic scoping. The modeller can easily put a variable or function into the global scope and then access it later from another action.

3.5.5 Additional Information

More information regarding the Statecharts environment in AToM^3 can be found in Appendix B.

4
The DEVS Formalism

The DEVS formalism was conceived by Zeigler [71, 72] to provide a rigorous common basis for discrete-event modelling and simulation. For the class of formalisms denoted as *discrete-event* [50], system models are described at an abstraction level where the time base is continuous ($\mathbb{R}$), but during a bounded time-span, only a *finite number* of relevant *events* occur. These events can cause the state of the system to change. In between events, the state of the system does *not* change. This is unlike *continuous* models in which the state of the system may change continuously over time. As an extension of Finite State Automata, the DEVS (Discrete Event Systems) formalism captures *concepts* from Discrete Event simulation. As such it is a sound basis for meaningful model exchange in the Discrete Event realm. The following specification was originally written by Hans Vangheluwe.

4.1 The DEVS Formalism

The DEVS formalism fits the general structure of deterministic, causal systems in classical systems theory. DEVS allows for the description of system behaviour at two levels. At the lowest level, an *atomic DEVS* describes the autonomous behaviour of a discrete-event system as a sequence of deterministic transitions between sequential states as well as how it reacts to external input (events) and how it generates output (events). At the higher level, a *coupled DEVS* describes a system as a *network* of coupled components. The components can be atomic DEVS models or coupled DEVS in their own right. The connections denote how components influence each other. In particular, output events of one component can become, via a network connection, input events of another component. It is shown in [71] how the DEVS formalism is *closed under coupling*: for each coupled DEVS, a *resultant* atomic DEVS can be constructed. As such, any DEVS model, be it atomic or coupled, can be replaced by an equivalent atomic DEVS. The construction procedure of a resultant atomic DEVS is also the basis for the implementation of an *abstract simulator* or solver capable of simulating any DEVS model. As a coupled DEVS may have coupled DEVS components, *hierarchical* modelling is supported.

In the following, the different aspects of the DEVS formalism are explained in more detail.

4.1.1 The Atomic DEVS Formalism

The atomic DEVS formalism is a structure describing the different aspects of the discrete-event behaviour of a system:

$$atomicDEVS \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle.$$

The *time base* T is continuous and is not mentioned explicitly

$$T = R.$$

The *state set* S is the set of admissible *sequential states*: the DEVS dynamics consists of an ordered sequence of states from S. Typically, S will be a *structured* set (a product set)

$$S = \times_{i=1}^{n} S_i.$$

This formalizes multiple (n) *concurrent* parts of a system. It is noted how a structured state set is often synthesized from the state sets of concurrent components in a coupled DEVS model.

The time the system *remains* in a sequential state before making a transition to the next sequential state is modelled by the *time advance function*

$$ta : S \to R^{+}_{0,+\infty}.$$

As time in the real world always advances, the image of ta must be non-negative numbers. $ta = 0$ allows for the representation of *instantaneous* transitions: no time elapses before transition to a new state. Obviously, this is an abstraction of reality which may lead to simulation *artifacts* such as infinite instantaneous loops which do not correspond to real physical behaviour. If the system is to stay in an end-state s *forever*, this is modelled by means of $ta(s) = +\infty$.

The internal transition function

$$\delta_{int} : S \to S$$

models the transition from one state to the next sequential state. δ_{int} describes the behaviour of a Finite State Automaton; ta adds the progression of time.

It is possible to *observe* the system output. The output set Y denotes the set of admissible *outputs*. Typically, Y will be a *structured* set (a product set)

$$Y = \times_{i=1}^{l} Y_i.$$

This formalizes multiple (l) output ports. Each port is identified by its unique index i. In a user-oriented modelling language, the indices would be derived from unique port *names*.

The output function

$$\lambda : S \to Y \cup \{\phi\}$$

maps the internal state onto the output set. Output events are *only* generated by a DEVS model at the time of an *internal* transition. At that time, the state *before* the transition is used as input to λ. At all other times, the non-event ϕ is output.

To describe the *total state* of the system at each point in time, the sequential state $s \in S$ is not sufficient. The *elapsed* time e since the system made a transition to the current state s needs also to be taken into account to construct the total state set

$$Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$$

The elapsed time e takes on values ranging from 0 (transition just made) to $ta(s)$ (about to make transition to the next sequential state). Often, the *time left* σ in a state is used:

$$\sigma = ta(s) - e$$

Up to now, only an *autonomous* system was described: the system receives no external inputs. Hence, the *input set* X denoting all admissible input values is defined. Typically, X will be a *structured* set (a product set)

$$X = \times_{i=1}^{m} X_i$$

This formalizes multiple (m) input ports. Each port is identified by its unique index i. As with the output set, port indices may denote *names*.

The set Ω contains all admissible input segments ω

$$\omega : T \to X \cup \{\phi\}$$

In discrete-event system models, an input segment generates an input *event* different from the *non-event* ϕ only at a finite number of instants in a bounded time-interval. These *external events*, inputs x from X, cause the system to interrupt its autonomous behaviour and react in a way prescribed by the external transition function

$$\delta_{ext} : Q \times X \to S$$

The reaction of the system to an external event depends on the sequential state the system is in, the particular input *and* the elapsed time. Thus, δ_{ext} allows for the description of a large class of behaviours typically found in discrete-event models (including synchronization, preemption, suspension and re-activation).

When an input event x to an atomic model is not listed in the δ_{ext} specification, the event is *ignored*.

In Figure 4.1, an example state trajectory is given for an atomic DEVS model. In the figure, the system made an internal transition to state $s2$. In the absence of external input events, the system stays in state $s2$ for a duration $ta(s2)$. During this period, the elapsed time e increases from 0 to $ta(s2)$, with the total state $= (s2, e)$. When the elapsed time reaches $ta(s2)$, first an output is generated: $y2 = \lambda(s2)$, then the system transitions instantaneously to the new state $s4 = \delta_{int}(s2)$. In autonomous mode, the system would stay in state $s4$ for $ta(s4)$ and then transition (after generating output) to $s1 = \delta_{int}(s4)$. Before e reaches $ta(s4)$ however, an external input event x arrives. At that time, the system forgets about the scheduled internal transition and transitions to $s3 = \delta_{ext}((s4, e), x)$. Note how an external transition does not give rise to an output. Once in state $s3$, the system continues in autonomous mode.

As an example atomic DEVS, consider the model of two traffic lights depicted in Figure 4.2. In autonomous mode, the light transition in intuitive fashion. If the “switch to manual” (M) external event is received, lights in both directions blink yellow. If the “switch to automatic” event is received, the system switches back deterministically to state RY to resume autonomous mode. The atomic DEVS representation is given below.

$$DEVS = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

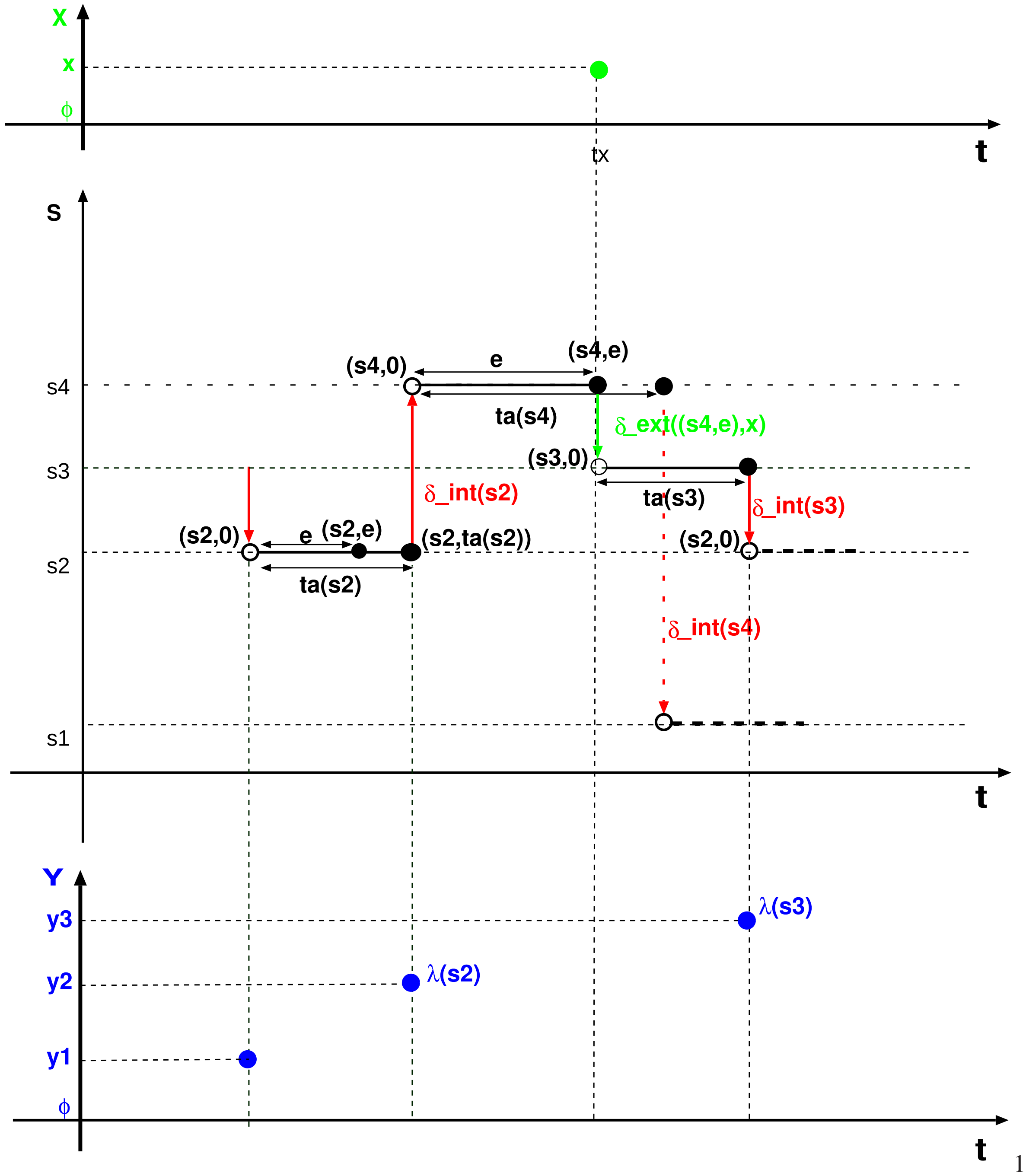

Figure 4.1: State Trajectory of a DEVS-specified Model

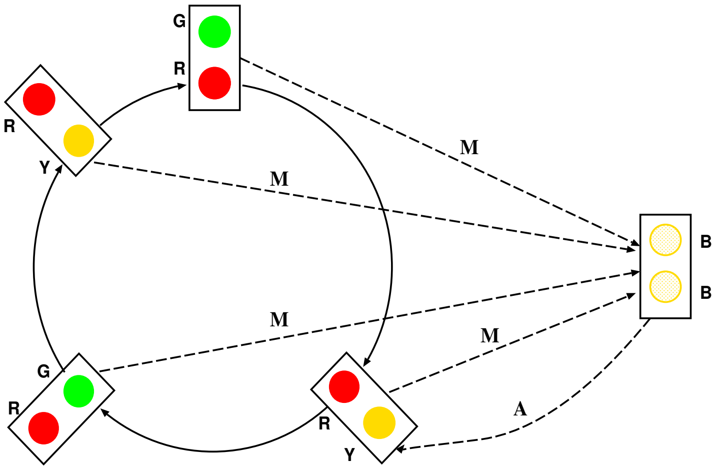

Figure 4.2: Traffic light example

$$
\begin{aligned}
&T = R\\
&X = \{M, A\}\\
&\omega : T \rightarrow X \cup \{\phi\}\\
&S = \{RG, RY, GR, YR, BB\}\\
&\delta_{int}(RG) = RY;\ \delta_{int}(RY) = GR\\
&\delta_{int}(GR) = YR;\ \delta_{int}(YR) = RG\\
&ta(RG) = 60s;\ ta(RY) = 10s\\
&ta(GR) = 50s;\ ta(YR) = 10s\\
&ta(BB) = +\infty\\
&\delta_{ext}((RG, e), M) = BB\\
&\delta_{ext}((RY, e), M) = BB\\
&\delta_{ext}((GR, e), M) = BB\\
&\delta_{ext}((YR, e), M) = BB\\
&\delta_{ext}((BB, e), A) = RY\\
&Y = \{GREY, YELLOW, BLINK\}\\
&\lambda(RG) = \lambda(RY) = \lambda(GR) = GREY\\
&\lambda(YR) = YELLOW\\
&\lambda(BB) = BLINK
\end{aligned}
$$

4.1.2 The Coupled DEVS Formalism

The coupled DEVS formalism describes a discrete-event system in terms of a network of coupled components.

$$coupledDEVS \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle$$

The component $self$ denotes the coupled model itself. X_{self} is the (possibly structured) set of allowed external inputs to the coupled model. Y_{self} is the (possibly structured) set of allowed (external) outputs of the coupled model. D is a set of unique component references (names). The coupled model itself is referred to by means of $self$, a unique reference not in D.

The set of *components* is

$$\{M_i | i \in D\}.$$

Each of the components must be an atomic DEVS

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i\rangle, \forall i \in D.$$

The set of *influencees* of a component, the components influenced by $i \in D \cup \{self\}$, is I_i. The set of all influencees describes the coupling network structure

$$\{I_i | i \in D \cup \{self\}\}.$$

For modularity reasons, a component (including $self$) may not influence components outside its scope –the coupled model–, rather only other components of the coupled model, or the coupled model $self$:

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}.$$

This is further restricted by the requirement that none of the components (including $self$) may influence themselves directly as this could cause an instantaneous dependency cycle (in case of a 0 time advance inside such a component) akin to an algebraic loop in continuous models:

$$\forall i \in D \cup \{self\} : i \notin I_i.$$

Note how one can always encode a self-loop ($i \in I_i$) in the internal transition function.

To translate an output event of one component (such as a departure of a customer) to a corresponding input event (such as the arrival of a customer) in influencees of that component, *output-to-input translation functions* $Z_{i,j}$ are defined:

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\},$$

$$\begin{array}{rcll} Z_{self,j} & : & X_{self} \rightarrow X_j & , \forall j \in D, \\ Z_{i,self} & : & Y_i \rightarrow Y_{self} & , \forall i \in D, \\ Z_{i,j} & : & Y_i \rightarrow X_j & , \forall i, j \in D. \end{array}$$

Together, I_i and $Z_{i,j}$ completely specify the coupling (structure and behaviour).

As a result of coupling of concurrent components, multiple state transitions may occur at the same simulation time. This is an artifact of the discrete-event abstraction and may lead to behaviour not related to real-life phenomena. A logic-based foundation to study the *semantics* of these artifacts was introduced by Radiya and Sargent [56]. In sequential simulation systems, such transition *collisions* are resolved by means of some form of *selection* of which of the components' transitions should be handled first. This corresponds to the introduction of priorities in some simulation languages. The coupled DEVS formalism explicitly represents a *select* function for *tie-breaking* between simultaneous events:

$$select : 2^D \rightarrow D$$

select chooses a unique component from any non-empty subset E of D:

$$select(E) \in E.$$

The subset E corresponds to the set of all components having a state transition simultaneously.

4.1.3 Closure of DEVS under coupling

As mentioned before, it is possible to construct a *resultant* atomic DEVS model for each coupled DEVS. This *closure under coupling* of atomic DEVS models makes *any* coupled DEVS equivalent to an atomic DEVS. By induction, any *hierarchically* coupled DEVS can thus be flattened to an atomic DEVS. As a result, the requirement that each of the components of a coupled DEVS be an atomic DEVS can be relaxed to be atomic *or* coupled as the latter can always be replaced by an equivalent atomic DEVS.

The core of the closure procedure is the selection of the most *imminent* (ie soonest to occur) event from all the components' scheduled events [71]. In case of simultaneous events, the *select* function is used. In the sequel, the resultant construction is described.

From the coupled DEVS

$$\langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select\rangle,$$

with all components M_i atomic DEVS models

$$M_i = \langle S_i, ta_i, \delta_{int,i}, X_i, \delta_{ext,i}, Y_i, \lambda_i\rangle, \forall i \in D$$

the atomic DEVS

$$\langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda\rangle$$

is constructed.

The resultant set of sequential states is the product of the total state sets of all the components

$$S = \times_{i\in D} Q_i,$$

where

$$Q_i = \{(s_i, e_i) | s \in S_i, 0 \le e_i \le ta_i(s_i)\}, \forall i \in D.$$

The time advance function ta

$$ta : S \to R^+_{0,+\infty}$$

is constructed by selecting the *most imminent* event time, of all the components. This means, finding the smallest time *remaining* until internal transition, of all the components

$$ta(s) = min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}.$$

A number of *imminent* components may be scheduled for a *simultaneous* internal transition. These components are collected in a set

$$IMM(s) = \{i \in D | \sigma_i = ta(s)\}.$$

From IMM, a set of elements of D, *one* component i^* is chosen by means of the $select$ *tie-breaking* function of the coupled model

$$\begin{array}{rccc} select & : & 2^D & \to & D \\ & & IMM(s) & \to & i^* \end{array}$$

Output of the selected component is generated *before* it makes its internal transition. Note also how, as in a Moore machine, input does not directly influence output. In DEVS models, *only* an internal transition produces output. An input can only influence/generate output via an internal transition similar to the presence of *memory* in the form of integrating elements in continuous models. Allowing an external transition to produce output could lead to infinite instantaneous loops. This is equivalent to algebraic loops in continuous systems. The output of the component is translated into coupled model output by means of the coupling information

$$\lambda(s) = Z_{i^*,self}(\lambda_{i^*}(s_{i^*})), \text{if } self \in I_{i^*}.$$

If the output of i^* is not connected to the output of the coupled model, the non-event ϕ can be generated as output of the coupled model. As ϕ literally stands for no event, the output can also be ignored without changing the meaning (but increasing performance of simulator implementations).

The internal transition function transforms the different parts of the total state as follows:

$$\delta_{int}(s) = (\ldots,(s'_j,e'_j),\ldots), \text{ where}$$

$$\begin{aligned}(s'_j,e'_j) &= (\delta_{int,j}(s_j),0) && ,\text{for } j = i^*,\\ &= (\delta_{ext,j}(s_j,e_j+ta(s),Z_{i^*,j}(\lambda_{i^*}(s_{i^*}))),0) && ,\text{for } j \in I_{i^*},\\ &= (s_j,e_j+ta(s)) && ,\text{otherwise}.\end{aligned}$$

The selected imminent component i^* makes an internal transition to sequential state $\delta_{int,i^*}(s_{i^*})$. Its elapsed time is reset to 0. All the influencees of i^* change their state due to an external transition prompted by an input which is the output-to-input translated output of i^*, with an elapsed time adjusted for the time advance $ta(s)$. The influencees' elapsed time is reset to 0. Note how i^* is not allowed to be an influencee of i^* in DEVS. The state of all other components is not affected and their elapsed time is merely adjusted for the time advance $ta(s)$.

The external transition function transforms the different parts of the total state as follows:

$$\delta_{ext}(s,e,x) = (\ldots,(s'_i,e'_i),\ldots), \text{ where}$$

$$\begin{aligned}(s'_i,e'_i) &= (\delta_{ext,i}(s_i,e_i+e,Z_{self,i}(x)),0) && ,\text{for } i \in I_{self},\\ &= (s_i,e_i+e) && ,\text{otherwise}.\end{aligned}$$

An incoming external event is routed, with an adjustment for elapsed time, to each of the components connected to the coupled model input (after the appropriate input-to-input translation). For all those components, the elapsed time is reset to 0. All other components are not affected and only the elapsed time is adjusted.

Some limitations of DEVS are that

- a conflict due to simultaneous internal and external events is resolved by ignoring the internal event. It should be possible to explicitly specify behaviour in case of conflicts;
- there is limited potential for parallel implementation;
- the *select* function is an artificial legacy of the semantics of traditional sequential simulators based on an event list;
- it is not possible to describe variable structure.

Some of these are compensated for in parallel DEVS ([16]).

4.1.4 Implementation of a DEVS Solver

The algorithm in Figure 4.3 is based on the closure under coupling construction and can be used as a specification of a –possibly parallel– implementation of a DEVS solver or "abstract simulator" [71, 41]. In an atomic DEVS solver, the last event time t_L as well as the local state s are kept. In a coordinator, only the last event time t_L is kept. The next-event-time t_N is sent as output of either solver. It is possible to also keep t_N in the solvers. This requires consistent (recursive) initialization of the t_Ns. If kept, the t_N allows one to check whether the solvers are appropriately synchronized. The operation of an abstract simulator involves handling four types of messages. The $(x, from, t)$ message carries external input information. The $(y, from, t)$ message carries external output information. The $(*, from, t)$ and $(done, from, t_N)$ messages are used for scheduling (synchronizing) the abstract simulators. In these messages, t is the simulation time and t_N is the next-event-time. The $(*, from, t)$ message indicates an internal event $*$ is due.

When a coordinator receives a $(*, from, t)$ message, it selects an imminent component i^* by means of the tie-breaking function *select* specified for the coupled model and routes the message to i^*. Selection is necessary as there may be more than one imminent component (with minimum next

message m	**simulator**	**coordinator**
$(*, from, t)$	simulator correct only if $t = t_N$	
	$y \leftarrow \lambda(s)$	**send** $(*, self, t)$ to i^*, where
	if $y \neq \phi$:	$i^* = select(imm\_children)$
	send $(\lambda(s), self, t)$ to *parent*	$imm\_children = \{i \in D \mid M_i.t_N = t\}$
	$s \leftarrow \delta_{int}(s)$	$active\_children \leftarrow active\_children \cup \{i^*\}$
	$t_L \leftarrow t$	
	$t_N \leftarrow t_L + ta(s)$	
	send $(done, self, t_N)$ to *parent*	
$(x, from, t)$	simulator correct only if $t_L \leq t \leq t_N$ (ignore δ_{int} to resolve a $t = t_N$ *conflict*)	
	$e \leftarrow t - t_L$	$\forall i \in I_{self}$:
	$s \leftarrow \delta_{ext}(s, e, x)$	**send** $(Z_{self,i}(x), self, t)$ to i
	$t_L \leftarrow t$	$active\_children \leftarrow active\_children \cup \{i\}$
	$t_N \leftarrow t_L + ta(s)$	
	send $(done, self, t_N)$ to *parent*	
$(y, from, t)$		$\forall i \in I_{from} \setminus \{self\}$:
		send $(Z_{from,i}(y), from, t)$ to i
		$active\_children \leftarrow active\_children \cup \{i\}$
		if $self \in I_{from}$:
		send $(Z_{from,self}(y), self, t)$ to *parent*
$(done, from, t)$		$active\_children \leftarrow active\_children \setminus \{from\}$
		if $active\_children = \emptyset$:
		$t_L \leftarrow t$
		$t_N \leftarrow min\{M_i.t_N \mid i \in D\}$
		send $(done, self, t_N)$ to *parent*

Figure 4.3: DEVS Simulation Procedure

$t \leftarrow t_N$ of topmost coordinator
repeat until $t \geq t_{end}$ (or some other termination condition)
  send $(*, main, t)$ to topmost coupled model top
  wait for $(done, top, t_N)$
  $t \leftarrow t_N$

Figure 4.4: DEVS Simulation Procedure Main Loop

remaining time).
When an atomic simulator receives a $(*, from, t)$ message, it generates an output message $(y, from, t)$ based on the old state s. It then computes the new state by means of the internal transition function. Note how in DEVS, output messages are only produced while executing internal events. When a simulator outputs a $(y, from, t)$ message, it is sent to its parent coordinator. The coordinator sends the output, after appropriate output-to-input translation, to each of the influencees of i^* (if any). If the coupled model itself is an influencee of i^*, the output, after appropriate output-to-output translation, is sent to the the coupled model's parent coordinator.

When a coordinator receives an $(x, from, t)$ message from its parent coordinator, it routes the message, after appropriate input-to-input translation, to each of the affected components.
When an atomic simulator receives an $(x, from, t)$ message, it executes the external transition function of its associated atomic model.
After processing an $(x, from, t)$ or $(y, from, t)$ message, a simulator sends a $(done, from, t_N)$ message to its parent coordinator to prepare a new schedule. When a coordinator has received $(done, from, t_N)$ messages from all its components, it sets its next-event-time t_N to the minimum t_N of all its components and sends a $(done, from, t_N)$ message to its parent coordinator. This process is recursively applied until the top-level coordinator or *root coordinator* receives a $(done, from, t_N)$ message.

As the simulation procedure is synchronous, it does not support a-synchronously arriving (real-time) external input. Rather, the environment or Experimental Frame should also be modelled as a DEVS component.

To run a simulation experiment, the *initial conditions* t_L and s must first be set in *all* simulators of the hierarchy. If t_N is kept in the simulators, it must be recursively set too. Once the initial conditions are set, the main loop described in Figure 4.4 is executed.

5
The `pythonDEVS` Simulator

Discrete Event system Specification (DEVS) were first introduced by Zeigler in 1976 as a rigorous basis for discrete-event modeling. In this chapter we first present an implementation of the classical formalism. An early prototype of a *DEVS Modeling and Simulation Package* is then introduced. The chapter then moves on to discuss a real-time execution framework. The motivation behind real-time DEVS execution is discussed as well as the implications of 0-time semantics.

5.1 Design and Implementation

This version of the *DEVS Modeling and Simulation Package* has been implemented using *Python*, an interpreted, very high-level, object- oriented programming language. The package consists of two files, the first of which (`DEVS.py`) provides a class architecture that allows hierarchical DEVS models to be easily defined. The simulation engine (SE) itself is implemented in the second file (`Simulator.py`). Based on the DEVS simulator described in [73], it uses the same message-passing mechanism. A detailed description of both the *model architecture* and the SE follows.

5.1.1 Model Architecture

The *model architecture* implemented in `DEVS.py` is a canvas from which hierarchical DEVS models can be easily described. It consists of a number of classes arranged to capture the essence of hierarchical DEVS. A model is described in a dedicated file by deriving coupled and/or atomic-DEVS *descriptive* classes from this architecture. These atomic models are then arranged hierarchically through composition. Methods and attributes form the standard interface that allows an SE, such as the one described in the next sub-section, to interact with the instantiated DEVS model. Our main concern with the architecture are twofold: remain as consistent as possible with the original hierarchical DEVS definition, and maintain a flexible approach to DEVS so as to encourage model reusability through parameterization.

The class architecture is represented in Figure 5.1: `BaseDEVS` is the root class which provides basic functionalities common to both atomic and coupled models.

Two classes are inherited from `BaseDEVS` to deal with the specifics of atomic and coupled DEVS formalisms (Figure 5.1). These three classes are all abstract in that they cannot be directly instantiated. Rather, a model is described by deriving *descriptive classes* from either the `AtomicDEVS` or the `CoupledDEVS` class. This provides them with a suitable constructor and overrides the de-

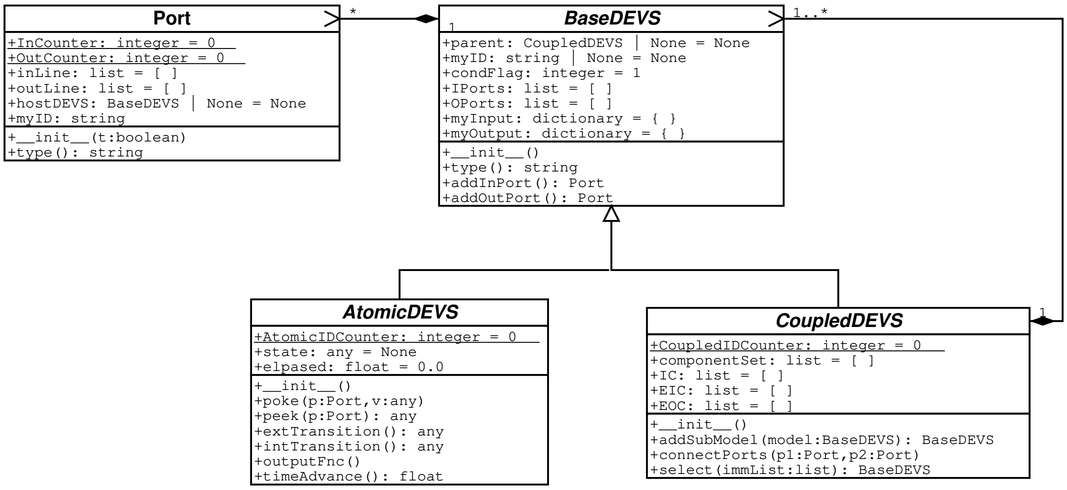

Figure 5.1: Modelisation class architecture

fault interface methods. Note that the constructors at every level of the class hierarchy have an active role. Hence, a descriptive class' constructor should always start by calling the parent class' constructor.

The constructor of the `AtomicDEVS` class merely initializes the `myID` attribute and provides a default initial value for the DEVS' *total state*, through the `state` and `elapsed` attributes. The remaining class definitions consist of default method declarations for the interface functions δ_{ext} (`extTransition`), δ_{int} (`intTransition`), ta (`timeAdvance`) and λ (`outputFnc`). These methods expect no parameter, and it is up to the modeler to be consistent with the corresponding functions' domain when overriding the methods. Except for `outputFnc` (which uses the `poke` method as described below), all the methods shall return a value compatible with the corresponding function range.

Since default values are provided for both attributes and methods, the minimal atomic-DEVS descriptive class is empty:

```
class MinimalAtomicDEVS(AtomicDEVS):
  pass
```

This atomic-DEVS is passive. It remains in its default state forever. A more interesting example is a *generator*, which sends a message (the integer 1 in this case) through its unique output port at a constant time interval:

```
class SimpleGenerator(AtomicDEVS):
  def __init__(self, n = 1):
    AtomicDEVS.__init__(self)
    self.interval = n
    self.message = 1
    self.OUT = self.addOutPort()
  def outputFnc(self):
    self.poke(self.OUT, self.message)
  def timeAdvance(self):
    return self.interval
```

As mentioned above, the `outputFnc` returns no value; instead it relies on the `poke` method to send `message` (second parameter) through the `OUT` output port (first parameter). The companion method, `peek`, returns the message on the input port that is given as a unique parameter, and is used exclusively in the `extTransition` function. Both `poke` and `peek` methods are defined in the `AtomicDEVS` class, and should not be overridden.

The `CoupledDEVS` class only has one method to override, the tie-breaking `select` function. This takes a list of sub-models which are in conflict. The `select` function should return the sub-model instance from this list who's transition is to fire next. All a `CoupledDEVS` sub-models should be included by passing their instance variables to the `addSubModel` function.

Coupling of ports is performed through the `connectPorts` method. Its first parameter is the source port and the second parameter is the destination port. A coupling is rejected and an error message issued if the coupling is invalid.

The source and destination ports are instances of the class `Port`. This class defines channels where events may pass between DEVS models. These channels are defined in the `Port`'s `inLine` and `outLine` attributes.

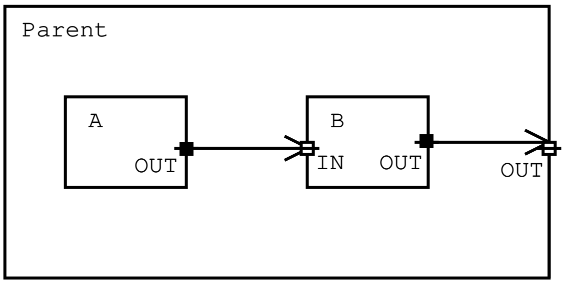

Figure 5.2: Simple Coupled-DEVS

Consider the example of the situation illustrated in Figure 5.2. There exists a descriptive class `SomeDEVS` for a DEVS (either atomic or coupled) with an input and output port locally known as `IN` and `OUT`. We want to connect the input port to the output port of the `SimpleGenerator` atomic-DEVS described above: both DEVS must of course be children of the same coupled-DEVS for the coupling to be performed. The coupled-DEVS descriptive class is defined below.

```
class Parent(CoupledDEVS):}
  def __init__(self, x = 1):
    CoupledDEVS.__init__(self)
    self.OUT = self.addOutPort()
    self.A = self.addSubModel(SimpleGenerator(x))
    self.B = self.addSubModel(SomeDEVS())
    self.connectPorts(self.A.OUT, self.B.IN)
    self.connectPorts(self.B.OUT, self.OUT)
```

Note that the parameter to the constructor is used to parameterize the `SimpleGenerator` atomic-DEVS. As for atomic-DEVS, the constructor first calls the parent class' constructor. Note also that the coupled-DEVS itself has an output port. The first coupling is an *internal coupling*, while the second is an *external output coupling*. This is a complete definition for a coupled-DEVS descriptive class. The default `select` function is used.

Once all the descriptive classes in the hierarchical DEVS model have been defined, the whole model can be build by instantiating the root DEVS. This is possible since the hierarchical representation of the model is built by *composition* rather than *aggregation*.

As a final warning, note that recursive definitions are illegal, since they are incompatible with a tree structure. As a trivial example, a coupled-DEVS' descriptive class `SomeCoupledDEVS` cannot call in its constructor the `addSubModel` method with an instance of `SomeCoupledDEVS`. This is mentionned since such recursive constructs will not be detected.

5.1.2 Simulation Engine

The *Simulation Engine* (SE) is the tool that simulates the behavior of a system given its model as specified in the previous section. Since the SE is independent from the model architecture presented above, a clean interface between the model and the simulator must be defined. This allows alternative SEs (eg. parallel implementations) to be written. The particular SE we describe here is loosely based on the DEVS simulator described in [73].

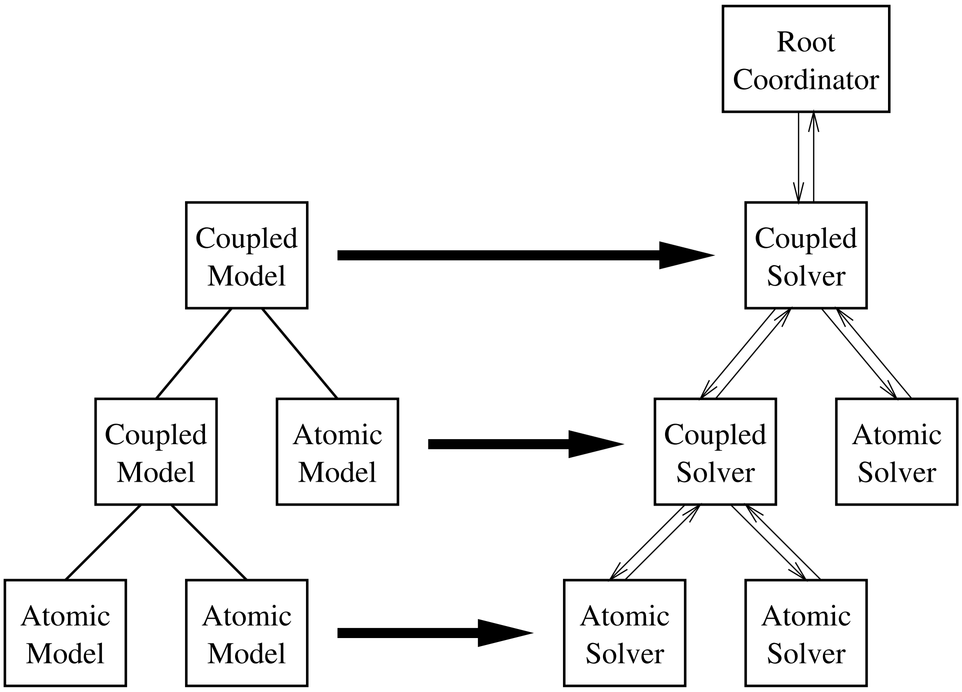

Figure 5.3: Mapping a hierarchical model onto a hierarchical simulator

The main challenge in writing a SE for a hierarchical DEVS model consists in determining, at run-time, the sequence (atomic or coupled) of DEVS activation. As it turns out, the hierarchical representation of the model (Figure 5.3), although it gives no information about couplings, helps tackle the problem by means of a message-passing mechanism*: in Zeigler's words [73],

> A hierarchical simulator for hierarchical DEVS coupled models consists of *devs-simulators* and *devs-coordinators* and uses four types of messages. An initialization method (i, t) is sent at the initialization time from the parent simulator object to all its subordinates, to synchronize their clocks. The scheduling of events is done by the

*The *transfer* of messages (data) from port to port shall not be confused with the *passing* of messages (control) from solver to solver.

internal state transition message $(*, t)$ and are sent from the coordinator to its imminent child. An output message (y, t) is sent from the subordinates to their parents to notify them of output events. The input message (x, t) is sent from the coordinator to its subordinates to cause external events.

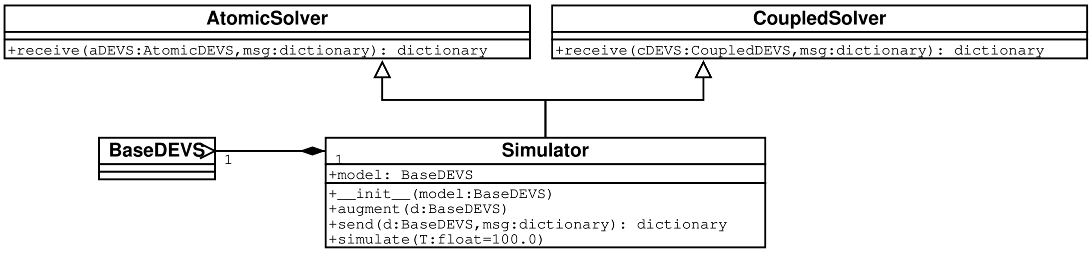

Figure 5.4: Simulation class architecture

Note that the t in the messages is the time stamp of that message. The other part of the message is either a command flag (i or $*$), or an input or output dictionary (x and y). Whereas Zeigler calls the coupled-DEVS and atomic-DEVS solvers *coordinators* and *simulators* respectively, *simulator* is a term that should be reserved for the whole SE. Consequently the terms *atomic-solvers* and *coupled-solvers* are preferred.

DEVS objects being passive by themselves, the role of the corresponding solvers is to operate on them when they are activated (ie. call the DEVS' methods and modify the DEVS' attributes). This being said, we have to find a mechanism to either *augment* each DEVS in the hierarchical model into a solver, or to *associate* the DEVS with an appropriate solvers.

```
S = Simulator(Parent(1))
```

Upon instantiation, the simulator class calls its `augment` method which traverses the hierarchical model to effectively augment the DEVS objects with required attributes. These consist of `timeLast` and `timeNext`, which respectively hold the simulation time when the last transition occurred and the scheduled time for the next internal transition. In the case of a coupled-DEVS, an `eventList` attribute is also added which is the list of pairs (tn_d, d), where d is a reference to a sub-model of the coupled-DEVS and tn_d is set to the `timeNext` attribute of that model (hence for a coupled DEVS `timeNext` is set to the *minimum* tn_d in its `eventList`).

The simulation is started by calling the `simulate` method as follows.

```
S.simulate(30)
```

The parameter is the length of the simulation (simulation, not wall-clock, time). The `simulate` code corresponds to that of Zeigler's *root-coordinator*. It starts by sending an initialization message (i, t) to the root-DEVS, which will cascade through the model. The main loop then repeatedly sends internal state transition messages $(*, t)$ to the root DEVS until the simulation clock reaches the length of the simulation. Messages are passed from DEVS to DEVS by means of the simulators' `send` method, which dispatches the messages to either the `AtomicSolver.receive` or the `CoupledSolver.receive` method as required. The progress of a simulation is dumped to the standard output.

A final note is in order to stress a difference between the message-passing pattern described here and that described in Zeigler's view. Since (y, t) messages are sent back to a parent coupled-DEVS

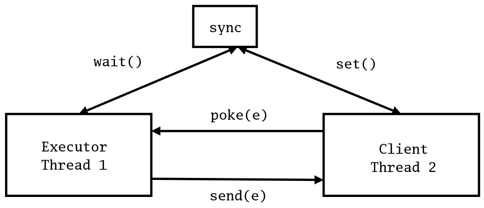

Figure 5.5: The `pythonDEVS-RT` threading model.

only in response to sending a $(*, t)$ message, the former message is "sent back" as a returned value rather than by means of the `send` method. Hence the receive methods in both solver classes treat only three kinds of messages, namely (i, t), $(*, t)$ and (x, t). This design choice, which does not impact the behavior of the simulated model, is motivated by the fact that extra-coding would otherwise have been required to distinguish between (x, t) and (y, t) messages. As a matter of fact, these messages are implemented as pairs whose first member is either an input or output dictionnary, which are essentially indistinguishable.

5.2 A Real-Time `pythonDEVS` Execution Environment

5.2.1 Motivation

Instead of simulating DEVS models where the implicit time variable is simulation time, we wish to execute a model where time units are equivalent to wall-clock time. The reason for this is to be able to execute programs with the DEVS real-time engine driving the dynamics of the application. Graphical programs are an example of this desired framework. The graphical application has entities such as buttons which generate events. These events are sent to the real-time DEVS engine which may send a return message. The graphical application can then change its state based on the received message.

In essence, if the current time is t and the time of the next internal transition is $t + x$, then the simulation environment should block for x time units.

5.2.2 Design Structure

Firstly the `pythonDEVS-RT Executor` class as well as the threading model are explained. Following, is a discussion of `pythonDEVS-RT` Executor's wait-dispatch loop. Finally, the interface between the Executor and client applications is outlined.

Threading Model

When an application uses a DEVS engine as its underlying dynamics, both the client and the engine are started in separate threads. This is illustrated in figure 5.5. The `sync` block is a global `Event` object taken from the `threading` library. This is the mechanism which synchronizes activities between the clients and the Executor. The Executor is in an unending wait-dispatch loop which utilizes calls to the `wait(x)` function. The method, `wait(x)`, will return if a call to `set()` is made from another thread or if x time has passed. Thus, when a client wishes to send an event to the DEVS engine, it will first `poke()` the event and then `set()` the `sync` object. The engine will then wake up, and process the event which was poked.

The Wait-Dispatch Loop

If the clock reads x, then the Executor must explicitly wait for $t_N - x$ time units, where t_N is the time of the next internal transition. Once the wait stage is passed, the Executor must decide what type of message to send to its sub-components. It searches all its input ports and checks if any of them correspond to components which are sending events. If there exists such an event, it is considered an incoming external event and is passed to the DEVS model's corresponding input port. If there are no external events, a timeout must have occurred and an internal event signal is sent to the DEVS model.

The code below is the Executor's wait-dispatch loop, taken from `RT_DEVS.py`.

```
# ... initialization ...
clockRT = 0.0
while 1:
  ta = self.model.myTimeAdvance # ta IS THE TIME OF THE NEXT INT TRANS
  if ta == INFINITY:
    initial = time()
    self.sync.wait() # WAIT FOREVER
    time_passed = time() - initial
  elif ta-clockRT > 0:
    initial = time()
    self.sync.wait(ta-clockRT) # WAIT UNTIL NEXT INT TRANS
    time_passed = time() - initial
  else: time_passed = 0.0
  if ta == INFINITY or ta-clockRT > 0.0: # SHOULD CLOCK MOVE FORWARD?
    if self.sync.isSet(): clockRT += time_passed # INTERRUPT
    else: clockRT += (ta-clockRT) # TIMEOUT
  self.sync.clear()
  if self.done:
    return
  # ... if external event, send ext trans signal, else send int trans signal ...
```

0-time Semantics

There are different ways to interpret 0-time semantics in a real-time execution environment. Consider the following atomic DEVS model, A_1. Any information not included is assumed to be the degenerate case.

$$
\begin{aligned}
A_1 &= \{S_1, ta_1, \delta_{e,1}, \delta_{i,1}, \lambda_1, X_1, Y_1\} \\
S_1 &= \{\varepsilon_{1,1}, \varepsilon_{1,2}\} \\
ta_1(\varepsilon) &= 0 \qquad \forall \varepsilon \in S_1 \\
\delta_{i,1}(\varepsilon) &= \begin{cases} \varepsilon_{1,1} & \text{if } \varepsilon = \varepsilon_{1,2} \\ \varepsilon_{1,2} & \text{if } \varepsilon = \varepsilon_{1,1} \end{cases}
\end{aligned}
$$

And also the atomic model A_2. Again, information not included indicates the degenerate case.

$$
\begin{aligned}
A_2 &= \{S_2, ta_2, \delta_{e,2}, \delta_{i,2}, \lambda_2, X_2, Y_2\} \\
S_2 &= \{\varepsilon_{2,1}, \varepsilon_{2,2}\} \\
ta_2(\varepsilon_{2,1}) &= 0 \\
\delta_{i,2}(\varepsilon_{2,1}) &= \varepsilon_{2,2}
\end{aligned}
$$

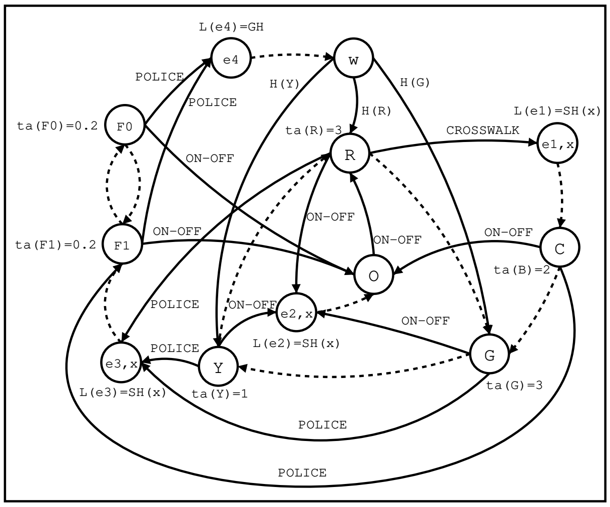

Figure 5.6: The DEVS crosswalk model.

If these models are included as child models within some coupled model C, it becomes unclear as to when the internal transition within A_2 will fire. There are two interpretations. In the first interpretation, A_1 continually schedules an internal transition to occur within 0 time units. Thus, the clock never advances and A_2 will not execute its internal transition.

Another interpretation is that a 0-time transition is an abstraction of reality. Executing the transition does take x time units and the clock should move forward this amount. Thus, the number of times $\delta_{i,1}$ will fire before $\delta_{i,2}$ depends on the implementation of the Executor and the machine running the simulation.

The implications of either approach depends on the desired use. Such a decision is left up to the reader. However, it should be noted that the first of the two methods described above is the one implemented in the `RT_DEVS` module. To implement the second approach, a slight modification to wait-dispatch loop is required.

The initial time should be stored in `initialRT` before the loop is entered. Next, the way `clockRT` is updated after `wait` returns must change. Instead of incrementing the clock by the amount of time that has passed while waiting, simply set the `clockRT` variable as follows.

$$\texttt{clockRT} = \texttt{time}() - \texttt{initialRT}$$

Client Interface

Client applications may wish to be controlled by a DEVS model which is executed by the real-time Executor. Naturally, any client program should be able to send events to a DEVS. However, the DEVS cannot send events back since the client programs may not be DEVS models. Instead, the `send` method emulates the ability to send events from a DEVS model to a particular client application.

This client functionality is derived from the class `Generator` found in the `RT_DEVS` module. The name `Generator` comes from the fact that class instances may *generate* events at any point in time. The client should override the `recv` method to receive the events sent using `send`. The client should also add output ports to poke events, but no input ports since events are not received via ports. `RT_DEVS` also includes a global `connectPorts` function which is used to connect client output ports to the DEVS model's input ports.

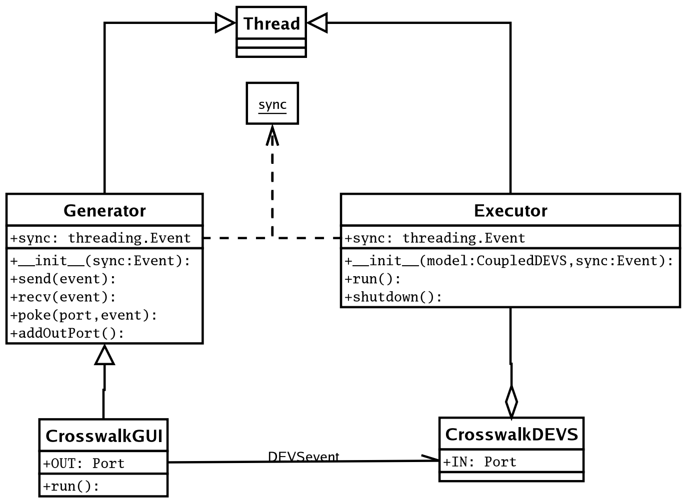

Figure 5.7: The crosswalk UML class diagram.

5.2.3 Crosswalk Example

This example can be downloaded from:
`http://msdl.cs.mcgill.ca/people/sborla/pythonDEVS-RT/`.

The model in figure 5.6 is a model of a crosswalk and traffic light. The light alternates between RED, YELLOW and GREEN. If the light is RED and the crosswalk button is pressed the light will turn GREEN faster. A 'Police Interrupt' will cause the RED light to flash repeatedly only until the 'Police Interrupt' is pressed again. After the 'Police Interrupt' is pressed a second time, the light will turn the same color it was before the first time this button was pressed. Moreover, the entire system can be turned on and off. Once turned off, the system forgets all information, and once turned on the system will always start in the RED state.

The program is run by typing "`python CrossWalkAPP.py`" at the shell prompt. The class diagram for the application can be seen in figure 5.7. The program starts by creating a new application object which instantiates and initializes the `Tkinter` graphical objects as well as the underlying DEVS engine.

6

Transforming Statecharts to DEVS

6.1 Motivation

The Statechart formalism makes use of very high level constructs not seen in many formalisms developed to this day. The high level constructs are part of its appeal. The more abstracted a formalism becomes, the closer it is to the natural human thought process. DEVS, on the other hand, has few high level syntactic elements. Thus, it is natural to transform the high-level to the low-level and try to learn about the semantics of the high-level formalism which is hidden in its engine. In effect, part of the semantics of Statecharts is removed from the engine and is modeled explicitly using DEVS (see Figure 6.1). The information required to describe the Statechart model with DEVS increases as a result of this transformation because the rich syntactic elements in Statecharts are now being described explicitly.

This transformation also makes it very easy to plug and play Statecharts with other formalisms. Since DEVS is so versatile, a transformation from other formalisms can be defined (e.g., Ordinary Differential Equations to DEVS [11]) and can be glued together. In the case of continuous formalisms, a hybrid environment is immediately possible, since Statecharts are discrete.

Another reason for this mapping is the extensions to the Statechart formalism. In UML 2.0, statecharts have ports. These ports increase the modularity and compositionality of the formalism. DEVS already uses ports for communication between components.

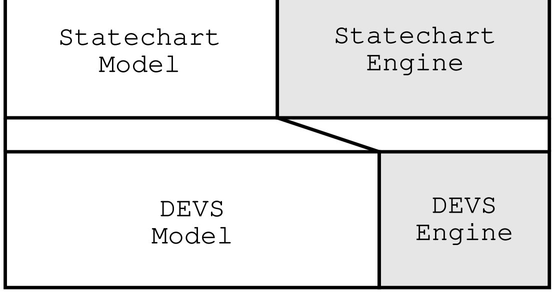

Figure 6.1: Transforming Statecharts to DEVS.

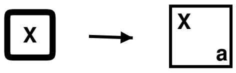

Figure 6.2: Converting a Basic Statechart state.

6.2 Introduction

The transformation performs the following high-level actions. *Hooks* are a sequence of DEVS transitions which start at some state ε, such that $ta(\varepsilon) = 0$. A *handle* is a sequence of DEVS transitions which starts with some reaction to an external event.

1. Transform states
2. Build enter & exit handles
3. Build hooks
   (a) Default hook
   (b) History hook
   (c) History star hook
4. Build handles
   (a) Current state query (CSQ) handle
   (b) Arbitrary event, X, handle
5. Build transitions

Appendix A gives a summary of the notation used in this chapter. In each section, a part of a DEVS model will be described. The input and output sets, X and Y, the state set, S, and the ta function will never be given. Instead, states will be introduced as the description of the transformation continues. If the state is written as ε, then $ta(\varepsilon) = 0$ and $S \leftarrow S \cup \{\varepsilon\}$. If the state is written as ω, then $ta(\omega) = \infty$ and $S \leftarrow S \cup \{\omega\}$. When building handles and hooks, an event, x, may be introduced. If x is broadcast, then $Y \leftarrow Y \cup \{x\}$ where Y is the output set for the source state DEVS of the transition. If x is a trigger for a transition, then $X \leftarrow X \cup \{x\}$ where X is the input set of the source state DEVS of the transition.

Most importantly, the δ_e function will almost always be written as $\delta_e(s, f)$. This is a short hand notation for $\delta_e((s,e), f)$ where e is the elapsed time parameter. This indicates that most external transitions do not depend on the elapsed time.

6.3 Transforming States

The transformation starts by building the skeleton of the DEVS model. The skeleton consists of the model hierarchy. The state, input and output sets along with the ta, δ_e, λ and δ_i functions are specified later. The goal of this section is to specify each DEVS model with its corresponding sub-models.

First, the transformation indicates that a basic state is converted to an atomic DEVS (Figure 6.2). These atomic models will be referred to as TERMINALS.

Second, the transformation specifies that a composite state or an orthogonal component is converted to a coupled DEVS with one sub-model (atomic) called a *relay* (Figure 6.3).

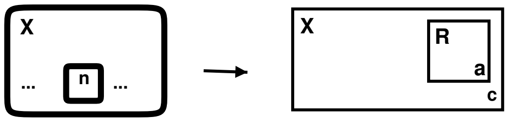

Figure 6.3: Converting a Composite Statechart state.

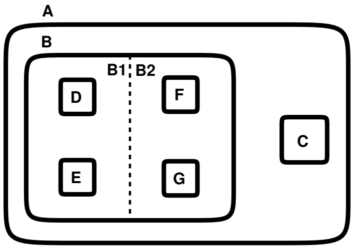

Figure 6.4: A Statechart Model.

The purpose of the relay is to act as a controller for the converted composite state. This is required since a coupled model has no state in its own right, nor does it specify any dynamics. A coupled model only specifies the static coupling among its sub-models. The dynamics for the composite state are included in the relay. These entities (coupled model with atomic relay) will be referred to as NON-TERMINAL.

This already makes it very easy to see the structure of the resultant DEVS model. Figure 6.4 is a diagram of a simple Statechart model and 6.5 is the resultant DEVS model according to the transformation described to this point.

At first glance, the hierarchical structure of a DEVS model looks very similar to its Statechart. However, the meaning of the two is *very* different. In a Statechart with no orthogonal regions, at most one sub-state can be current at a time. A DEVS model with sub-models means each sub-model is concurrent. Something must be done to bridge the large semantic gap between these two formalisms. This is one of the goals of this transformation.

Now that the shell of the DEVS model is built, all the sub-models may be coupled. Coupling involves specifying the *input-to-output translation* ($Z_{i,j}$) function. This specification indicates channels where events may pass. It is crucial that sub-models can send events to their parent and that parent models can send events to their children. Also, it would be nice if a parent could choose to broadcast to all of its children or send an event to only one particular child model. These are

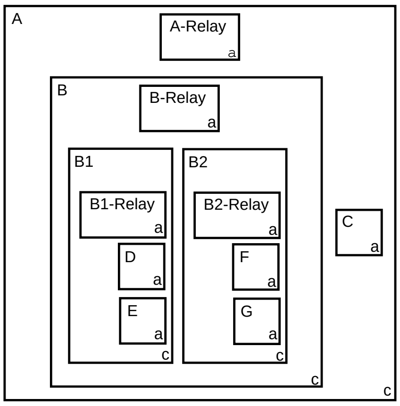

Figure 6.5: The DEVS shell.

constraints which must be taken into consideration when designing the Z function.

6.4 Ports and Coupling

Once the shell of the DEVS model is constructed, it is possible to specify all ports as well as the entire coupling. The specification of the output-to-input translation function, $Z_{i,j}$, is a very important part of transforming Statechart models to DEVS models.

6.4.1 Port Names

Every single atomic DEVS in the resulting model must have both a $\vdash$ input port and a $\perp$ output port. The $\vdash$ port is the sole input port. This is from where all incoming events come. The $\perp$ port is a port which must connect the current DEVS with its parent model. Think of $\vdash$ as an arrow pointing inward (input port) and the $\perp$ port as an arrow pointing upwards to a parent model.

Relays have additional ports, including the $\top^\star$ output port and $\top_i$, $\forall i \in D$ where D belongs to the containing coupled DEVS. Think of $\top$ as an arrow pointing downwards to a child model. The $\top^\star$ port connects the current DEVS with all of its children (i.e., they connect the current relay with multiple children). The $\top^\star$ port is used by a composite state to broadcast to each of its orthogonal regions. The ports for each child (i.e., they connect only the current relay with one child) are named after the children themselves. For example, consider a Statechart, P, with child state, C. $RELAY_P$ will have a port named C. This way, the relay can easily send events directly to any of its children.

Note that using this interface, all terminals only have two ports, $\vdash$ (sole input port) and $\perp$ (parent output port). This is because a terminal has no children, and thus needs no output ports for communicating with sub-models.

6.4.2 Coupling

Each Coupled DEVS model must specify the output-to-input translation function, $Z_{i,j}$, such that events put on output ports in its relay must be sent to the proper adjacent models. Since the domain and range of the Z function are the X and Y sets and these sets have not yet been specified, the coupling of ports will be explained in plain English. For example, output port $\top$ connects to input port $\vdash$.

In general, Z for a coupled model A with its relay $r \in D$ is as follows. $RELAY_A$'s parent port connects to A's parent port (recall that $RELAY_A$ and A can be thought of as interchangeable). A's $\vdash$ port connects to $RELAY_A$'s $\vdash$ port. The port named $i \in D$, $D \in A$ connects to M_i, $i \in D$, $i \neq r$. $RELAY_A$'s $\top^\star$ port connects to sub-model 1's $\vdash$ port, sub-model 2's $\vdash$ port, ..., sub-model n's $\vdash$ port.

Figure 6.6 is a visual representation of this description.

6.5 The Event Framework

The resulting DEVS model now has enough structure for events to traverse its hierarchy. Each event generated and received by a DEVS model must adhere to a protocol so that each sub-model can handle the events in a uniform way.

Each event must have an *address* parameter. The address indicates to where the event should be routed. A terminal or relay DEVS can examine the address and take action accordingly.

Events are structured sets (see Appendix A). They have four named coordinates or fields. These are the name field, Ψ, the address field Δ, the broadcast field, β and the history field, H.

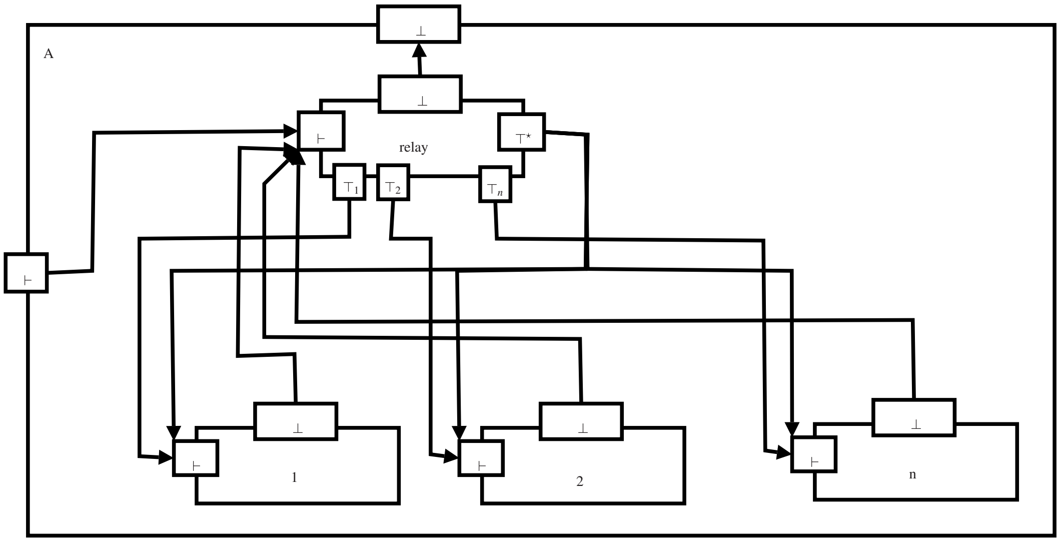

Figure 6.6: Coupling between a parent and its children.

6.5.1 The Name Field, Ψ

The name field is simply a string of characters used to identify the type of the event. If a Statechart has a transition with trigger, e, then the equivalent DEVS event, x will have $x.\Psi =$ "e".

6.5.2 The Address Field, Δ

The address field is an ordered set which indicates where an event should be routed. If the set is empty, the event has reached it destination. Otherwise, each DEVS model that receives the event should look at the first element in the set using the $\uparrow$ operator. This element will indicate upon which output port the event should be sent. This is why each relay and terminal state must have the same way of naming ports.

6.5.3 The Broadcast Field, β

This is a boolean value which indicates if the event should be broadcast. If so, the event will continue to its destination given by Δ, then will be pushed out port $\top^{\star}$.

6.5.4 The History Field, H

The history field takes on different values in different contexts. When H is a field of an α event, the history field is also a boolean value. It indicates if the α event should enter a history state instead of the usual default state. If $\beta = 1 \wedge H = 1$,the α event will emulate deep history semantics.

In the context of a $\chi^{\star}$ event or a state, the H field takes on the value of an actual state. This actual state is the history state.

6.6 Maintaining the Current State

Note that while the hierarchy of a Statechart model and its DEVS skeleton seems very similar, the meaning of these two constructs is quite different. In Statecharts, only one state is current at a time (unless orthogonal). In DEVS, a state from *all* atomic models is current. Every single atomic model is executing in parallel. Thus, every atomic model, TERMINAL or NON-TERMINAL must have two important states which indicate whether that particular Statechart state is *current* or *not-current*. These states are specified as follows.

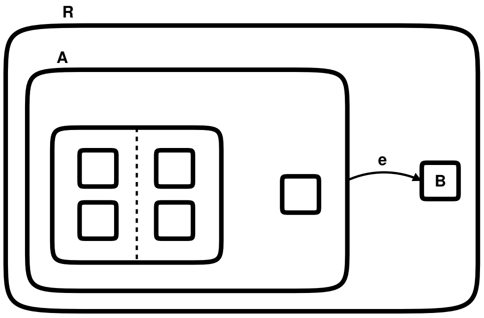

Figure 6.7: State A has Deep Children.

- $\Sigma \in S$, $ta(\Sigma) = \infty$: Denotes a current state (in-scope)
- $\Gamma \in S$, $ta(\Gamma) = \infty$: Denotes a non-current state (out-of-scope)

A coupled model does not have an explicit state set. Thus, the Σ and Γ states are kept in the coupled model's relay. In a sense, the relay is a controller for a particular coupled model.

6.6.1 Synchronizing the Current State

Transitions are the constructs in Statecharts which change the current state. In fact, when a transition fires, the source state is exited and the destination state is entered. As such, a mechanism is required to exit, as well as enter, converted DEVS states.

When a state is exited, its exit action must be executed. However, if that state has deep children, these states must first be exited and their exit actions executed. Conversely, when a state is entered, its enter action must be executed. However, if that state has children, then the default state must also be entered and its enter action executed. This entering continues down the hierarchy until a terminal is reached.

Exiting a state is signaled by the exit-all event χ. Acknowledgement of the exit is signaled by the exit-all-acknowledgement event $\chi^{\star}$.

Conversely, entering a state is signaled by the enter event α. An α acknowledgement event is not required since enter actions are executed as α events are received. That is, parent enter actions are executed before child enter actions.

It is the firing of a transition which generates these special events. The reaction to each of these special events will be explained first to simplify the explanation of the Statechart transitions later. In the following section, the transition is examined which will help fully understand how these two mechanisms, χ and α, work together.

6.6.2 The χ and $\chi^{\star}$ Events

Each terminal model as well the relays must handle the χ and $\chi^{\star}$ events. The objective is to move from an in-scope state (Σ) to an out-of-scope state (Γ). However, a particular state, such as state A in Figure 6.7, may have deep children. Before moving out-of-scope, all of the state's deep children must execute their exit actions and move to the Γ state.

This behavior is modelled as follows. When a *terminal* receives a χ event, it should only execute its exit action then send a $\chi^{\star}$ to its parent.

$$\begin{aligned}\delta_e(\Sigma,\chi) &= \varepsilon\\ \delta_i(\varepsilon) &= \Gamma\end{aligned}$$

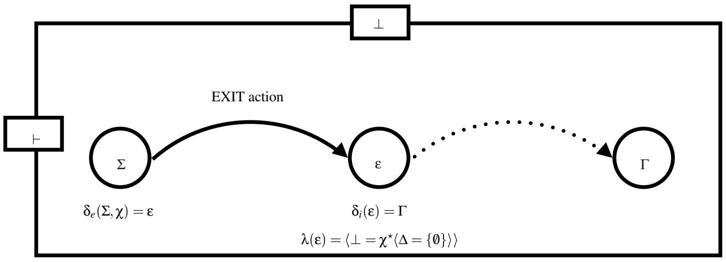

Figure 6.8: Terminal reaction to χ.

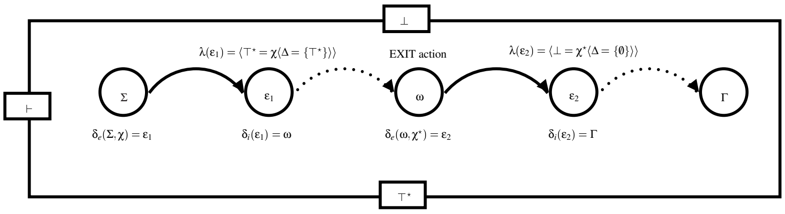

Figure 6.9: Non-terminal reaction to χ.

$$\lambda(\varepsilon) \quad = \quad \langle \perp = \chi^{\star}\langle \Delta = \{\emptyset\}\rangle\rangle$$

When a *relay* receives a χ event from its parent (they can only be received from the parent) it must first send another χ to all its children then wait for acknowledgement ($\chi^{\star}$). When the acknowledgement is received, it may execute the state's exit action and send an acknowledgment to its parent.

Notice that only the current state(s) will respond to the event, since δ_e is a function of Σ. If there are no orthogonal regions, then only $k=1$ sub-states will be current. If there are n orthogonal regions, then $k=n$ sub-states will be current. Thus, the relay must collect k, $\chi^{\star}$ acknowledgements before it can send its own $\chi^{\star}$ acknowledgement. Notice the recursive nature of exiting a state. χ events are continually pushed deeper until they reach a terminal at which point $\chi^{\star}$ events are sent back to the parent nodes.

$$\begin{array}{rcl}
\delta_e(\Sigma,\chi) & = & \varepsilon_{\top} \\
\lambda(\varepsilon_{\top}) & = & \langle \top^{\star} = \chi\langle\Delta = \{\top^{\star}\}\rangle\rangle \\
\delta_i(\varepsilon_{\top}) & = & \omega_1 \\
\delta_e(\omega_1,\chi^{\star}) & = & \omega_2 \\
\delta_e(\omega_2,\chi^{\star}) & = & \omega_3 \\
 & \ldots & \\
\delta_e(\omega_{k-1},\chi^{\star}) & = & \varepsilon_{\perp} \\
\lambda(\varepsilon_{\perp}) & = & \langle \perp = \chi^{\star}\langle\Delta = \{\emptyset\}\rangle\rangle \\
\delta_i(\varepsilon_{\perp}) & = & \Gamma
\end{array}$$

A visual representation of the reaction to the χ event can be seen in Figures 6.8 and 6.9. The reaction at a terminal state can be seen in Figure 6.8 and at a non-terminal state can be seen in figure 6.9. The Figures represent states which do not have orthogonal components. That is, only one $\chi^{\star}$ event is required to acknowledge that all sub-models are safely out-of-scope.

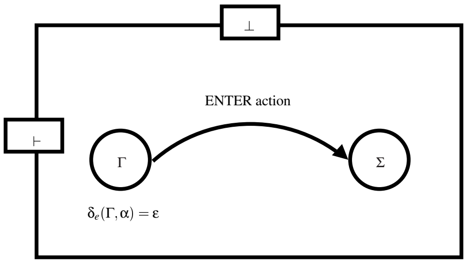

Figure 6.10: Terminal reaction to α.

6.6.3 The α Event

The α event is used to signal that a state is to move in scope (Σ). When a relay receives an α event, it must signal its default sub-state to also move in scope. These semantics are constructed at compile time in the *default hook sequence*. The *default hook sequence* starts at a state named ε_{def} and simply sends an enter event to a particular state's default sub-state. For a Statechart, *A*, with default state *D*, the following sequence would be added to $RELAY_A$.

$$\begin{aligned}\delta_i(\varepsilon_{def}) &= \Sigma \\ ta(\varepsilon_{def}) &= 0 \\ \lambda(\varepsilon_{def}) &= \langle \top_D = \alpha\langle \Delta = \{\emptyset\}\rangle\rangle\end{aligned}$$

If *A* has orthogonal components, then α events must be sent to each component. This, too, is easily built into the *default hook sequence*. The relay will simply broadcast the α event to each orthogonal component. This is done by outputting α on the $\top^\star$ port. The relays for each orthogonal component will then send the α to the default state.

In general, once the *default hook sequence* is in place, the δ_e for relays is set as follows.

$$\delta_e(\Gamma,\alpha) = \varepsilon_{def} \text{ if } |\alpha.\Delta| = 0$$

The *default hook sequence* for a composite state with orthogonal components is therefore different from a composite state without orthogonal components. The sequence for a composite state with orthogonal components must broadcast an α event then move in-scope.

$$\begin{aligned}\delta_i(\varepsilon_{def}) &= \Sigma \\ ta(\varepsilon_{def}) &= 0 \\ \lambda(\varepsilon_{def}) &= \langle \top^\star = \alpha\langle \Delta = \{\emptyset\}\rangle\rangle\end{aligned}$$

When a terminal receives an α event, there is no need to relay anymore α events since terminals have no children. Thus the terminal simply moves from the Γ to the Σ state and executes its enter action.

$$\delta_e(\Gamma,\alpha) = \Sigma$$

A visual representation of the reaction to the α event can be seen Figures 6.10 and 6.11. The reaction is a terminal can be seen in Figure 6.10 and in a non-terminal can be seen in Figure 6.11. The non-terminal sequence shown is for a composite state without orthogonal components.

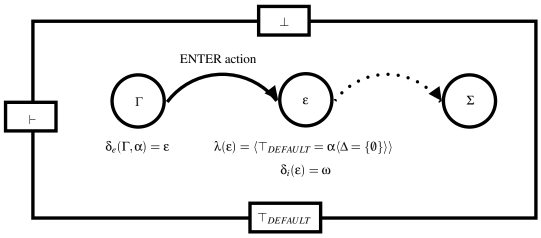

Figure 6.11: Non-terminal reaction to α.

6.6.4 The Relay Sequence

Consider relay R. R receives event E such that $|E.\Delta > 0|$, but $length(\varphi) > 0$. Thus, this event's destination is some other state in the hierarchy. It is now R's job to route E accordingly. This is the main function of the relay sequence. The relay sequence starts at a state named ε_Δ. It must pop an event off the address stack, and push the event out the port with the same name as the popped value. Note that the *top* function returns the top value of a stack.

$$\begin{aligned}
\delta_e(\Sigma, E) &= \varepsilon_\Delta \text{ if } |E.\Delta > 0| \\
\delta_i(\varepsilon_\Delta) &= \Gamma \text{ if } E.\Psi = \text{"}\alpha\text{"} \wedge \uparrow E.\Delta = \perp \\
ta(\varepsilon_\Delta) &= 0 \\
\lambda(\varepsilon_\Delta) &= E
\end{aligned}$$

6.6.5 The Broadcast Sequence

The broadcast sequence, which starts at state ε_β, is similar to the relay sequence. The address parameter of the incoming event must be empty, and the broadcast field of the event is set to TRUE. In this case, a new address stack is assigned to the address field. This stack consists of only one value, $\top^\star$.

$$\begin{aligned}
\delta_e(\Sigma, E) &= \varepsilon_\beta\langle \mu = E\rangle \text{ if } |E.\Delta| = 0 \wedge E.\beta = 1 \\
\delta_i(\varepsilon_\beta) &= \Sigma \\
ta(\varepsilon_\beta) &= 0 \\
\lambda(\varepsilon_\beta) &= E\langle \uparrow (\varepsilon_\beta.\mu.\Delta = \varepsilon_\beta.\mu\langle \Delta = \Uparrow (\varepsilon_\beta.\mu.\Delta)\rangle\rangle
\end{aligned}$$

Broadcasting Optimization - Avoid $\top^\star$

When events are broadcast through the $\top^\star$ port, they are received by all the sub-states (including the states that are in a Γ state). This is unnecessary since out-of-scope states never 'hear' broadcasted events. Thus, if the parent state were to track which of its children, X, is in a Σ state, then the output event can be pushed out port X. The out-of-scope states will remain dormant, and some extra computation is avoided.

Tracking which sub-state is in-scope is very easy and can be done by looking at the top of the address stack for an α event. If the top of the stack is X, where $X \neq \perp$, then the parent can add a field to its current state indicating that X is its current sub-state.

In the case of orthogonal components, this cannot be done. The output events must still be pushed out the $\top^\star$ event. The orthogonal relays will remember their current sub-state.

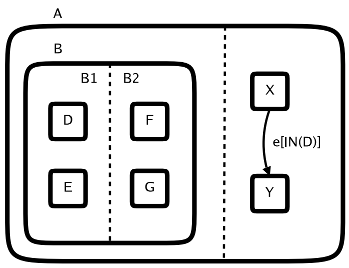

Figure 6.12: State reference.

6.7 Transforming Transitions

The DEVS formalism is much more modular and compositional than the Statechart formalism. Unlike in Statecharts, one cannot draw inter-level transitions. This is one factor which breaks the compositionality of the Statechart formalism. These are semantics which must be considered when modeling Statechart transitions using DEVS.

One thing Statecharts and DEVS have in common is that a firing transition is a function of the current state and the trigger event. Thus, it is natural to build the start of the transition at the source of the transition. Even though the transition may span several model borders, it is still necessary to know when the transition fires. For example, if there is a transition T going from state X to Y with trigger e, a DEVS transition should be built in the atomic component named X from state Σ to Γ with trigger e. That is, X's δ_e function should look something like the following.

$$\delta_e(\Sigma, E) \quad = \quad \Gamma \text{ if } E.\Psi =' e'$$

Of course, there is much more to a transition than what has been explained so far. There must be a way of ensuring that the destination, Y, moves from state Γ to Σ. Moreover, the transition action must be executed and broadcasted outputs must be generated. Also, the source state may have deep children. These children must move the Γ state as well.

These are exactly the mechanisms, χ and α already described. Once χ and α are already in place, changing the current state for a transition, t, becomes very easy.

- Execute transition action
- Do χ, $\chi^\star$ sequence for the source of t
- Do α sequence for the destination of t

For example, consider a transition which reacts to event, E, has source A and destination B. The path from A to B is $X_1, X_2, \ldots, X_n$.

$$\begin{aligned}
\delta_e(\Sigma, E) &= \varepsilon_1 \\
\delta_i(\varepsilon_1) &= \omega_1 \\
\lambda(\varepsilon_1) &= \langle \top^\star = \chi\langle \Delta = \{\emptyset\}\rangle\rangle \\
\delta_e(\omega_1, \chi^\star) &= \varepsilon_2 \\
\lambda(\varepsilon_2) &= \langle X_1 = \alpha\langle \Delta = (X_2, \ldots, X_n)\rangle\rangle \\
\delta_i(\varepsilon_2) &= \Gamma
\end{aligned}$$

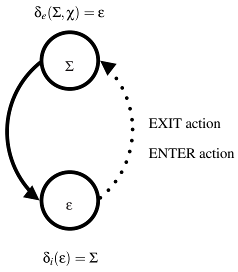

Figure 6.13: Terminal reflexive or inward transition optimization.

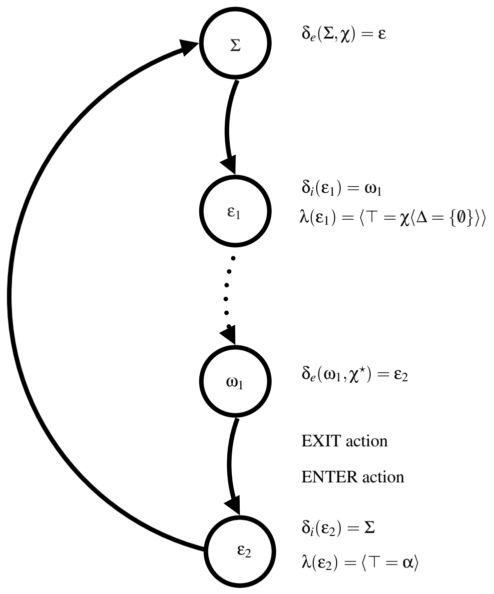

Figure 6.14: Non-terminal reflexive or inward transition optimization.

6.7.1 Reflexive or Inward Pointing Transitions

Reflexive and inward pointing transition (inward means the destination of the transition is a sub-state of the source of the transition) are a special case which can provide a slight optimization. If this type of transition fires, an α event is sent to itself. This event must pass though the parent relay. Instead of passing the α to the parent, the relay can simply remain in the Σ state and can execute its exit and enter actions, respectively.

A simple transformation can provide such an optimization. For terminals, simply execute the exit action, then enter action and move to the Σ state. See Figure 6.13.

For non-terminals, first perform the χ sequence. Upon receiving the $\chi^{\star}$, simply execute the exit action, then the enter action and move to the Σ state. Finish with the α sequence. See Figure 6.14.

6.8 Boolean Guards

Transitions also have boolean guards which can cause a transition to not fire if the value of the guard is false. Thus, before a transition action is executed, the boolean guard must be calculated. If the value is true, then the transition will fire, and the source state is exited and the destination state is entered. Otherwise, the transition does not fire, and the system must move back to the Σ state.

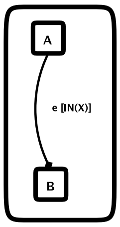

Figure 6.15: A state reference boolean guard.

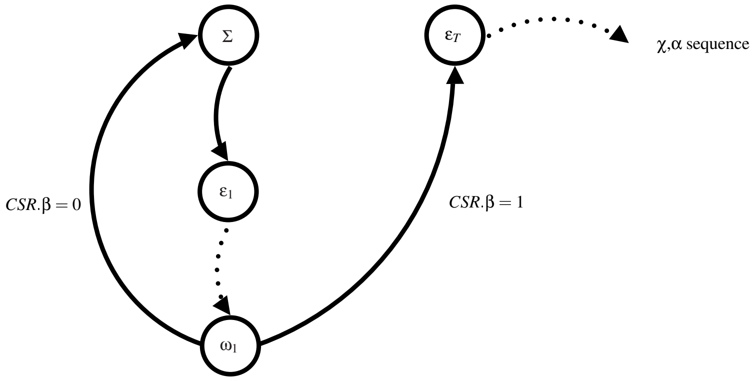

Figure 6.16: Transformed transition.

To make matters more complicated, the boolean guard may be a state reference. In this case, a *Current State Query-Current State Response* (CSQ-CSR) sequence must be performed in order to determine the value of the boolean expression.

6.8.1 State Reference

The `IN` function is a boolean function used to determine if a particular node in the current state tree is in-scope. For example, consider the Statechart in Figure 6.12. The transition from X to Y has a boolean guard exhibiting the `IN` function.

For example, consider the Statechart in Figure 6.15. The boolean guard on the transition from A to B is a state reference. This gets converted into a complex sequence of steps.

$$\begin{aligned}
\delta_e(\Sigma, e) &= \varepsilon_1 \\
\delta_i(\varepsilon_1) &= \omega_1 \\
\lambda(\varepsilon_1) &= \langle \uparrow X = CSQ\langle \Delta = \Uparrow X \rangle \rangle \\
\delta_e(\omega_1, CSR) &= \begin{cases} \varepsilon_T & \text{if } CSR.\beta = 1 \\ \Sigma & \text{otherwise} \end{cases} \\
\delta_i(\varepsilon_T) &= \chi,\alpha \text{ sequence}
\end{aligned}$$

Figure 6.16 shows, graphically, how the transition in Figure 6.15 would be converted.

The CSQ-CSR Sequence

Every state can be queried to determine if it is current. Thus, at every state there exists a sequence which responds to a CSQ event and returns a CSR event. In order for the CSR to be returned to the sender the CSQ event must come equipped with a return address. Recall that DEVS events and states are structured sets in this framework. The return address is stored in the CSQ structured set's

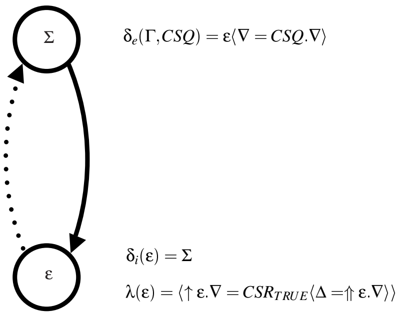

Figure 6.17: Current state query returns TRUE.

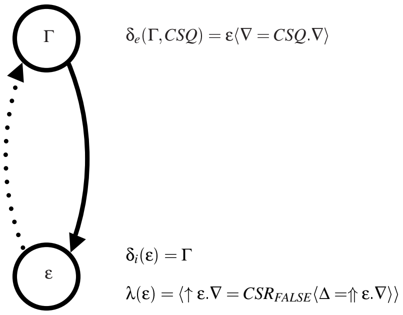

Figure 6.18: Current state query returns FALSE.

coordinate, ∇. This way, the queried terminal or non-terminal can simply set the address parameter of the CSR event to the return address of the CSQ event's ∇ field. The following is the reaction from the Σ state.

$$\begin{aligned}\delta_e(\Sigma, CSQ) &= \varepsilon\langle\nabla = CSQ.\nabla\rangle\\ \delta_i(\varepsilon) &= \Sigma\\ \lambda(\varepsilon) &= \langle\uparrow \varepsilon.\nabla = CSR_{TRUE}\langle\Delta = \Uparrow \varepsilon.\nabla\rangle\rangle\end{aligned}$$

The reaction from the Γ state is listed below.

$$\begin{aligned}\delta_e(\Gamma, CSQ) &= \varepsilon\langle\nabla = CSQ.\nabla\rangle\\ \delta_i(\varepsilon) &= \Gamma\\ \lambda(\varepsilon) &= \langle\uparrow \varepsilon.\nabla = CSR_{FALSE}\langle\Delta = \Uparrow \varepsilon.\nabla\rangle\rangle\end{aligned}$$

Notice that the ∇ value is stored in the ε state. This is required since λ is a function of ε. In other words, $\delta_e(\Gamma, CSQ) = \varepsilon\langle\nabla = CSQ.\nabla\rangle$ gives λ access to the return address, ∇. These two sequences can be see in Figures 6.17 and 6.18.

6.9 Transforming Hyper-transitions

Hyper-transitions employ the same mechanism required to execute a transition with source and destination set cardinalities equal to one. They use χ events to exit the source state and α events to enter the destination states. However, it is more difficult to determine where the transition starts. The transition is function of an event and x source states. A reaction to the event should not be

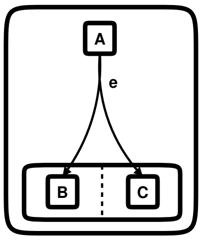

Figure 6.19: Hyper-transition with source set cardinality equal to one.

built into each source state. Knowing if all the source states 'heard' the event is difficult. Thus, this is not a good approach.

Another method utilizes a mechanism already described: the state reference function, `IN`. This approach is as follows. At some point in the hierarchy, a relay will 'listen' for the trigger event. Once this event occurs, the relay should continue to check the status of each source state by using the CSQ and CSR events. If at least one state is not current, the hyper-transition does not fire. Otherwise, the transition fires, and α events should be sent to each destination state.

6.9.1 Hyper-transitions with Source Set Cardinality Equal to One

Notice that, under this scheme, hyper-transitions with a source set cardinality equal to one and destination set cardinality greater than or equal to one may still be converted using the previous method for transitions. It is a source set cardinality greater than one that requires extra synchronization.

Consider the hyper-transition in Figure 6.19. A's δ_e will respond to event e in the normal way. However, once the source is exited, the relay must send two α events. The path from A to B is $\rho_{A,B}$ and the path from A to C is $\rho_{A,C}$.

$$\begin{aligned}
\delta_e(\Sigma, e) &= \varepsilon_1 \\
\delta_i(\varepsilon_1) &= \varepsilon_2 \\
\lambda(\varepsilon_1) &= \langle \uparrow \rho_{A,B} = \alpha\langle \Delta = \Uparrow \rho_{A,B} \rangle\rangle \\
\delta_i(\varepsilon_2) &= \Gamma \\
\lambda(\varepsilon_2) &= \langle \uparrow \rho_{A,C} = \alpha\langle \Delta = \Uparrow \rho_{A,C} \rangle\rangle
\end{aligned}$$

This sequence can be seen graphically in Figure 6.20.

6.9.2 Hyper-transitions with Source Set Cardinality Greater than One

This special case requires extra synchronization. As stated earlier, the state reference mechanism, CSQ-CSR sequence, is used to check if a particular hyper-transition with a source set cardinality greater than one can fire. Once this sequence is triggered by the trigger event, the status of every source state must be checked. If all x source states are in-scope, the transition can fire (or will check its guard if one exists). This involves sending x χ events, then waiting for x $\chi^{\star}$ acknowlengements. If at least one source state is not in-scope the transition does not fire.

Some thought must be put into where to install the event trigger. It must be added into the δ_e of the relay belonging to the deepest coupled DEVS which is the parent of all the source states. This is also known as the least common ancestor. In this location, the relay can pick up all narrowcasted trigger events which would cause this Statechart transition to fire.

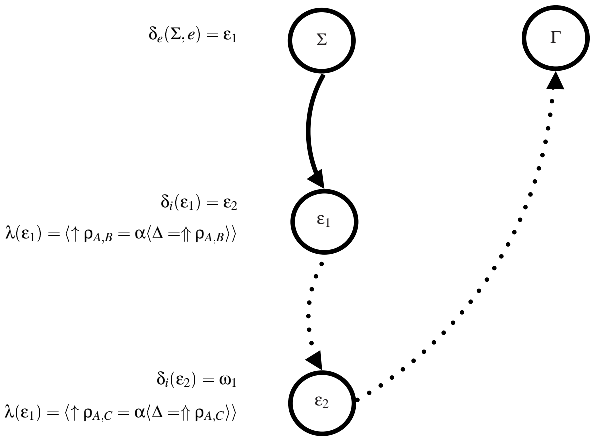

Figure 6.20: Hyper-transition DEVS sequence with source set cardinality equal to one.

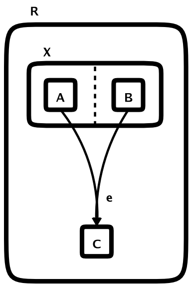

Figure 6.21: Hyper-transition with source set cardinality greater than one.

For example, consider the Statechart in Figure 6.21. There is a hyper-transition from state $A \times B$ to C which is triggered by event e. Notice that there would be a coupled DEVS, X with a relay named $RELAY_X$. The event trigger must be added to the δ_e belonging to $RELAY_X$. Thus, the δ_e within $RELAY_X$ must react to the trigger event.

This setup also applies to the general case. For any hyper-transition, there will always be some contour, X, which engulfs all the source states of the hyper-transition. This state must be current for the transition to fire. Thus, it is a natural place to install the hyper-transition mechanism.

Once the hyper-transition in Figure 6.21 is triggered, the following sequence of activities take place.

$$
\begin{aligned}
\delta_e(\Sigma, e) &= \varepsilon_1 \\
\delta_i(\varepsilon_1) &= \omega_1 \\
\lambda(\varepsilon_1) &= \langle A = CSQ\langle \Delta = \{\emptyset\}, \nabla = \{\perp\} \rangle \rangle \\
\delta_e(\omega_1, CSR_{FALSE}) &= \Sigma \\
\delta_e(\omega_1, CSR_{TRUE}) &= \varepsilon_2 \\
\delta_i(\varepsilon_2) &= \omega_2 \\
\lambda(\varepsilon_2) &= \langle B = CSQ\langle \Delta = \{\emptyset\}, \nabla = \{\perp\} \rangle \rangle \\
\delta_e(\omega_2, CSR_{FALSE}) &= \Sigma \\
\delta_e(\omega_2, CSR_{TRUE}) &= \chi, \alpha \text{ etc.}
\end{aligned}
$$

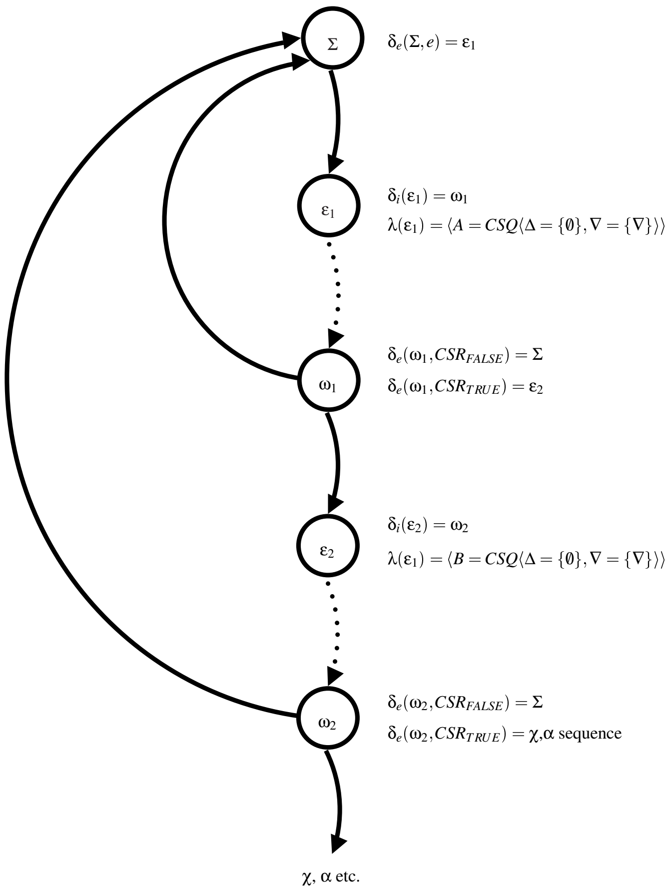

Figure 6.22: Hyper-transition with source set cardinality greater than one, converted to DEVS.

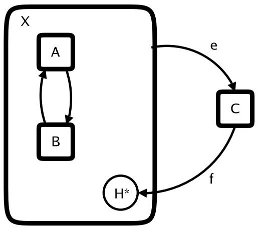

Figure 6.23: A Statechart with History.

A visual representation of this sequence can be seen in Figure 6.22. It is now clear that a CSQ-CSR sequence occurs for each source state. This is to ensure that each source state is current at the time the transition fires. The first CSR that returns a *FALSE* value forces the relay to stop processing the transition and to return to a Σ or Γ state.

This relay remembers if the sequence started at Σ or Γ by maintaining a `start` field in each state throughout the entire sequence. If, when a CSR response returns, the value of the response is FALSE, the relay must return to the state it was in previously. Thus, the δ_e function can simply examine the `start` field and take action appropriately.

6.10 Incorporating History

The History mechanism fits in nicely with the infrastructure already built. When a state with children wishes to exit, it signals to its children to exit as well. The exit-acknowledgement is returned from the state that was in-scope. This state is the last state to be in-scope before the parent state exited. Thus, the history information can be easily added as a field of the $\chi^\star$ event. When the parent receives this event, it stores the history value as a field in its current state.

A transition which terminates on a History node is converted to DEVS as follows. The transition will fire at the source state. An α event will be generated for the destination. The address field of the α event will have the history's parent contour at the bottom of the stack. This ensures that the α's destination is the parent contour of the history node.

Another important field specified in the α event is the *history* field. A value of 1 indicates to the relay that the α event means to enter the *history* state, not the *default* state. Once the relay sees the history field in the α event, it can look at the history field of its current state (obtained from the $\chi^\star$ event). The relay should then send an α event for this history node.

If a transition terminates on an $H^\star$ node, the transformed DEVS transition is very similar to one which terminates on an H node. The history information is received in exactly the same way for a $H^\star$ node. When a relay receives an α event which has an *history-star* field set to $True$, it will get the history information from the current state, just as with a normal history. However, the α event which is sent must have the *history-star* field set to $True$ again. This will cause the next sub-node to enter its history state, and so forth.

Consider the Statechart in Figure 6.23. Recall that while a History node looks like a state, it is not a state at all. A History node is a section of a transition. Thus, in a converted DEVS model, a History mechanism may appear to be elusive.

When X receives a χ event, it will send a χ to its sub-states. Either A or B will respond. Without loss of generality, let us say that B is the responding state. B will store history information in the H field of the $\chi^\star$ event as follows.

$$
\begin{aligned}
\delta_e(\Sigma,\chi) &= \varepsilon \\
\delta_i(\varepsilon) &= \Gamma
\end{aligned}
$$

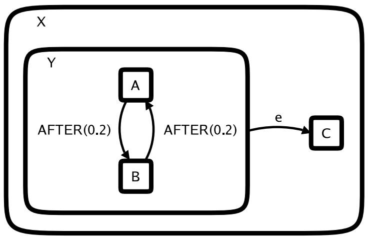

Figure 6.24: The resulting DEVS model can deadlock.

$$\lambda(\varepsilon) \quad = \quad \langle \perp = \chi^{\star}\langle H = B\rangle\rangle$$

The history information is gathered in $RELAY_X$ from the $\chi^{\star}$ event as follows.

$$\delta_e(\omega, \chi^{\star}) = \Gamma\langle H = \chi^{\star}.H\rangle$$

When an α is received such that $\alpha.H = 1 \wedge \alpha.\beta = 1$, the relay responds as follows.

$$\begin{aligned}
\delta_e(\Gamma, \alpha) &= \varepsilon\langle H = \Gamma.H\rangle \text{ if } \alpha.H = 1 \wedge \alpha.\beta = 1 \\
\delta_i(\varepsilon) &= \Sigma \\
\lambda(\varepsilon) &= \langle \varepsilon.H = \alpha\langle \beta = 1, H = 1, \Delta = \{\emptyset\}\rangle\rangle
\end{aligned}$$

In the case of deep children, the α events will be continually sent down the DEVS hierarchy until terminal states are reached. If a terminal state receives an α event such that $\alpha.H = 1 \wedge \alpha.\beta = 1$ it can treat it as a normal α event, since it has no children.

An α such that $\alpha.H = 1 \wedge \alpha.\beta = 0$ follows almost exactly the same sequence. The difference occurs when sending the α event to the history state. Instead of sending another α such that $\alpha.H = 1 \wedge \alpha.\beta = 0$ to perpetuate the entering of history states, an α event such that $\alpha.H = 0 \wedge \alpha.\beta = 0$ is sent. This way, the history state is entered and if it has sub-states, their default states will be entered.

6.11 Event Queueing

One small detail remains which is vital to the implementation of the converted DEVS models. A DEVS simulator is a time-slicing simulator and treats concurrent behaviour sequentially. It executes each atomic DEVS as if they are concurrent objects. Concurrent programs are subject to race conditions and deadlock. In the framework described, race conditions and deadlock can occur in several places.

Consider the Statechart in Figure 6.24. The DEVS model converted from this model can deadlock. Before an explanation is given as to how this may occur, first look at the skeleton of the resulting DEVS model in Figure 6.25.

When the transition from A to B fires, a sequence of steps occur. Eventually, the atomic model, A, will send an α event to B. When this happens, A sends the α event to its parent, then sets its current state to Γ. Now, neither A nor B represent an in-scope state. At this time, the transition from Y to C may fire. If this happens, $RELAY_X$ will send a χ to $RELAY_Y$ which will in turn broadcast another χ to both A and B. However, recall that the response to χ is as follows.

$$\delta_e(\Sigma, \chi) = \ldots$$

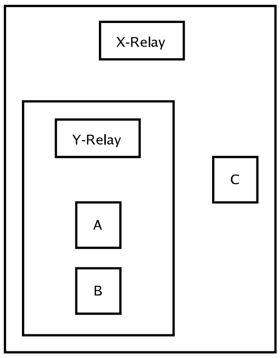

Figure 6.25: The DEVS skeleton of the Statechart in Figure 6.24.

The only sub-model which will respond has a current state Σ. But in this case, no sub-model has a current state Σ. Both A and B are out-of-scope. Thus, the event is forgotten and a $\chi^{\star}$ is never returned. This causes $RELAY_X$ and $RELAY_Y$ to wait in some ω state forever.

The general solution uses event queues and locks. This idea is as follows. When a model is in an intermediate state it is locked. All incoming events are queued and dealt with once the model has finished its job. When finished, it returns to either the Σ or Γ state. If the event queue is not empty, there is more work to be done. In this case, the model must continually dequeue events and process them until the event queue is empty. Note that when all event queues are empty throughout the entire DEVS model, a *macro step* has completed.

The event queue is stored in a field of the current state as Q. Q is an ordered set. Moreover, the δ_e function now has an extremely simple definition. Its sole job is to funnel events into the event queue and maintain the current state.

$$\delta_e(k,x) = k\langle Q = k.Q \Downarrow x\rangle$$

It is also necessary that there exist dequeue states, ε_Σ and ε_Γ such that,

$$\begin{aligned}\delta_i(\Sigma) &= \varepsilon_\Sigma\\ \delta_i(\Gamma) &= \varepsilon_\Gamma\end{aligned}$$

This ensures that after returning to a stable state of Σ or Γ, the model will move to the dequeue state and continue to handle events.

It is only necessary to move to the dequeue state if there are events in the queue. Thus, the *ta* function can be defined to stay in a stable state if there are no events, otherwise return a time advance of 0.

$$\begin{aligned}ta(\Sigma) &= \begin{cases}\infty & \text{if } |\Sigma.Q| = 0\\ 0 & \text{otherwise}\end{cases}\\ ta(\Gamma) &= \begin{cases}\infty & \text{if } |\Gamma.Q| = 0\\ 0 & \text{otherwise}\end{cases}\end{aligned}$$

Under this configuration, all the definitions that were specified for the δ_e function are now moved to the δ_i function under the dequeue states, ε_Σ and ε^Γ. That is, if the current state is a dequeue state, an event is dequeued, examined and action is taken. These actions occurred previously in the δ_e function, but now occur in the δ_i function.

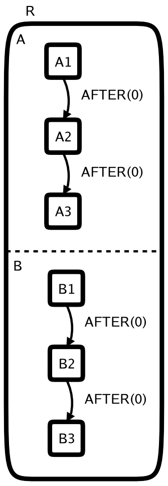

Figure 6.26: Using the *select* function for orthogonal priority.

6.12 The Use of the *select* Function

The *select* function can provide very useful variations in the generated models' semantics. The most evident of these uses is for orthogonal components. Orthogonal components represent a product state set. They are independent systems running concurrently. Thus, an execution environment with a single executor must somehow multiplex the activities within adjacent orthogonal components. The pythonDEVS-RT executor is one such environment. It must decide which transition will fire next if several are scheduled to fire at the same time. This is precisely what happens in orthogonal components. Transitions are scheduled to fire at the same time, and the *select* determines which will fire next.

Consider the Statechart in Figure 6.26. The total transition execution sequence depends on which semantics are used. Perhaps $A3$ is reached before $B2$ is reached or vice versa. The meaning of this Statechart depends on its use and the version of the semantics used to execute it.

Most desired semantics can be attained by carefully defining the *select* function. If A should take priority over B then the *select* for the coupled DEVS corresponding to R should be defined as:

$$select(E) = A$$

where E is the set of conflicting components $\{A, B\}$. Another intuitive interpretation of how to execute the Statechart in Figure 6.26 is to execute transitions from A and B in turn. That is, execute A's transition, then B's then A's, etc. Perhaps, the next transition to be executed should be determined by sampling probabilities from a random distribution. The possibilities are endless.

Another use of the *select* function is the implementation of inside-transition-first or outside-transition-first semantics. The Statechart in Figure 6.27 will have a conflict if the current state is $B1$. Which transition fires, and what is the resulting current state again depend on the semantics used. If outside-transition-first semantics are used, the current state becomes $A1$. If inside-transition-first semantics are used, the current state changes to $B2$.

To use outside-transition-first semantics, simply specify *select* to return the relay which is closest to the root. Terminals have the lowest priority. To use inside-transition-first semantics, terminals have the highest priority. Thus the *select* function should return either a terminal or the deepest relay in the hierarchy.

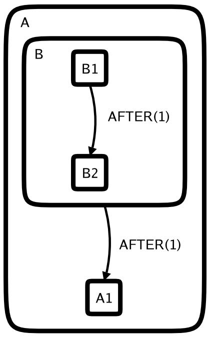

Figure 6.27: Using the *select* function for transition priority.

6.13 Putting it All Together

This final section illustrates how all the mechanisms and sequences described work together. It also shows the final rough structure of the δ_e and δ_i functions. This will be done using Python code with ... to extrapolate areas of the code which are repetitive.

Notice that there are several class member variables. These can be thought of as coordinates of the current state structured set. The values of these coordinates are continuously and implicity passed on to the new current state in every δ_e and δ_i function. These member variables help simplify the implementation.

The following is a code listing for typical DEVS components. The following class *TypicalAtomic* illustrates a typical atomic DEVS. Relays and terminals are very similar in structure. The following is only to give the reader a sense of how the mechanisms described above work together and are structured in one of many possible implementations.

```
class TypicalAtomic(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.current_child = None      # track which sub-state is current
    self.history = 'PLAYING'       # initialize the history
    self.Q = []                    # the event queue
    self.delta_ext = 0             # has the delta external function fired?
    self.inscope = 0               # is the current context INSCOPE?
    self.scope = 2                 # what is out depth in the hierarchy
    self.timeout = 0               # has a timeout occured?
    self.elapsed_since_inscope = 0 # the elapsed time since we've been INSCOPE
    self.ports = {}                # add ports
    self.ports['_IN_'] = self.addInPort('_IN_')
    self.ports['_PARENT_'] = self.addOutPort('_PARENT_')
    self.ports['_ALL_'] = self.addOutPort('_ALL_')
    self.states = {}               # add states
    self.states['INSCOPE'] = DEVSstate('INSCOPE')
    self.states['OUTSCOPE'] = DEVSstate('OUTSCOPE')
    ...
    self.state = self.states['OUTSCOPE']

  def extTransition(self):
    if self.inscope: self.elapsed_since_inscope += self.elapsed # time has passed
```

```
    else: self.elapsed_since_inscope = 0 # reset
    event = self.peek(self.ports['_IN_']) # get incoming event
    self.Q.append(event) # append event to the event queue
    self.delta_ext = 1   # an external transition has occured
    return self.state    # return the same state as when we started

  def intTransition(self):
    if self.inscope: self.elapsed_since_inscope += self.elapsed # time has passed
    else: self.elapsed_since_inscope = 0 # reset
    if not self.timeout and round(self.elapsed_since_inscope, 6) >= 0: # a TIMEOUT has occure
      self.Q.append(EVENT('_TIMEOUT_')) # append the TIMEOUT event to the event queue
      self.elapsed_since_inscope = 0 # reset
      self.timeout = 1 # a timeout has occured
    def get_new_state(): # wrapper for getting the new state
      if self.state == 'INSCOPE_DQ': # dequeue and process an event
        while len(self.Q) > 0:
          self.event=self.Q[0];del self.Q[0] # dequeue
          if self.event == 'X1': return self.states['epsilon1433']
          elif self.event == '_TIMEOUT_': return self.states['epsilon1449']
          elif self.event == 'EXITALL': return self.states['epsilon1427']
          ... # other states here
          else: return '__relay_hook' # route the event elsewhere
      elif self.state == 'OUTSCOPE_DQ':
        ... # dequeue and process an event
      elif self.state == 'epsilon1446': # do something
        return self.states['OUTSCOPE']
      elif self.state == 'epsilon1434': # do something
        return self.states['epsilon1437']
      ... # other states here
    new_state = get_new_state() # get the new state
    if new_state == 'INSCOPE': # check if context is INSCOPE
      self.inscope = 1
      if len(self.Q) == 0: self.timeout = 0 # no timeouts
    elif new_state == 'OUTSCOPE': # check if context is OUTSCOPE
      self.inscope = 0
      if len(self.Q) == 0: self.timeout = 0 # no timeouts
    return new_state # return the new state to the simulator

  def timeAdvance(self):
    if self.state == 'INSCOPE' and len(self.Q) > 0: ta = 0
    elif self.state == 'OUTSCOPE' and len(self.Q) > 0: ta = 0
    elif self.delta_ext: ta = 0;self.delta_ext=0
    elif self.state == 'epsilon1427': ta = 0
    elif self.state == 'OUTSCOPE': ta = INFINITY
    ...
    if self.inscope and not self.timeout:
      ta_after = max(0, 5-self.elapsed_since_inscope) # this is from an AFTER(5) trigger
    else:
      ta_after = INFINITY
```

```
    return minEx(ta, ta_after)

  def outputFnc(self):
    if self.state == 'epsilon1454': # particular state
      def __outputFnc(self):
        event = DEVSevent('ENTER')
        event.set_param('address', ['_PARENT_', 'STOPPED'])
        event.set_param('return_address', ['_PARENT_', 'CAN_SEEK'])
        return event
      event = __outputFnc(self) # get the output event
      port_name = event.get_param('address')[0] # get the port the event should be poked out
      del event.get_param('address')[0] # dequeue the address list
      self.poke(self.ports[port_name], event) # poke the event
    ...
```

The next short code listing illustrates a typical coupled DEVS in class *TypicalNonterminal*.

```
class TypicalNonterminal(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    self.ports = {}
    self.ports['_IN_'] = self.addInPort('_IN_')
    self.ports['_PARENT_'] = self.addOutPort('_PARENT_')
    self.relay=self.addSubModel(RELAY_TypicalNonterminal())
    self.A = self.addSubModel(SUBSTATE_CLASS1())
    self.B = self.addSubModel(SUBSTATE_CLASS2())
    self.relay.ports['A'] = self.relay.addOutPort('A')
    self.relay.ports['B'] = self.relay.addOutPort('B')
    ...
    self.connectPorts(self.relay.ports['A'], self.A.ports['_IN_'])
    self.connectPorts(self.A.ports['_PARENT_'], self.relay.ports['_IN_'])
    ...
  def select(self, imm):
    min_scope = INFINITY
    highest = imm[0]
    for devs in imm:
      if is_relay(devs):
        if devs.scope < min_scope: highest = devs; min_scope = devs.scope
    return highest
```

6.14 Conclusion

This chapter illustrated, in some detail, a method of converting models in the Statechart formalism to DEVS. This procedure opens up the possibility for using DEVS as a target specification from other, not necessarily discrete event, formalisms. Fine tuning concurrent semantics becomes extremely easy using the `select` function. The final DEVS models can be simulated using `pythonDEVS` or executed in real-time using `pythonDEVS-RT`.

7

Code Generator

The code generator provides a programming interface to transform a Statechart into a DEVS model. This chapter will first examine the static representations that the code generator uses. This includes all classes and static design information. Finally, the execution of the code generator is examined. This illustrates how Statechart models finally end up as DEVS models.

7.1 Static Representations

The code generator converts valid Statechart models into valid DEVS models usable by the pythonDEVS and pythonDEVS-RT packages. The code generator uses a canonical Statechart data structure representation. This representation features two different objects which reference each other in a tree-like structure. A list of the classes used and a brief description of what each represents is given below.

- `SC2DEVS_NODE`: `Basic` and `Composite` states, `History` nodes
- `SC2DEVS_REGION`: `Orthogonal` component
- `SC2DEVS_TRANSITION`: `Hyperedge` relation
- `SC2DEVS_TRIGGER`: A `Hyperedge` trigger
- `SC2DEVS_GUARD`: A `Hyperedge` boolean guard expression
- `SC2DEVS_COUPLED_DEVS`: A coupled DEVS model
- `SC2DEVS_ATOMIC_DEVS`: An atomic DEVS model
- `SC2DEVS_CONTAINER`: A non-terminal Statechart state as an atomic DEVS model
- `SC2DEVS_TERMINAL`: A terminal Statechart state as an atomic DEVS model

The class diagram in Figure 7.1 shows the static relationships between each of the Statechart components. Notice that a node can reference another node. Also, nodes and transitions reference each other. Transitions reference trigger and guard objects. Regions are place holders for orthogonal components.

The structure of the resulting DEVS models contains only two types of objects, `SC2DEVS_CONTAINER` and `SC2DEVS_TERMINAL`. These objects form a tree where the terminal models form the leaves of the tree and the container models are the intermediate nodes. The class diagram for the DEVS models can be seen in Figure 7.2. Note that every container has, as a member variable, a `SC2DEVS_RELAY` instance. This instance is the *relay*.

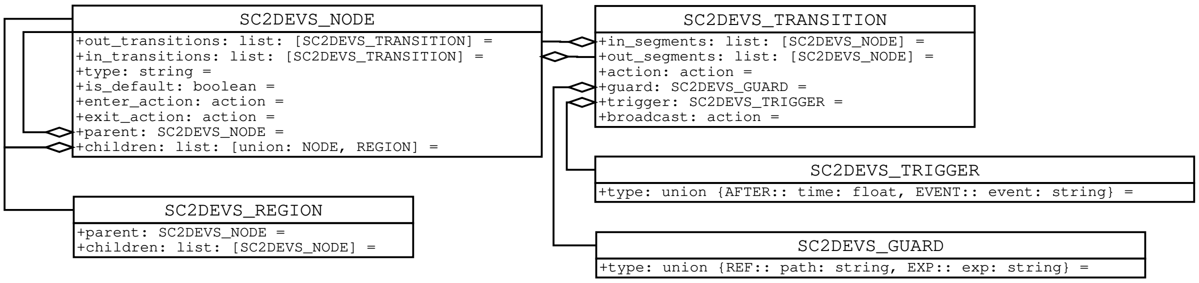

Figure 7.1: Statechart code generator class diagram.

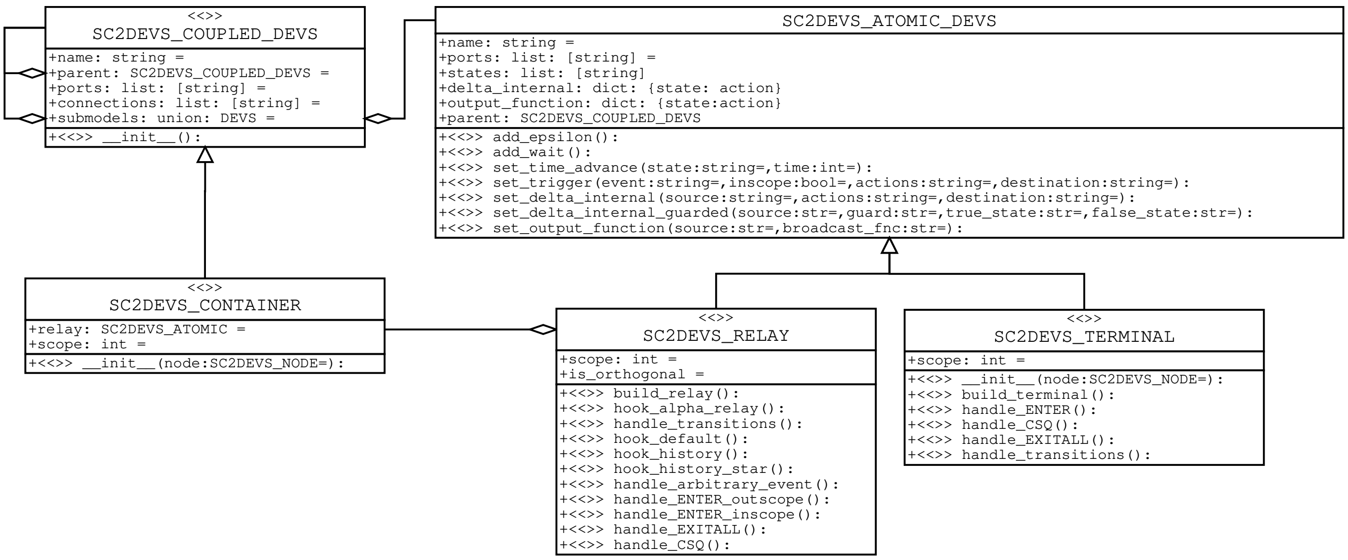

Figure 7.2: DEVS code generator class diagram.

The Statechart models built using AToM^3 include several unnecessary pieces of information, such as graphical information. Thus, the code generator starts by rebuilding the Statechart model using its internal representation, keeping only the relevant model details. Once the canonical representation is built, the transformation begins using several passes. A new DEVS representation is constructed. Finally the DEVS model is output in one file, compatible with pythonDEVS.

7.2 The Execution of the Code Generator

As indicated above, the code generator starts by building a representation of the Statechart model to be transformed. This representation provides efficient access to important data. Next, the syntax of the Statechart is verified. If the syntax is not correct, the code generator prints the error messages and exits. If the syntax is correct, the transformation ensues and the resulting DEVS model is written to disk.

The top-level sequence of calls are as follows.

1. `build_statechart`
2. `verify_syntax`
3. `build_devs`
4. `emit_devs`

7.2.1 `build_statechart`

`build_statechart` simply takes the AToM^3 data-structure and returns the same Statechart, but expressed using its cannonical Statechart representation. The value returned is an instance of `SC2DEVS_NODE` which represents the root of the Statechart. All other nodes, transitions, triggers, etc. are reachable from this root node.

7.2.2 `verify_syntax`

Since the implementation in this thesis uses meta-modelling, all syntactic information is specified at the meta-model level. This makes syntax verification at the code generation stage unnecessary. However, this step is included for compatibility with other tools which do not have meta-modeling capabilities.

7.2.3 `build_devs`

The next step, `build_devs`, builds the final DEVS model. This function consists of one pass through the Statechart hierarchy and returns the root DEVS model. The root is an instance of class `SC2DEVS_CONTAINER`. The root is always wrapped by a coupled DEVS in case the root, as the modeller sees it, consists of orthogonal regions. All sub-models can be accessed from the root coupled DEVS.

The function does a depth first traversal on the Statechart data-structure. A Statechart is a higraph and so the depth first traversal must mark nodes as they are discovered so that they are not processed again. If a node is a basic state, a `SC2DEVS_TERMINAL` is built. Otherwise, a `SC2DEVS_CONTAINER` is built, and the function recurses on all the children.

The `SC2DEVS_TERMINAL` and `SC2DEVS_RELAY` classes each basically have one large constructor. The constructor is separated into several smaller methods, each of which builds a particular hook or adds a response to a particular event.

The methods that start with `hook` are sequences that start at an epsilon state, ε, such that $ta(\varepsilon) = 0$. They provide a convenient way to latch into a sequence which performs reusable actions. The sequence will performs these actions then attempt to resume stability by ending finishing at either Σ or Γ. *

Methods that start with `handle` build a response to a particular event. These sequences always start at one of the dequeue states ε^{Σ} or ε^{Γ}. They, too, attempt to resume stability (finish the macro-step) by ending their sequence at one of Σ or Γ.

Transitions are built as they are encountered. The triggers and guards are built as needed. For example, a transition which is triggered by event *e* will be transformed such that the existence of the *e* event is checked in the delta internal function for the inscope dequeue state, ε^{Σ}. This is where history nodes are processed. A history node is actually a segment of a transition. Thus, if a history nodes is the destination of a particular transition, that transition sequence will then latch onto the `history_hook` sequence.

7.2.4 `emit_devs`

Finally, the emit function writes the DEVS model to a file which is readable by the `pythonDEVS` packages. The final emitted DEVS model may then be combined with other DEVS models or with the `Generator` interface.

The model on its own cannot be executed. The modeller must pass an instance of the root coupled DEVS to an `Executor` instance for the model to be executed in real-time. This also allows the modeller to use the root Statechart-DEVS model as a sub-model of some other coupled DEVS.

*The end of a sequence depends on which sequence is being executed.

8
Case Studies

The chapter discusses simple models built using the Statechart meta-model for $AToM^3$ and their subsequent execution using pythonDEVS-RT. Each model is an engine for one or more client modules. The client sends events to the engine. The engine will then return events to the client. In this way, thin clients are built, and the core dynamics of the total application are modelled using Statecharts.

8.1 Crosswalk

This application features a graphical client which emulates a pedestrian crosswalk and the Statechart which models its dynamics. The graphical interface is seen from the pedestrian's perspective: there is a traffic light and a crosswalk button (which shortens the time of a red light). There are also buttons that a police officer may use. These buttons include an interrupt which causes the traffic light to flash yellow and an on/off switch.

The Statechart which models the dynamics of this system can be seen in Figure 8.1. The states' enter and exit are visible from this diagram. Thus, these actions have been sumarized in Table 8.1. Note that the exit actions have not been specified because no state has an exit action.

The class diagram of the entire system is illustrated in Figure 8.2. From the class diagram it is easy to see how both `CrosswalkGUI` and the `Executor` class depend on the `SYNC` object. Events are `poke`d from the graphical client to the `ROOT` DEVS. Then the client signals the `SYNC` object which wakes up the `Executor` to handle the event. The Statechart model will use `send` to relay events

Table 8.1: Crosswalk states' enter actions

State	Enter action
`RED`	`GUI.send('RED')`
`GREEN`	`GUI.send('GREEN')`
`YELLOW`	`GUI.send('YELLOW')`
`OFF`	`GUI.send('DEAD')`
`YELLOW_ON`	`GUI.send('YELLOW')`
`YELLOW_OFF`	`GUI.send('DEAD')`

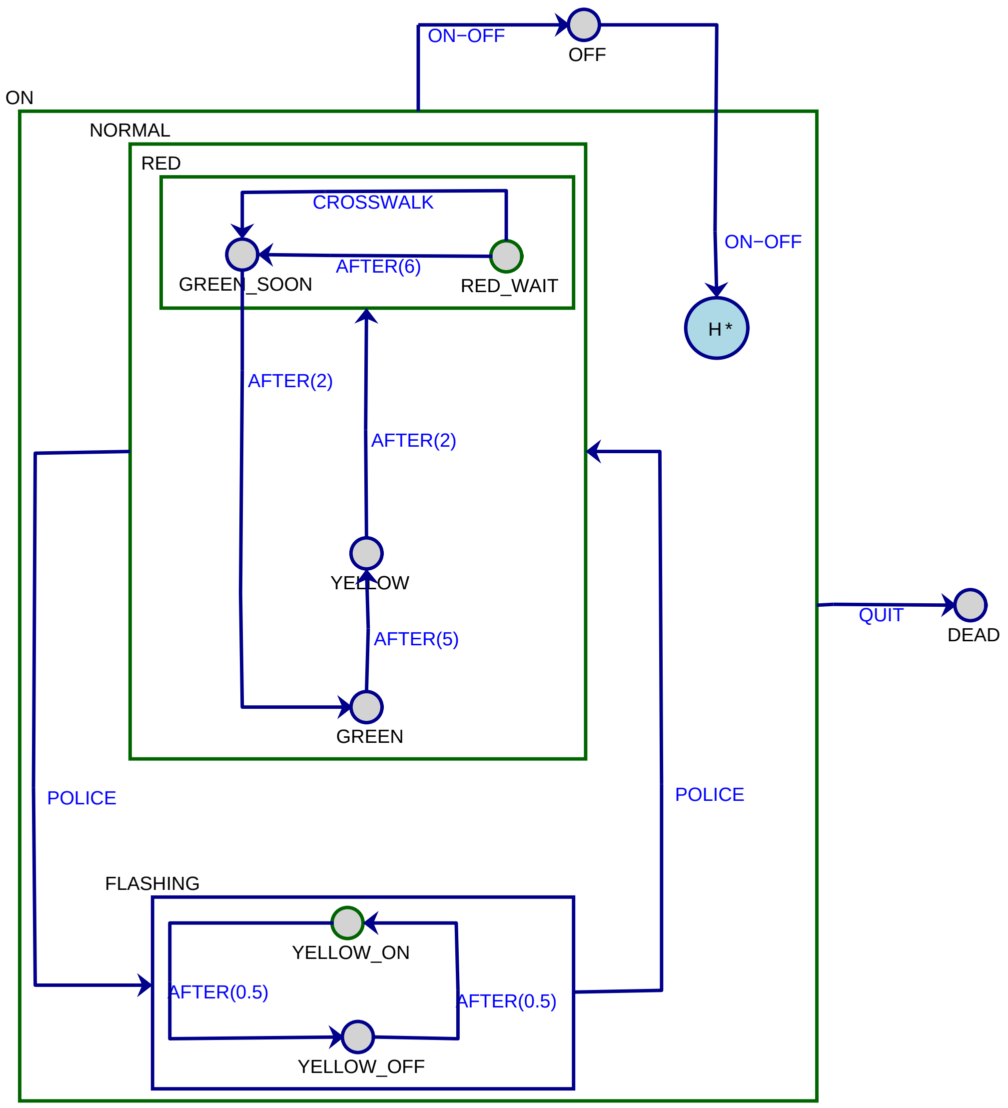

Figure 8.1: The Statechart crosswalk model.

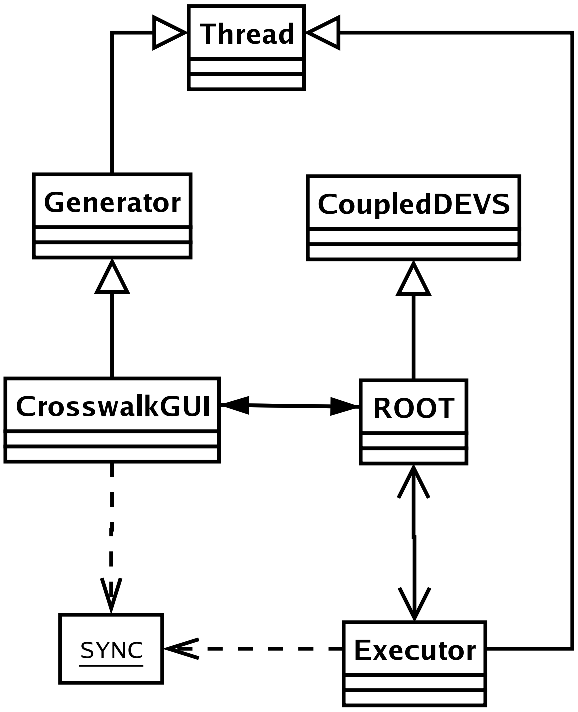

Figure 8.2: The crosswalk application class diagram.

back to the graphical client.

A screenshot of the graphical client can be seen in Figure 8.3.

8.2 An MP3 Player

This is a simple application capable of playing audio files (usually MP3). The application has a list of requirements which must be considered when building the Statechart model. The list of requirements is as follows.

- Action requirements
  1. Load: Load an MP3 file when the application starts.
  2. Play: Play the current, loaded mp3 file. Update current time.
  3. Pause: If Playing, pause playing. Time stops.
  4. Stop: Stop playing: Time reset to 0:0.
  5. Seek forward: If playing or paused, move time forward faster. If the end of the track is reached, stop playback and set time to 0:0.
  6. Seek reverse: If playing or paused, move time backwards, quickly. If the beginning of the track is reached, stop playback and set time to 0:0.
  7. Off: Stops playing, closes environment.
- Display requirements
  1. A button for each ACTION
  2. File: The current loaded file
  3. Time: The current time (mins:secs)

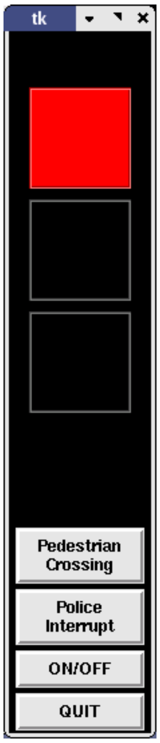

Figure 8.3: The crosswalk graphical client.

4. Total: The length in time of the current track (mins:secs)

There also exists a `MP3AudioDriver` package which the Statechart model can use to gain access to an MP3 decoder and audio devices. The package provides simple functionality such as `play`, `stop`, `pause` etc. For more details, refer to the `Driver` class in Figure 8.4.

In total, there are three layers to the application. Figure 8.5 illustrates each of these layers. The high level layers are on the top and the low level layers are on the bottom.

The dynamics of the control layer are modelled using the Statechart formalism. The Statechart used for this application can be seen in Figure 8.7. Since the enter and exit actions are not visible from the Statechart diagram, they have been summarized in Table 8.2. Note that the exit actions are not specified because no state in the model has an exit action. Figure 8.6 shows the appearance of the graphical windows which is created when the application starts. The buttons correspond to the action requirements and the left text area corresponds to the display requirements.

The seeking functionality uses careful synchronization between orthogonal regions. This ensures that when the `Display` region finishes seeking, the `Audio` region may carry out the seek operation.

Verification of each requirement is simplified by referring to the Statechart model. One can use a coverability graph to determine the sequence of actions required for a particular state to be reached.

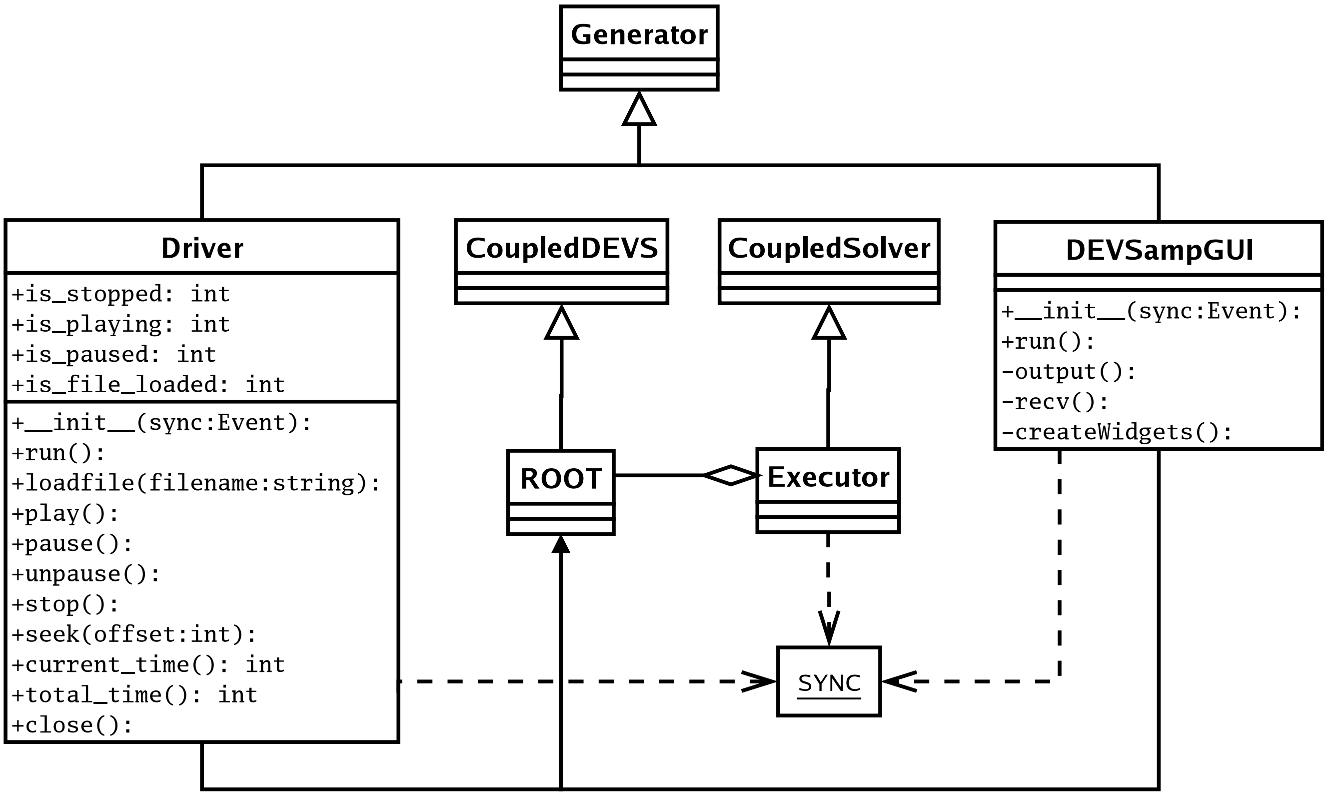

Figure 8.4: The MP3 application class diagram.

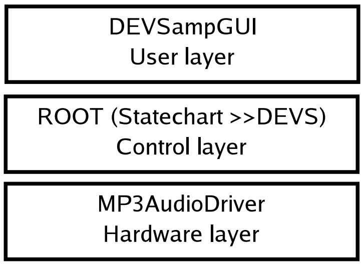

Figure 8.5: The MP3 application's layers.

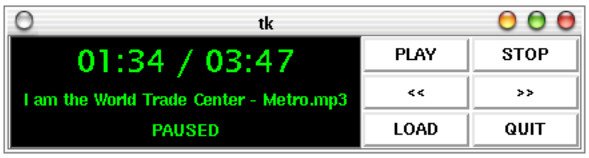

Figure 8.6: The MP3 application's graphical appearance.

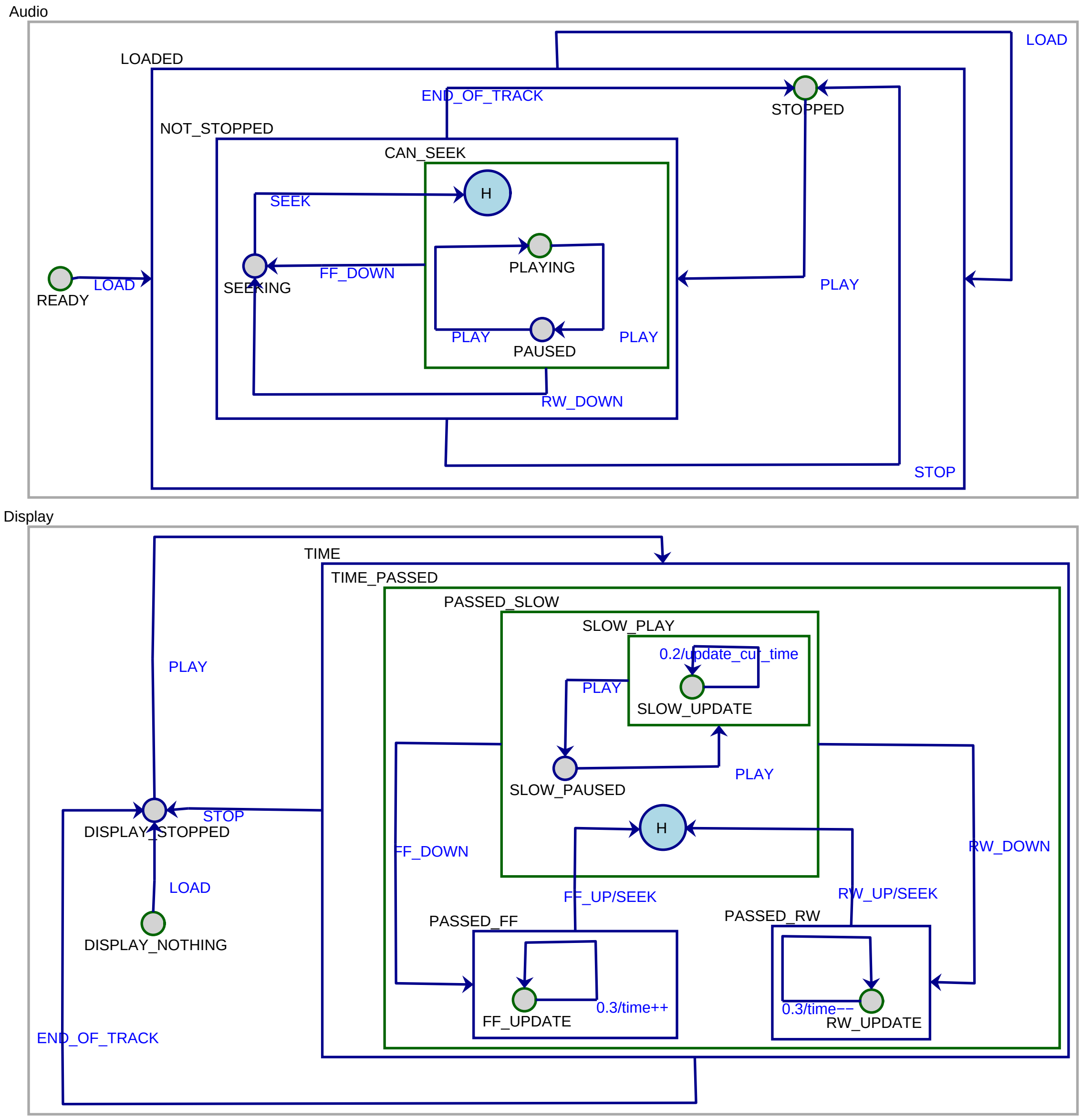

Figure 8.7: The MP3 application’s Statechart control dynamics.

Table 8.2: DEVSamp states' enter actions

State	Enter action
SEEKING	`if DRIVER.is_playing(): DRIVER.pause()`
PLAYING	`DRIVER.play()`
PAUSED	`DRIVER.pause()`
STOPPED	`DRIVER.stop()`
DISPLAY_NOTHING	`GUI.send(EVENT("DISPLAY", [("info", "LOAD FILE"), ("state", ""), ("time", (0,0))]))`
DISPLAY_STOPPED	`filename = self.event.get_param('filename'); GUI.send(EVENT("DISPLAY", [("time", (0,0)),("info", filename),("state", "STOPPED")])); GUI.send(EVENT("BUTTON", [("playpause", "PLAY")]))`
SLOW_PLAY	`GUI.send(EVENT("DISPLAY", [("state", "PLAYING")])); GUI.send(EVENT("BUTTON", [("playpause", "PAUSE")]))`
SLOW_PAUSED	`secs = round(DRIVER.current_time()/1000, 0); mins = secs/60; secs = secs%60; GUI.send(EVENT("DISPLAY", [("state", "PAUSED"), ("time", (mins, secs))])); GUI.send(EVENT("BUTTON", [("playpause", "PLAY")]))`
PASSED_FF	`seek = DRIVER.current_time(); total_time = DRIVER.total_time(); GUI.send(EVENT("DISPLAY", [("state", "FAST FORWARD")]))`
PASSED_RW	`seek = DRIVER.current_time(); GUI.send(EVENT("DISPLAY", [("state", "REWIND")]))`

9
Conclusions

Conclusions will now be made regarding the topics discussed and contributions made in this thesis document. These topics and contributions are as follows.

- The Statecharts to DEVS transformation
- The Statechart meta-model
- DEVS code generation from Statechart model

9.1 Statecharts to DEVS Transformation

Transforming Statecharts to DEVS can provide an easy way to make large changes in semantics. It also allows for combining with other formalisms which use DEVS as a target specification.

9.1.1 Semantic Variations

This transformation can be a useful way of fine tuning the operational semantics of the high level Statechart models. One example of this involves specifying the priorities of transitions in conflict. These conflicts cannot be determined by looking at the static model. The conflicts occur when the model is executed. The `select` function is the perfect mechanism for dealing with conflicting Statechart transitions.

Deciding between *inside transitions first* vs. *outside transitions first* semantics is the most intuitive time to use some conflict resolution mechanism. For example, if event *e* occurs in the Statechart in

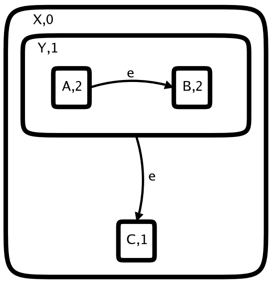

Figure 9.1: Potentially conflicting transitions.

Figure 9.1 and the current state is A, the new current state could either be B or C. Some semantics specify that inside transitions should take priority over outside transitions, while other semantics dictate exactly the opposite. The semantics used depends on the end usage and it would be nice to be able to have control over this option.

If the depth of each node is known then making *inside* vs. *outside* decisions is easy. The number after the comma for each state in Figure 9.1 indicates the depth of the node. For *outside transitions first* the select function chooses the node with the smaller depth. For *inside transitions first* the select function chooses the node with the larger depth. For example, in Figure 9.1 an *outside transition first* scheme would choose C as the new current state since,

$$depth(C) < depth(A) \Longleftrightarrow TRUE.$$

Where the select function is as follows. Note that $\{A,C\}$ is the input set, since A and C are the atomic models in conflict.

$$select(\{A,C\}) \quad = \quad \begin{cases} C & \text{if } depth(C) < depth(A) \\ A & \text{otherwise} \end{cases}$$

9.1.2 DEVS as a Target Specification

Consider models $X_1, X_2, \ldots, X_n$. Each X_i model is specified in a unique formalism. If each can be converted to an equivalent DEVS model, X_i', then each can be bound into one coupled DEVS, X. Each X_i' may communicate anyway that the modeller wishes.

This is not restricted to discrete event formalisms. There is work being done to convert Ordinary Differential Equations to DEVS [11]. This gives way to a hybrid environment in which a particular physical system may be modelled using a continuous or discrete event formalism. More work is needed in this area.

9.2 The Statechart Meta-model

A Statechart meta-model was presented with several syntactic constraints. This meta-model immediately enables automatic generation of an environment in which syntactically correct Statechart models can be built.

9.3 DEVS code generation

A code generator was also presented. It takes as input a Statechart specification and automatically outputs a DEVS model. The DEVS models output can be simulated using the pythonDEVS simulation package or they can be executed in real-time using pythonDEVS-RT.

9.4 Final Remarks

Statecharts are a very usable formalism for specifying the dynamics of physical systems. They are widely used in software systems and electronics design. Coupled with a transformer to DEVS and a real-time execution environment, Statecharts can be used in many different instances. They can be injected almost anywhere control logic is needed.

More research is needed in the area of multi-formalism modelling. This work will no doubt lead to interesting advances in modeling and simulation.

A
Notation

Here, a notation and framework will be introduced which enriches and simplifies the DEVS formalism and simplifies model descriptions. This framework applies to the input and output sets, X and Y, respectively. It would be nice if each DEVS, atomic or coupled, had *ports* through which events could enter or exit that particular model. The input-to-output translation function, Z, would then connect ports together. The connections could be thought of as channels where events could flow.

In order to define DEVS with ports, the input and output sets, X and Y are defined as *structured sets*. These structured sets use named coordinates and an operator to access information more easily. This framework was first developed by Jean Sébastien and is extended below.

Structured Set - Definition 1 *Let C be the set of valid coordinates. This can be thought of as the set of valid port names. Let $n \geq 0$ and $U^n \subseteq C^n$ be the set of all valid n-tuples of coordinates:*

$$U^n = \left\{(c_1, c_2, \ldots, c_n) \,|\, c_i \in C \wedge c_i = c_j \Rightarrow i = j, i, j \in \{1, 2, \ldots, n\}\right\}$$

Let $V = V_1 \times V_2 \times \ldots \times V_n$, a cartisian product of n sets of values where $\emptyset \notin V_i, \forall i$. V can also be written as:

$$V = \{(v_1, v_2, \ldots, v_n) \,|\, v_i \in V_i, i = 1, 2, \ldots, n\}$$

An element of U^n is an n-tuple or coordinates:

$$u = (c_1, c_2, \ldots, c_n)$$

Finally, let S be a structured set with respect to V if it is of the form $S = \{u\} \times V$, or:

$$S = \{((c_1, c_2, \ldots, c_n), (v_1, v_2, \ldots, v_n)) \,|\, v_i \in V_i, i = 1, 2, \ldots, n\}$$

Projection Operator - Definition 2 *The projection operator, ".", is used to associate a coordinate from U^n with a value from V_i for some i. The projection operator is a binary operator defined as follows.*

$$. : S \times C \longrightarrow \left(\bigcup_{i=1}^{n} V_i\right) \cup \{\emptyset\}$$

The projection operator can be used on an element $s \in S$, $s = ((c_1, c_2, \ldots, c_n), (v_1, v_2, \ldots, v_n))$, as follows.

$$s.c = \begin{cases} v_i & \text{if } \exists i \in \{1, 2, \ldots, n\} \mid c = c_i \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

With these definitions, DEVS with ports can now be constructed. X and Y are defined as structured sets. The named coordinate set, U^n, is the set of named ports and the sets of values, V_i, are the sets of events associated with a particular port.

Event Structured Set - Definition 3 *All events in a DEVS converted from a Statechart model are structured sets. The coordinate set used is a 4-tuple.*

$$(\Psi, \Delta, \beta, H)$$

Ψ is a string which is the name of the event. Δ is the address field. Note that if X is the set of all port names then $\Delta \in 2^X$ and Δ is ordered. $\beta \in \{0, 1\}$ indicates if the event should be broadcast. $H \in \{0, 1\}$ indicates if the α event should enter a history state or not.

Events - Notation 1 *$e\langle M\rangle$ is a short hand notation for an element of a structured set. M is a list of expressions of the form $a = b$. $e\langle a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n\rangle$ is short for the following.*

$$((\Psi, a_1, a_2, \ldots, a_n), (\text{"}e\text{"}, b_1, b_2, \ldots, b_n)) \text{ where } b_i \in V_i$$

The same notation given anonymously as $\langle a_1 = b_1, a_2 = b_2, \ldots, a_n = b_n\rangle$ is meant to indicate the same element of a structured set without the Ψ field.

$$((a_1, a_2, \ldots, a_n), (b_1, b_2, \ldots, b_n)) \text{ where } b_i \in V_i$$

Operators - Definition 2 *The $\uparrow$ and $\Uparrow$ operators are unary operators and are defined on ordered sets. $\uparrow$ returns the first element in the list and $\Uparrow$ returns the list without the first element or $\emptyset$ otherwise.*

$$\uparrow \{s_1, s_2, \ldots, s_n\} = \begin{cases} s_1 & \text{if } n > 0 \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

$$\Uparrow \{s_1, s_2, \ldots, s_n\} = \begin{cases} (s_2, \ldots, s_n) & \text{if } n > 1 \\ \{\emptyset\} & \text{otherwise} \end{cases}$$

The $\downarrow$ operator is a binary operator which returns a new list with the left hand list at the front of the new list and the right hand list at the end of the list. The operator is used as follows.

$$\{a_1, a_2, \ldots, a_n\} \downarrow \{b_1, b_2, \ldots, b_m\} = \begin{cases} \{\emptyset\} & \text{if } n = 0 \wedge m = 0 \\ \{a_1, a_2, \ldots, a_n\} & \text{otherwise if } m = 0 \\ \{b_1, b_2, \ldots, b_m\} & \text{otherwise if } n = 0 \\ \{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m\} & \text{otherwise} \end{cases}$$

This notation makes it very easy to define the λ function. For example, let e be an event to be broadcast and with an address of $(\perp, \perp, k)$. e is written as:

$$e\langle \Delta = (\perp, \perp, k), \beta = 1\rangle$$

A λ function should now put e on the output port indicated by the first element in the ordered $e.\Delta$ set.

$$\lambda(s) = \langle \uparrow e.\Delta = e\langle \Delta = \Uparrow e.\Delta, \beta = e.\beta\rangle\rangle$$

B
The Statecharts-AToM^3 Environment

B.1 Introduction

This appendix explains the usage of the Statecharts to DEVS environment in AToM^3. It discusses how to build graphical Statechart models, how to transform a Statechart to a DEVS model and then how to execute the DEVS models in real-time.

B.2 Buttons Overview

When the environment starts you should see the screen in Figure B.1. The left-hand toolbar features buttons which are specific to Statecharts while the top toolbar provides more general functions. The following sub-sections explain the usage of each of the Statechart buttons. For more information on the top toolbar, please refer to the AToM^3 website at `http://atom3.cs.mcgill.ca`.

B.2.1 `New Composite`, `Basic`, `History`, `Orthogonal` and `Hyperedge` Buttons

These buttons do exactly what their names imply. For example, clicking on `New Composite` then clicking on the canvas will create a new composite state. Note that while building Statechart models you will almost never need to use the `New Hyperedge` button. This is present only for the purpose of building Statechart graph grammars.

B.2.2 `New visual_settings` Button

This button draws a large square on the canvas. When edited (ie. by clicking the `Edit entity` button then on the `visual_settings` object) the modeler will be presented with a series of visual options in a dialog box. This dialog can be seen in Figure B.2. Each of the options change visual settings based on their descriptive names.

B.2.3 `To DEVS` Button

The `To DEVS` button converts a Statechart model to an equivalent DEVS model. The output is written in `python` and is for usage with the `pythonDEVS-RT` package. The output file containing the `pythonDEVS` code is set using the global model attribute `devs_output_dir`. The DEVS model which is output is a subclass of `CoupledDEVS`. This represents the root of the Statechart. It is the responsibility of the modeler to combine this model with the `Executor` and any other desired

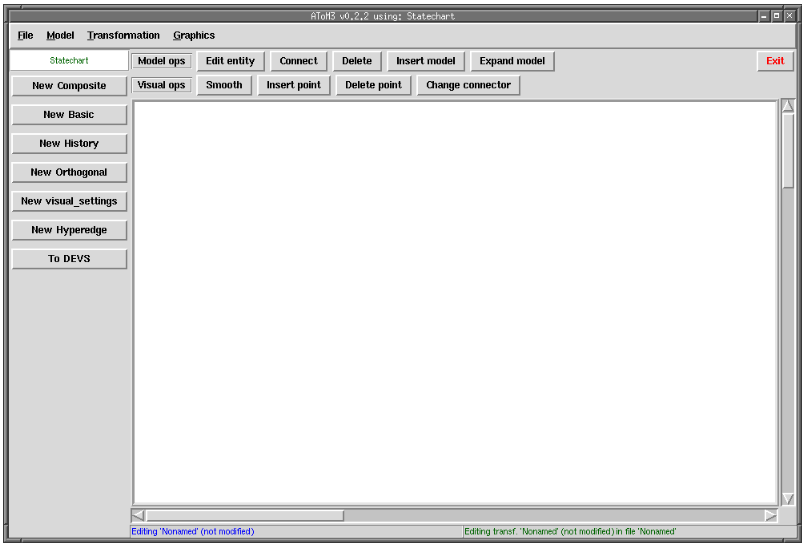

Figure B.1: The starting screen.

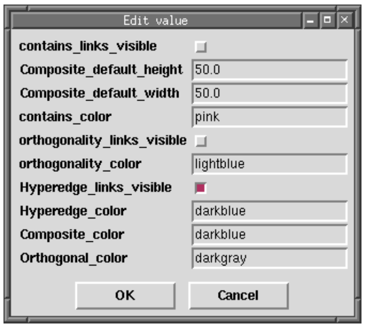

Figure B.2: The visual settings dialog.

Figure B.3: The connect dialog.

packages to construct a working application. This will be discussed in more detail.

B.3 Operational Overview

Now that you know how to put basic components on the canvas, a discussion on how to connect them in various ways, is given. The following sub-sections discuss simple operations and how to perform them.

B.3.1 Connecting States with a Hyperedge

In order to connect two states with a `Hyperedge`, first click the `connect` button located on the top toolbar. Next click the source state and then the destination state. If the source state is of type `Basic`, then the Hyperedge will be drawn automatically. If the source state is of type `Composite` The modeler will be presented with another dialog as seen in Figure B.3.

Choosing the `Hyperedge` option will draw the desired connection between the states. If you wish to create an *n*-ary Hyperedge, simply perform this operation again. However, first click the middle segment of the transition as the source, and the desired state as the destination.

B.3.2 Nesting States

Nesting a state within a `Composite` state is very similar to connecting them via a `Hyperedge`. However, when presented with the connection dialog, choose the `contains` option. This will nest the destination state within the source state. The container state will automatically resize to envelop everything within itself.

B.3.3 Destroying a `contains` Relation

If you wish to destroy the `contains` relation between two states, you should use the `visual_settings` component. Put a `visual_settings` on the canvas, and edit it. Set the `contains_links_visible` option and click `OK`. Now you will be able to see the contains links. Simply click the `Delete` button on the top toolbar then on the link you wish to delete.

B.3.4 Adding History

To add a history state, simply add the history state anywhere on the canvas, then nest it within the desired composite state. You can connect hyperedges to the history state, but outgoing history hyperedges are not supported in this version. Remember that the history state is not a state at all. Rather it represents a section of a transition.

B.3.5 Editing States

Editing states requires clicking the edit button, then on the component to be edited. When editing a state, the modeler will be presented with the dialog in Figure B.4. The options presented in this

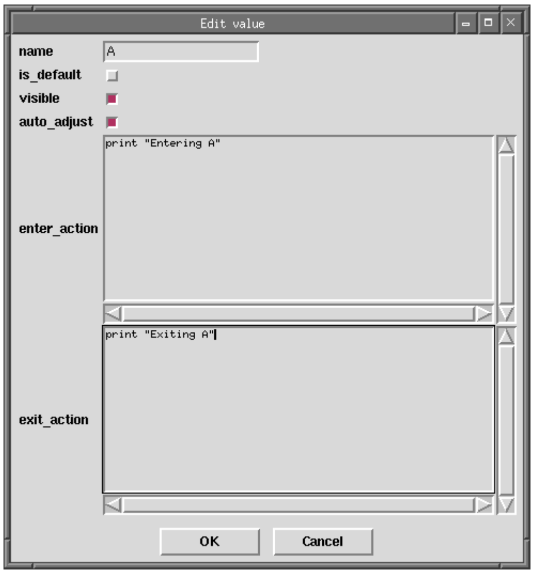

Figure B.4: The edit state dialog.

dialog are explained in table B.1.

Table B.1: Options for the state editing dialog.

Option	Description
`name`	The name of the state.
`is_default`	Indicates if the state is default.
`visible`	Will hide/reveal the interior of a `Composite` state.
`auto_adjust`	Should always be set.
`enter_action`	The action to be executed upon entering the state.
`exit_action`	The action to be executed upon exiting the state.

B.3.6 Creating n-ary Transitions

To create a transition with more than one source state and/or more than one destination state perform the following steps. To create another source state simply click on the source state, then on the transition. A link will be created. To add a destination state click on the transition, then on the destination state and a link will be created. Recall that these hyperedges move the system from a state, x, which is the product of x' basic states where $x' > 1$, to a new state y which is the product of y' basic states where $y' > 1$. This means that you must ensure that each individual source and destination state resides in a separate orthogonal component.

B.3.7 Editing Transitions

Editing transitions requires clicking the edit button, then on said transition. The dialog in Figure B.5 is presented when editing transitions. Each option is explained in Table B.5.

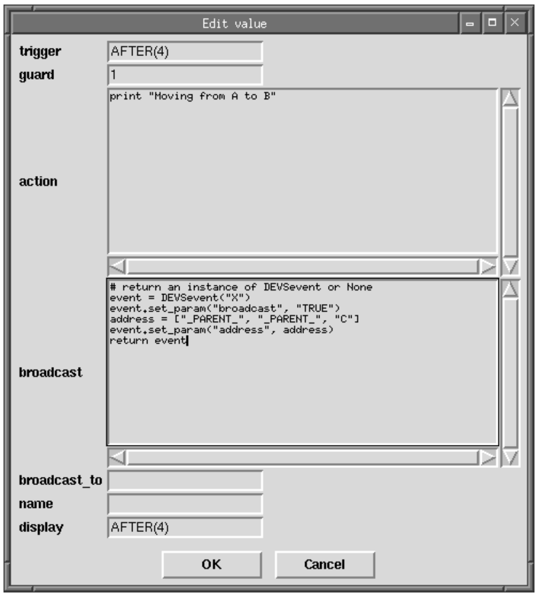

Figure B.5: The edit transition dialog.

Table B.2: Options for the transition editing dialog.

Option	Description
`trigger`	The transition trigger. See section B.3.8 for more information.
`guard`	Valid `python` boolean expression.
`action`	The action to be executed when the transition fires.
`broadcast`	Used to broadcast or narrowcast events. See section B.3.7 for more information.
`broadcast_to`	Deprecated.
`name`	Deprecated.
`display`	The text to be displayed for the transition.

B.3.8 Broadcasting

The broadcast field of the transition edit dialog is used to narrowcast and broadcast events. The modeler can write python code to construct an event in a variable, `x`. Then the last statement should be `return x` which will indicate that `x` is the instance to be broad or narrowcasted.

The `x` instance should somehow be an instance of `RT_DEVS.EVENT` (you can also use `RT_DEVS.DEVSevent` which is an alias for the same class). This object has special methods for setting and getting event fields. These methods are `get_param(name)` which returns the field value associated with `name`. The method `set_param(name, value)` sets the field `name` to `value`.

Any parameters may be added to an event, but there are some special parameters that the modeler should know. These special parameters are listed in table B.3.

Note that the modeler may use both the `address` parameter to narrowcast an event coupled with broadcasting. The effect is that once the event reaches its destination via the `address` parameter, it will then be broadcast only within the destination state.

For example, consider the statechart model in Figure B.6. The transition from state `PASSED_FF` to

Table B.3: Special Event Fields.

Field	Description
`broadcast`	`"TRUE"` or `"FALSE"`. Indicates if the event is to be broadcasted or not. If not specified, "FALSE" is implied.
`address`	List of port names. The list is used by DEVS atomic relays to route events to be narrowcasted. The special name `"_PARENT_"` is used to indicate the containing state. The name `"_ROOT_"` is a direct route to the `"_ROOT_"` outport of the root coupled DEVS.

`WAIT_SEEK_DONE` generates a `SEEK` event. This indicates synchronization between the `Audio` component and the `Display` component. Note that the only transition which responds to the `SEEK` event is that which emanates from the `SEEKING` state. Thus, broadcasting the event is not needed, since its exact route is known. Thus in order to route the event from its place of origin to the `SEEKING` state, the `address` parameter should be specified as `["_PARENT_", "_PARENT_", "_PARENT_", "_PARENT_", "Audio", "SEEKING"]`.

Notice that this forces the event to travel from `PASSED_FF` to `TIME_PASSED` to `TIME` to `Display` to the `ROOT` to `Audio` to `SEEKING`. Note that if you wish an event to be *heard* without broadcasting you should narrowcast it to the source state of the desired transition.

B.3.9 Triggers

Triggers can either be a string indicating the name of the trigger or an `after` expression. To specify an event name, simply enter it without quotes. A timed trigger has the following syntax: `AFTER(exp)` where `exp` is a valid python expression evaluating to a number. Be sure to enter `AFTER` in capitals.

B.4 Executing a Generated DEVS Model

A `pythonDEVS` model on its own will not do anything. It must be linked with an `Executor` in order to put it into motion. This is as simple as passing an instance of your model to the `RT_DEVS.Executor` constructor. Once you have constructed an `Executor` instance it can be executed by calling the `execute()` method. This will execute your Statechart model in real-time.

B.5 Building Larger Applications

In this section, the example in Figure B.7 will be used. This Statechart is meant to be the control layer between the MP3 decoder/audio hardware and the player's graphical interface (See Figure B.8. The MP3 decoder and audio hardware as well as the graphical interface are implemented as DEVS components themselves. This makes it very easy to combine generated Statechart models with these DEVS models. All the files used in the application are listed in table B.4.

Each of the main components or globally scoped variables may be needed in various places thus it is best to instantiate them in another file, `config.py`, so they may be included later using the `import` statement.

```
import DEVS_DEVSamp, DEVSampGUI, MP3AudioDriver
GUI = DEVSampGUI.DEVSampGUI()
```

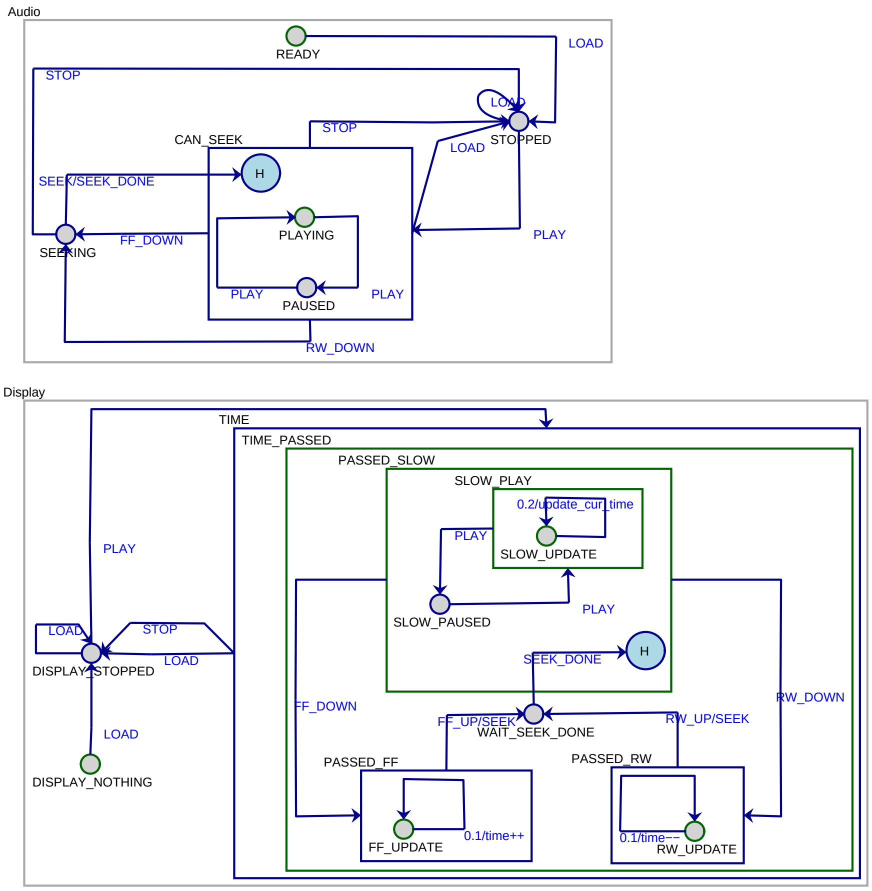

Figure B.6: An MP3 Player.

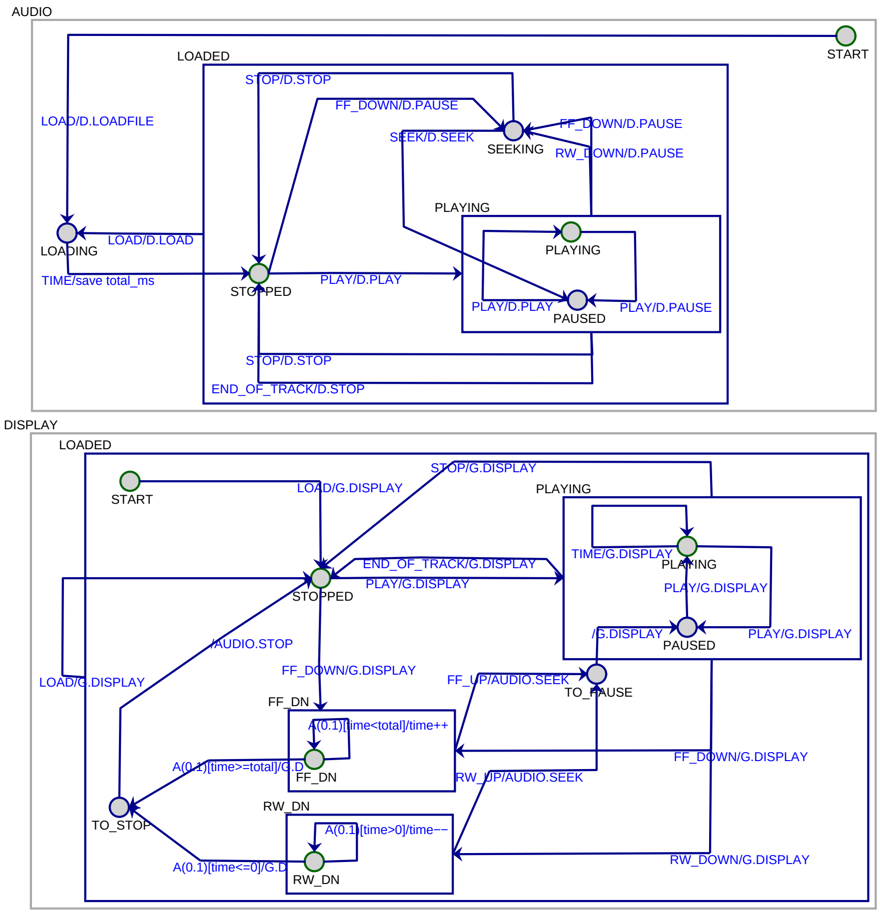

Figure B.7: The MP3 Player Control Layer Statechart.

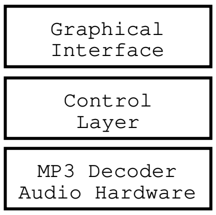

Figure B.8: Application layers.

Table B.4: Other files used in the DEVSamp application.

File	Description
main.py	The file which glues everything together.
config.py	Contains globals.
MP3AudioDriver.py	MP3 decoder and audio hardware interface.
DEVSampGUI.py	The application's graphical interface.
DEVS_DEVSamp.py	The Statechart pythonDEVS model.

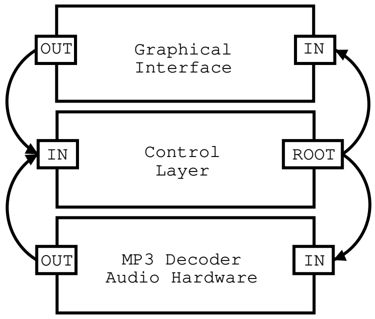

Figure B.9: Port connections among the high level components.

```
DEVS = DEVS_DEVSamp.ROOT()
DRIVER = MP3AudioDriver.Driver()
```

The three main components in the application are each DEVS objects. Thus, they can be coupled within a Main coupled DEVS. This will be written in the main.py file along with the code to execute() the Main model.

```
import config, RT_DEVS
class Main(RT_DEVS.CoupledDEVS):
  def __init__(self):
    RT_DEVS.CoupledDEVS.__init__(self)
    self.addSubModel(config.GUI)
    self.addSubModel(config.DEVS)
    self.addSubModel(config.DRIVER)
    self.connectPorts(config.GUI.ports['OUT'], config.DEVS.ports['_IN_'])
    self.connectPorts(config.DRIVER.ports['OUT'], config.DEVS.ports['_IN_'])
    self.connectPorts(config.DEVS.ports['_ROOT_'], config.GUI.ports['IN'])
    self.connectPorts(config.DEVS.ports['_ROOT_'], config.DRIVER.ports['IN'])
E = RT_DEVS.Executor(Main())
E.execute()
```

Each connection is now visible in the high level diagram in Figure B.9. Notice that the _ROOT_ port is connected to the IN ports of both the graphical and the driver components. It would probably be better plug the _ROOT_ port into an atomic DEVS which filters the events so they are not sent to the adjacent models when not needed. This is a non-crucial optimization. Optimizations will be discussed further in the following sections.

Also notice that it is easy to instantiate the three high-level components within the config.py file. This makes it easy to access them in various parts of the program. However, beware of breaking the compositional structure of the DEVS semantics. It is easy to import the driver instance at any point of the program and simply call some of its methods. However, if the DEVS model is meant to be executed on a network or in low-level electronic components, you may not have this luxury. You should always use events to communicate between components.

B.5.1 The `_IN_` Port

The most important port of the top-level DEVS generated from a Statechart is `_IN_`. This is only way to ensure events are heard within the walls of the Statechart model. Remember that events pushed through the `_IN_` port can use the `address` parameter to provide narrowcasting. Otherwise, you may wish to set the `broadcast` parameter to `"TRUE"` which will broadcast the event across the entire Statechart.

B.5.2 The `_PARENT_` Port

This port refers to the containing composite Statechart state or orthogonal component. The `"_PARENT_"` port will never be routed to a DEVS representing a basic state.

B.5.3 Substate Ports

There exist an output port for each substate of a particular composite Statechart state. The output port `x` will connect to the input port `_IN_` of the substate (or orthogonal region) named `x`.

B.5.4 The `_ROOT_` Port

An important port when linking top-level DEVS components is `_ROOT_`. You may want an output for a particular transition to be sent to one of the adjacent top-level components. Instead of fully specifying the path of the `address` parameter, you can simply set it to `_ROOT_`. This is a special port which pushes an event directly to the `_ROOT_` output port of the coupled DEVS representing the root of the Statechart. This is merely an optimization. That is, you can also specify the full address path, if desired.

B.5.5 Dynamically Changing an Event Path

This is made possible by intercepting a particular event, modifying its address parameter, then ensuring its output. However, this is the equivalent to responding to an event and outputting another event which happens to be the same as trigger event and has a different address.

B.5.6 The Top-level DEVS Component Interfaces

The modeler must have a full working knowledge of the DEVS components they are using in their application. In this case, the other top-level components are `MP3AudioDriver` and `DEVSampGUI`. A high-level description of their semantics is enough to allow the modeler to successfully interface them in an application.

The `MP3AudioDriver` Interface

The class to instantiate is `MP3AudioDriver.Driver`. This atomic model has one input port `IN` and one output port `OUT`. Table B.5 is a description of the events which may be passed to the driver to change its state.

Table B.6 is a description of events which may come out of the `OUT` port. A brief description is also given, so the modeler may interpret the event's meaning and rough timing.

Table B.5: Reactive Events for `MP3AudioDriver`.

Event Name	Parameters	Description
`LOADFILE`	`filename`:The name of the file to be loaded.	Loads an MP3 file.
`PLAY`		Starts playback from the current file pointer.
`PAUSE`		Stops playback.
`STOP`		Stops playback and sets the file pointer to the beginning of the file.
`SEEK`	`offset_ms`: The driver will set the file pointer such that the current time will be `offset_ms`.	Move the file pointer freely. If `offset_ms` is less than 0, the driver will move to a STOPPED state. If `offset_ms` is greater than the total track time, the driver will output an `END_OF_TRACK` event.
`CLOSE`		Closes the driver.

Figure B.10: The DEVSamp Graphical Interface.

The `DEVSampGUI` Interface

The class to be instantiated is `DEVSampGUI.DEVSampGUI` which is a sub-class of `RT_DEVS.AtomicDEVS`. The appearance of this component can be seen in Figure B.10. Notice that, each of the widgets used are labeled in red. Table B.7 describes the possible inputs you may give to this component. Table B.8 lists all the possible outputs from this component. Each output basically corresponds to a button click. Note that the parameters are not specified because each event has the same parameters. These parameters are `broadcast` which is set to `"TRUE"` and `address` which is set to `[]`.

B.6 Simple Examples

Some simple Statecharts will be examined as well as how to build them in the Statechart modeling environment. The first example is seen in Figure B.11.

Start up the AToM^3 that came with the Statechart package you downloaded. Statecharts will be the

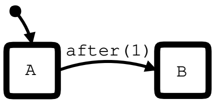

Figure B.11: A Simple Statechart.

Table B.6: Output Events for `MP3AudioDriver`.

Event Name	Parameters	Description
`PLAYING_PULSE_TIME`	`offset_ms`:The current time in ms. `broadcast`: `"TRUE"`. `address`: `[]`.	Once the driver is put into the playing state, these events are pulsed out approximately every 0.2 s in order to indicate the current time.
`LOADING_PULSE_TIME`	`total_ms`:The total track time in ms. `broadcast`: `"TRUE"`. `address`: `[]`.	This event is only output right after receiving a `LOADFILE` event. It indicates the total time of the track just loaded.
`END_OF_TRACK`		Indicates the end of the current track has been reached. Can be output while playing or seeking.

Table B.7: Inputs to `DEVSampGUI`.

Event Name	Parameters	Description
`DISPLAY`	`time`:A tuple of the form `(mins,secs)` [7]. `info`: A string [8]. `state`: A string which should indicate the state of the player [9]. `playpause`: A string indicating which to write on the PLAY/PAUSE button.	Sets the various display widgets on the graphical interface.

default meta-model loaded. Next put two Basic states on the canvas. Name the first *A*, the other *B* and make sure that *A* is the default state. Create a transition between the two of them and set the trigger as *X*. You can also set the display as *X*. Your statechart should now look like the one in figure B.12. Recall that there is an implicit composite state enveloping this model, named `ROOT`. Next specify some enter/exit actions for both *A* and *B* which simply print something. For example the edit dialog for *A* should look something like the one if Figure B.13. You can do the same for the transition from *A* to *B*. Its edit dialog should look something like the one in figure B.14.

Now, this Statechart model will be transformed to a DEVS model. First the output directory must be specified. Select ”Edit Model Attributes” from the ”Model” menu. Specify a full physical path filename, of a path relative to the current directory in the space for ”devs_output_file”. Finally, click the ”To DEVS” button on the left toolbar. You will see some output on `stdout` which should look something like the following.

```
START TRANSFORMATION
ABOUT TO BUILD STATECHART
```

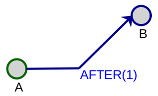

Figure B.12: The Statechart in AToM^3.

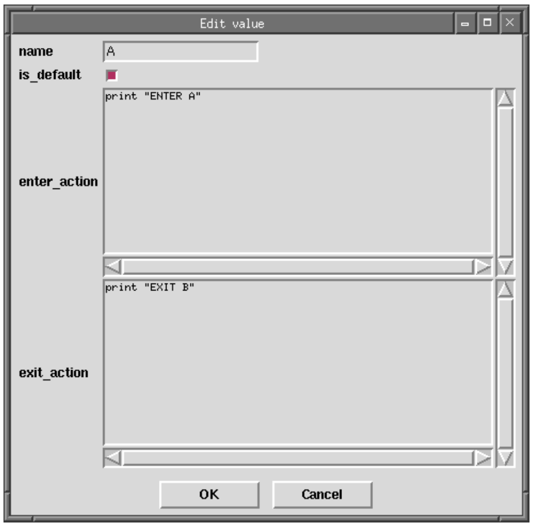

Figure B.13: A's edit dialog in AToM^3.

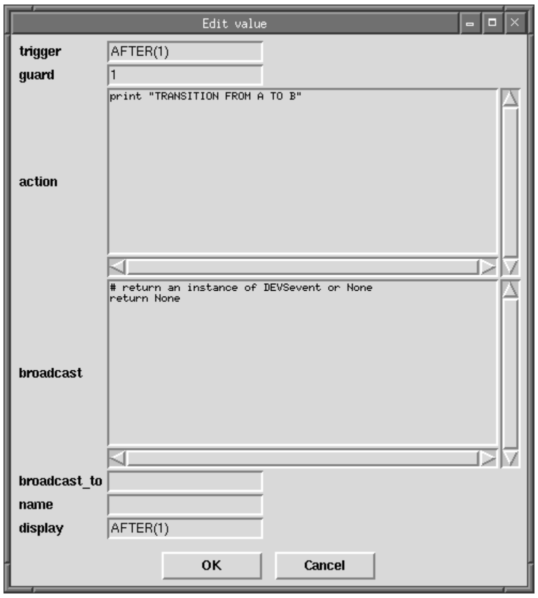

Figure B.14: Transition dialog in AToM^3.

Table B.8: `DEVSampGUI` Outputs.

Event Name	Description
`PLAY`	Button 1 click.
`STOP`	Button 4 click.
`LOAD`	Button 3 click.
`QUIT`	Button 6 click.
`RW_DOWN`	Button 2 down.
`RW_UP`	Button 2 up.
`FF_DOWN`	Button 5 down.
`FF_UP`	Button 5 up.

```
ABOUT TO VERIFY SYNTAX
ABOUT TO BUILD DEVS
ABOUT TO EMIT DEVS
END TRANSFORMATION
```

This means everything went according to plan. If you see any python errors, then something went wrong and you must retrace your steps.

You can now examine the output file, in this case `THEMODEL.py`, for the presence of the Statechart model as a DEVS model. In the same directory as the output file you should place the `RT_DEVS.py` package. Finally, a `main.py` file is constructed where everything is glued together. The `main.py` file should look something like the following.

```
from RT_DEVS import Executor
from THEMODEL import ROOT
Executor(ROOT()).execute()
```

Now execute the main file using the command `python main.py`. This will start by producing the following output.

```
ENTER A
```

Then after about a second, you will see the rest of the output.

```
TRANSITION FROM A TO B
EXIT A
ENTER B
```

Now suppose a small modification is made to the original Statechart design (Figure B.15). The transition from *A* to *B* is now triggered by the event *X*. The transition edit dialog should looking something like the one in figure B.17. Your Statechart should now look like the one in Figure B.17.

Again, click the ”To DEVS” button and the Statechart model has been transformed to a DEVS model. However, if an attempt is made to execute this model, state *A* will never be exited. This is due to the fact that an *X* event is never generated. In order to generate an *X* event, another AtomicDEVS model could be contructed which would output an event to the `ROOT` Statechart. This model can be placed into the `main.py` specified below.

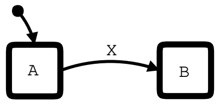

Figure B.15: The Simple Statechart slightly modified.

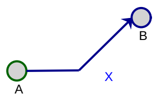

Figure B.16: The modified Statechart in AToM^3.

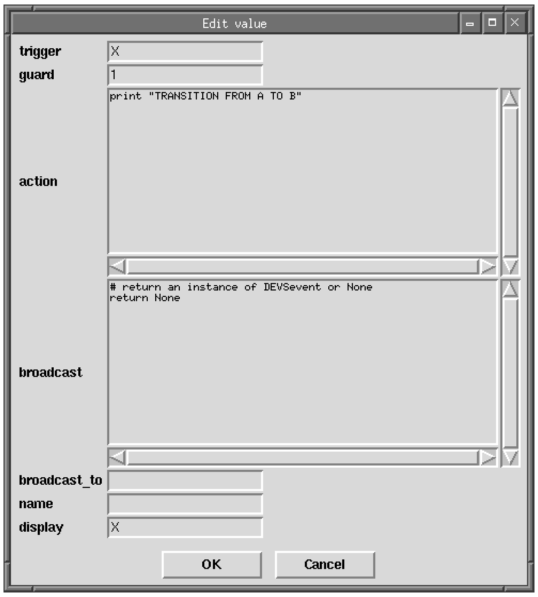

Figure B.17: Modified transition dialog in AToM^3.

Notice that after 5 seconds the event X is output with a zero length address parameter and a set broadcast parameter. This event is poked out the `OUT` port. Thus, there must be a coupled DEVS which connects this `OUT` port to the `_IN_` port of the `ROOT`. This coupled DEVS will be called `Main` and should also be placed in the `main.py` file specified below.

The final `main.py` file looks like the following.

```
from RT_DEVS import Executor,AtomicDEVS,CoupledDEVS,INFINITY,EVENT
from THEMODEL import ROOT

class Generator(AtomicDEVS):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.states = {"A":1,"B":2}
    self.ports = {"OUT":self.addOutPort("OUT")}
    self.state = self.states["A"]
  def timeAdvance(self):
    if self.state == self.states["A"]: return 5
    else: return INFINITY
  def outputFnc(self):
    if self.state == self.states["A"]:
      self.poke(self.ports["OUT"], EVENT("X", [("address", []), ("broadcast", "TRUE")]))
  def intTransition(self):
    if self.state == self.states["A"]: return self.states["B"]
    else: return self.state

class Main(CoupledDEVS):
  def __init__(self):
    CoupledDEVS.__init__(self)
    G = self.addSubModel(Generator())
    R = self.addSubModel(ROOT())
    self.connectPorts(G.ports["OUT"], R.ports["_IN_"])

Executor(Main()).execute()
```

Notice that instead of executing the `ROOT` model, the `Main` model is executed.

Bibliography

[1] Arthur Allen and Dennis de Champeaux. Extending the statechart formalism: Event scheduling and disposition. *Conference on Object Oriented Programming Systems and Languages and Applications*, pages 1–16, 1995.

[2] Scott Ambler. How to draw uml activity diagrams. 2000.

[3] Uri M. Ascher and Linda R. Petzold. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM, Philadelphia, PA, 1998.

[4] AToM^3. `http://atom3.cs.mcgill.ca`.

[5] Osman Balci. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In Michael A. Abrams, Peter L. Haigh, and John C. Comfort, editors, *Proceedings of the 1988 Winter Simulation Conference*, pages 287–295, 1988.

[6] R. Bardohl, C. Ermel, and I. Weinhold. Agg and genged: Graph transformation-based specification and analysis techniques for visual languages. volume 72, 2002.

[7] J.R. Beauvais. Modeling statecharts and activitycharts as signal equations. *ACM Transactions on Software Engineering and Methodology*, 10(4):397–451, October 2001.

[8] Claude Berge. *Topological Spaces: including a treatment of multi-valued functions, vector spaces, and convexity*. Dover Publications, Mineola, NY, 1997.

[9] Jean-Sbastien Bolduc, Gordon Broderick, and Denis Thrien. From stability to tracking: Robustness of cellular automata based controllers. volume 1, pages 619–624, Beijing, China, octobre 1997.

[10] Jean-Sbastien Bolduc and Hans L. Vangheluwe. Mapping ODEs to DEVS: Adaptive quantization. 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'03).

[11] Jean-Sbastien Bolduc and Hans L. Vangheluwe. Expressing ODE models as DEVS: Quantization approaches. *Proceedings of the AIS'2002 Conference*, pages 163–169, april 2002.

[12] Jean-Sébastien Bolduc and Hans Vangheluwe. A modeling and simulation package for classic hierarchical DEVS. MSDL, School of Computer Science, McGill University, 2002.

[13] Carlos R. Borges. *Elementary Topology and Applications*. World Scientific Publishing, River Edge, NJ, 2000.

[14] Francis Ceschino. Modification de la longueur du pas dans l'intgration numrique par les mthodes pas lis. *Chiffres: Revue de L'Association franaise de calcul*, 2:101–106, 1961.

[15] William Chan, Richard J. Anderson, Paul Beame, David H. Jones, David Notkin, and William E. Warner. Optimizing symbolic model checking for statecharts. *IEEE Transactions on Software Engineering*, 27(2):170–190, February 2001.

[16] A.C.-H. Chow. Parallel DEVS: A parallel, hierarchical, modular modeling formalism and its distributed simulator. *Transactions of the Society for Computer Simulation International*, 13(2):55–68, June 1996.

[17] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[18] J. de Lara and H. Vangheluwe. Atom^3: A tool for multi-formalism modelling and meta-modelling. *Lecture Notes in Computer Science (FASE)*, 2306:174–188, 2002.

[19] Maria Christina Ferreira de Oliveira. A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems*, 19(1):28–52, January 2001.

[20] Sung Deok Cha and Hyoung Seok Hong. Specification and analysis of real-time systems in statecharts. In IEEE, editor, *Proceedings of the 2nd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '96)*, 1996.

[21] Hilding Elmqvist, Martin Otter, and Franois E. Cellier. Inline integration: A new mixed symbolic/numeric approach for solving differential-algebraic equation systems. pages xxiii–xxxiv, Prague, Czech Republic, 1995.

[22] Christian Engstler and Christian Lubich. Multirate extrapolation methods for differential equations with different time scales. *Computing*, 58(2):173–186, 1997.

[23] C. William Gear. *Numerical Initial Value Problems in Ordinary Differential Equations*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[24] Jeffrey B. Green. Correspondences between statecharts, event structures and concurrent regular expressions. In *Proceedings of the 36th Annual Conference on Southeast Regional Conference*, pages 178–184. ACM Press, 1998.

[25] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, Berlin, second revised edition, 2000.

[26] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer-Verlag, Berlin, second edition, 1996.

[27] David Harel. Statecharts: A visual formalism for complex systems. 1987.

[28] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[29] David Harel and Eran Gery. Executable object modeling with statecharts. *IEEE Proceedings of the ICSE*, pages 246–257, 1996.

[30] David Harel and Eran Gery. Executable object modeling with statecharts. In *International Conference on Software Engineering*, pages 246–257. IEEE Computer Society Press, 1996.

[31] David Harel and Chaim-arie Kahana. On statecharts with overlapping. *ACM Transactions on Software Engineering and Methodology*, 1(4):399–421, October 1992.

[32] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[33] David Harel, A. Pnueli, J.P. Schmidt, and R. Sherman. On the formal semantics of statecharts. *Symposium on Logic in Computer Science*, pages 54–64, June 1987.

[34] David et al. Harel. Statemate: A working environment for the development of complex reactive systems. In *International Conference on Software Engineering*, pages 396–406. IEEE Computer Society Press, 1988.

[35] Ayaz Isazadeh, Lamb, and Glenn H. MacEwen. Viewcharts: A behavioral specification language for complex systems. In *Proceedings of the 4th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS '96)*. IEEE, 1996.

[36] Gray J., Bapty T., and Neema S. Aspectifying constraints in model-integrated computing. *OOPSLA*, 2000.

[37] H.M. Jarvinen. Object-oriented specification of reactive systems. In *International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[38] Bolduc Jean-Sebastien and Hans Vangheluwe. A modeling simulation package for classical hierarchical DEVS. July 2002.

[39] Nima Kaveh and Wolfgang Emmerich. Deadlock detection in distribution object systems. In *Foundations of Software Engineering*, pages 44–51. ACM, 2001.

[40] Hassan K. Khalil. *Nonlinear Systems*. Prentice Hall, Upper Saddle River, NJ, second edition, 1996.

[41] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, 13(3):135–154, September 1996.

[42] Ernesto Kofman and Sergio Junco. Quantized-state systems: A DEVS approach for continuous system simulation. *Transactions of SCS*, 18(3):123–132, 2001.

[43] Harbir Lamba and Andrew M. Stuart. Convergence proofs for numerical IVP software. Technical Report SCCM-98-05, Stanford University, CA, 1998.

[44] K.R.P.H. Leung. Extending statecharts with ad lib and multi-thread features. In *Proceedings of the Seventh Asia-Pacific Software Engineering Conerfence*. IEEE, 2000.

[45] Gerald Luttgen and Michael Mendler. The intuitionism behind statecharts steps. *ACM Transactions on Computational Logic*, 3(1):1–41, January 2002.

[46] Gerald Luttgen and Michael von der Beeck. A compositional approach to statecharts. *Foundations on Software Engineering*, pages 120– 129, 2000.

[47] M. Minas. Specifying graph-like diagrams with diagen. *Science of Computer Programming*, 44:157–180, 2002.

[48] P. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling. *ACM Transactions on Modeling and Computer Simulation*, 12:1–7, 2002.

[49] Alexandre Muzy and Gabriel A. Wainer. Cell-DEVS quantization techniques in a fire spreading application. pages 542–548, San Diego, CA, USA, 2002.

[50] Richard E. Nance. The time and state relationships in simulation modeling. *Communications of the ACM*, 24(4):173–179, April 1981.

[51] OMG. Object Management Group. `http://www.omg.org`.

[52] Constantinos C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, march 1988.

[53] Mor Peleg and Dov Dori. The model multiplicity problem: Experimenting with real-time specification methods. *IEEE Transactions on Software Engineering*, 26(8):742–759, August 2000.

[54] Ernesto Posse and Jean-Sbastien Bolduc. Generation of DEVS modelling & simulation environments. 2003 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'03), 2001.

[55] Python. `http://www.python.org`.

[56] Ashvin Radiya and Robert G. Sargent. A logic-based foundation of discrete event modeling and simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(1):3–51, 1994.

[57] G. Rozenberg. Handbook of graph grammars and computing by graph transformation. 1, 1999.

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

[59] Sanjit Seshia and R. K. Shyamasundar. A translation of statecharts to esterel. *World Congress on Formal Methods*, pages 983–1007, 1999.

[60] Arcot Sowmya and S. Ramesh. Extending statecharts with temporal logic. *IEEE Transactions on Software Engineering*, 24(3):216–231, March 1998.

[61] Steven H. Strogatz. *Nonlinear Dynamics and Chaos: with applications to physics, biology, chemistry, and engineering*. Perseus Books, Reading, MA, 1994.

[62] Andrew M. Stuart and Antony R. Humphries. *Dynamical Systems and Numerical Analysis*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, MA, 1998.

[63] Olaf Stursberg, Stefan Kowalewski, and Sebastian Engell. On the generation of timed discrete approximations for continuous systems. *Mathematical and Computer Modelling of Dynamical Systems*, 6(1):51–70, 2000.

[64] H. Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. *IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134, 2000.

[65] H. Vangheluwe, J. de Lara, and P. Mosterman. An introduction to multi-paradigm modelling and simulation. *In Proceedings of AI, Simulation and Planning - AIS 2002*, pages 9–20, 2002.

[66] Hans Vangheluwe. The discrete event system specification (DEVS) formalism. *CS522*, September 2001.

[67] Daniel Varro. A formal semantics of uml statecharts by model transition systems. *Springer-Verlag: Lecture Notes in Computer Science*, 2505:378–392, January 2002.

[68] M. Von Der Beek. A comparison of statechart variants. *Springer-Verlag: Lecture Notes in Computer Science*, 863:128–148, September 1994.

[69] Gabriel A. Wainer and Bernard P. Zeigler. Experimental results of timed cell-DEVS quantization. Tucson, AZ, USA, 2000.

[70] Bernard Zeigler. *Theory of Modeling and Simulation*. Academic Press, 2000.

[71] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, London, 1984.

[72] Bernard P. Zeigler. *Theory of Modelling and Simulation*. Robert E. Krieger, Malabar, Florida, 1984.

[73] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, San Diego, CA, second edition, 2000.