A Modelica Statecharts Compiler

Reehan Shaikh

School of Computer Science, McGill University
Montréal, Québec, Canada
reehan.shaikh@cs.mcgill.ca
`http://moncs.cs.mcgill.ca/people/reehan`

Abstract. We present a method in which statecharts, which are commonly described visually, can be textually described using Modelica syntax. We also provide a compiler used to compile these Modelica statecharts to DES specification files. These DES specification files are then used by the Statechart Compiler (SCC) to produce executable code in some target language such as Java, C++, etc...; Python will be our language of choice.

Key words: Modelica, Statechart, Compiler, AToM^3

1 Introduction

Statecharts have become a widely used formalism for modelling complex systems. They are intuitively easy to learn, which is good for those who are trying to use such a formalism. They are extremely powerful and expressive. Moreover, the widespread work done in the field of statecharts has provided researchers with a plethora of information to start and continue with. We extend that research by first looking into Modelica specifications of statecharts.

Modelica[1] is a modelling language which has also been diligently looked at and has grown to become the de facto for model specifications. It is high-level, multi-domain and object-oriented, used mostly for complex, heterogeneous physical system descriptions. The abundance of recent work done with Modelica also suggests that it too will soon become a widely used method of describing models.

We wondered, with two such extensively used methods of modelling, why there wasn't a link between them and as such, have tried to narrow that gap. Thus, secondly, we look at and provide a compiler that compiles statecharts described textually in Modelica to DES[2] specification files. These specification files are also text-based and human-readable. The DES specification is then given to the Statechart Compiler (SCC)[2] which compiles the specification into an executable in some target language.

This paper is divided as follows. Section 2 discusses related work done by Song[3] using DEVS and Modelica. Section 3 explores our work with statecharts and Modelica. We finally conclude and highlight some future work in Section 4.

[1] www.modelica.org

2 Related Work - Modelica DEVS compiler

2.1 (py)DEVS

The *D*iscrete *EV*ent system *S*pecification was first established in 1976 by Bernard P. Zeigler[1]. This formalism gave a detailed structure to discrete-event modelling. PyDEVS is an implementation of this formalism using *Python*[2], an extremely high-level, interpreted and object-oriented language. The PyDEVS package contains two files; **DEVS.py** gives the class architecture for hierarchical model definitions while **Simulator.py** implements the simulation engine. We will go into further detail about the modelling architecture, so that we may relate it to Modelica, but no description of the simulation engine is given as it doesn't fall into the scope of this paper. For those interested, please refer to [1].

Atomic DEVS An *atomic* DEVS model is used for describing a simple system and is primarily referred to by its *state*. Each atomic model starts in an initial state and has an associated set of states. The *time-advance* function is used to calculate when the next internal transition is scheduled for. The *internal* transition function allows an atomic model to change its state. Prior to the internal transition, the *output* function permits an atomic model to send messages to other models by the use of the *poke* method. These messages are sent over channels connected by *ports*. Ports are either input or output and there is a clear distinction between them (i.e. the same port cannot be used for input and output).

Messages are received via the *peek* method, exclusively used in the *external* transition function. If a message is received, an external transition is triggered. This also allows the atomic model to change state but based on external stimuli only. When a model is interrupted by a message, the elapsed time since the last transition is stored. Access to this value is also exclusive to the external transition function. Once there is an external transition, the time-advance function is called to schedule the next internal transition.

The initialization of an atomic model allows the modeller to set the initial state, set the starting elapsed time (so that the model can begin its time stamping of events at some time other than the default 0.00) and define the model's set of ports. The default behaviour of the atomic model's time-advance function is to return ∞(infinity). The internal and external transition functions both return the current state if not overridden. The default output function does absolutely nothing. The constructor will set the elapsed time to the default 0.00 and the model's state to None - Python's null object. No ports are defined.

In summary, an atomic model will first call its time-advance function to schedule the next internal transition at time x. It will then set itself to its initial state. If no external stimuli interrupts the model before time x is reached, then the model outputs its messages if any, an internal transition takes place and the time-advance function schedules the next internal transition at time y. If the

[2] www.python.org

model is interrupted before time x, no messages are sent, an external transition is triggered and the time-advance function is called to schedule the next internal transition at time z.

Coupled DEVS A *coupled* DEVS model is used to describe complex systems. It consists of, possibly several, submodels that are either atomic or coupled. A coupled model has only one function of importance to the modeller, the *select* function. This method is used as a tie-breaker if and when two submodels of the coupled model have events scheduled at exactly the same time. It allows the modeller to choose a specific model whose events should be carried out first.

A coupled model may also have ports. The only difference is that the input (output) port of the coupled model must be connected to the input (output) port of one of the coupled model's submodels. Essentially, only atomic models can send messages to each other. If some atomic model A1 (that is a submodel of a coupled model C1) must send a message to atomic model A2 (that is a submodel of C2), then A1 must send the message from its output port to the output port of C1, C1 will send the message from its output port to the input port of C2 and finally C2 will deliver the message from its input port to the input port of A2. Hence, three separate channels must be defined for this communication to take place. The channel that connects C1 and C2 must be defined in a coupled model C3 which has as submodels C1, C2 and possibly other models. The channel that connects C1 and A1 must be defined in C1. Finally, C2 defines the channel that connects it to A1.

The initialization of a coupled model first instantiates all its submodels and then defines the channels between submodels as well as between itself and any of its submodels. The default select function will return the first element in the list of colliding submodels. When a model is instantiated, it is given a unique *myID* attribute and this list is lexicographically sorted based on this attribute. The default constructor for a coupled model will do absolutely nothing.

2.2 Modelica

Modelica is a well-known multi-domain, object-oriented modelling language used primarily to describe heterogeneous complex physical systems. It is very powerful and, as such, is used to describe pretty much any type of model, physical or not, complex or not, etc.... It is declarative, as opposed to most programming languages which are imperative. Hence, it allows the modeller to specify **what** the solution is whereas in most programming languages, **how** the solution would be executed is specified. This is a key difference in understanding that Modelica is at an abstraction level higher than programming languages and users don't need to worry about implementation details. This gives the opportunity for modellers, who are usually domain experts, not to be technically inclined in the software domain. This allows these experts to concentrate on their expertise.

Basic primitive types *Real*, *Integer*, *Boolean* and *String* are available in Modelica. An *enumeration* type is given to represent list/array behaviour. Thus, Modelica comes in the flavour of typical programming languages. There are *classes* which encapsulate related tasks. There is also a notion of class inheritance and composition, consequently allowing hierarchical models to be built. *Packages* give rise to class organization. Conceived on the idea of reuse, Modelica comes with a **partial model** clause that lets modellers build incomplete models that may be reused in complete models. For example, suppose a model of a motor is built. This (partial) model may be used in a (complete) model of a vehicle, a motorbike, a motorboat, etc....

clause	**specialization**
model	like a class, not used in connections
connector	allows connections to be connected, no equations
record	groups data, no equations allowed, not used in connections
block	not used in connections, causal behaviour with input/output
type	alias used to extend an existing class

Table 1. Specialized Modelica classes

Certain types of *specialized* classes exist that give the modeller a bit more restrictive use of classes. Table 1 shows some of these classes and their specializations (restrictions). Modelica follows the use-before-declaration approach, hence variables can be used before declaring them. Nonetheless, a Modelica compiler should verify that all variables are declared. There are functions, if-then-else blocks and for/while loops, much like regular programming languages. One of the most important features for physical modelling is equations, specifically non-causal ones where the relationship between variables is described. For example, a + b = c is the same as c = a + b. These equations can be manipulated as sum-to-zero relations or as equivalent relations at connections and in some sense, depict the behaviour of the given class. For a detailed example of a real-world circuit, please refer to [6, pg. 7].

2.3 μModelica

Song[3] discusses how to model DEVS specifications using Modelica. Just as in *pyDEVS*[1], a library of base classes is provided for users to inherit from. These classes give the basic functionality required for building a DEVS model. In Modelica, the *extends* clause allows a class to inherit from a superclass. To build a DEVS model, one must declare all the Atomic- and Coupled-DEVS components as classes that inherit from superclasses *AtomicDevs* and *CoupledDevs* respectively. Each subclass of AtomicDevs can then define its *internal* and *external* transition functions as well as the *output* and *timeAdvance* functions. These functions are exactly like standard Modelica functions with no special meaning. The same goes for a CoupledDevs' *select* function.

The keywords *input* and *output*, used in Modelica to respectively depict the input and output of a function, have a second special meaning within DEVS models. They also allow the compiler to respectively distinguish between input and output ports. Recall that a pyDEVS port can either be an input or output port, but not both. So, when declaring a DevsPort, one must also declare what kind of DevsPort it is by combining the declaration with one of the keywords. The standard **connect** clause is used to connect two ports together. See Appendix A for an actual DEVS Modelica model of a generator which produces, at random intervals, a job to be processed by some processor.

The original μModelica compiler developed by Xu[5] focused primarily on the algebraic part of Modelica[6]. An abstract syntax tree (AST) of the input is built. Import statements are resolved. There is some type-checking done and scope analyses/symbol tables are also available. The compiler expands class inheritance and then flattens the Modelica code. Finally, the differential algebraic equations are analyzed and Octave code is produced. The compiler is very extensible. The visitor pattern is implemented to traverse the AST. Song[3] used this feature to implement a custom visitor that handled DEVS specifications. Upon reaching a DEVS node in the AST, the appropriate pyDEVS code was output to file. This also allowed very easy implementation of a new Statecharts visitor, described in the following section. For those interested in exactly how the DEVS compiler works, please refer to chapter 4 in [3].

3 Modelica Statecharts compiler

3.1 Statecharts

First we discuss the statecharts formalism. The building blocks of a statechart are as follows (in no particular order): basic states, composite components, orthogonal components, history states and transitions from components and/or states to components and/or states (including self-loops). There is also a notion of containment where a child component and/or state is contained within some other parent component. At the top of the parent-child hierarchy, there exists a root node so that one knows where the hierarchy begins.

Composite components are like Boolean OR operators where the system must reside in only one child of the composite (an OR over the set of children). Orthogonal components are composed of orthogonal sections. Each orthogonal section behaves like a composite component such that the system must be in exactly one child within that section. As a whole, an orthogonal component is like the Boolean AND operator where the system must reside in exactly one child within each of the orthogonal component's orthogonal sections (an AND over the sections). This effectively means the total state of an orthogonal component must be an element of the cross-product of all the states from each orthogonal section.

Each composite component, including the root if it is a composite, must have a default child. This allows for a modeller to enter a component without having to specify exactly which child needs to be entered and hence, the default child is entered. Orthogonal components cannot have a default orthogonal section because, essentially, each orthogonal section is active at the same time and thus, all are default simultaneously. Nonetheless, each orthogonal section itself must have a default child. Moreover, at the start of a simulation, the system must be able to reason about its initial state configuration and having a default child also permits that.

A transition may have a trigger (event) that enables its firing, a guard (Boolean condition) that restricts its firing and some action to be carried out when the transition is taken. Each of these are optional and any combination of them is allowed. Transitions, when drawn visually, start from a source node and end at a destination node. These nodes are mandatory when defining a transition. If no trigger is given, when the system enters the source node of that transition, the transition is unconditionally taken. If no guard is present, it defaults to *True*. If the source node is a component, then the transition is taken irregardless of which child of the source component the system is currently in. If the destination node is a component, then the system enters that component's default state when the transition is fired.

Basic states and composite components may carry enter and exit actions. History states can either be regular or deep and reside within some component. If a regular history state is entered, the system goes to the component's default child. If a deep history state is entered, the system proceeds to the component's most recent active child before last leaving the component. There are also ports and servers but these aren't taken care of by the compiler at this moment.

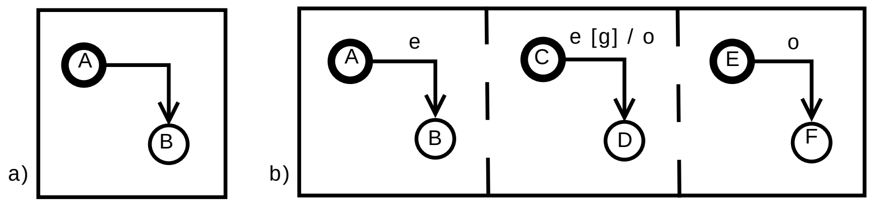

Fig. 1. a) Composite and b) Orthogonal components

Please refer to Figure 1 for a simple example of a) a composite component and b) an orthogonal component. Note that a circle is just a basic state and a heavy circle depicts a default child. The dashed line(s) within an orthogonal component separate each orthogonal section. If within the composite component, the system can only be in state (A or B) at any given time, but not both. If residing within the orthogonal component, the system must be in state [(A or B) and (C or D) and (E or F)].

There is only one unconditional transition (in the composite component). If the system enters state A of the composite, it will perform any action associated with state A and then go on to state B unconditionally. The other three transitions (in the orthogonal component) all involve an event. One transition is guarded by a condition g and has action code associated with it to output the event *o*. Suppose the orthogonal component is in its default states and we give it the event *e*. In the left most orthogonal section, the state will change from A to B. If the guard *g* is False, there will be no changes in the middle and right orthogonal sections. If the guard is True, the middle section will fire its transition, produce the event *o* which will then be given as input to the orthogonal component and finally change its state from C to D. Then the right most orthogonal section will fire its transition based on the produced event *o* and change its state from E to F.

3.2 μModelica

Building on the concept of using base classes, the same approach will be used to model statecharts in Modelica. A library of base classes for basic and history states, composite and orthogonal components, as well as for transitions (also known as hyperedges) is available for modellers to inherit from. This inheritance approach also doubles as a tagging technique so that the compiler can keep track of user-defined classes and their kind. Each base class comprises of input variables that hold the key information for that class. For example, the **Hyperedge** class contains input variables BaseStatechart *src* for the source node, BaseStatechart *des* for the destination node, String *trigger* for the trigger event, Boolean *guard* for the guard condition and String *action* for the action code to be executed. Please refer to Appendix B for the full library code.

One might think that Modelica's *connect* function would be useful for depicting transitions. A problem arises when transitions carry additional specifications, such as conditions, event triggers and/or actions, that have no place to reside within the connect function. One might argue that a wrapper class to encapsulate transitions and then use the *connect* clause is a good idea, but that would require additional bookkeeping of transitions and goes on to be a mess in the compiler's design.

Developing a Modelica statechart Developing a statechart in Modelica should and will be as straightforward as drawing it in some modelling tool. The following describes the generic process for developing a Modelica statechart but refers the reader to our example statechart (see Figure 2, developed using AToM^3[8]) so that a concrete sense of the development process can be realized.

Basic states are at the lowest level of the hierarchy, so they do not need to be modelled. Any basic state will be an instantiation of the class **Basic**, provided by the Modelica statecharts library. Basic states, when instantiated, can be given modifiers (in Modelica, modifiers are known as parameters to a class constructor that modify the default class behaviour). These are: Boolean *default*, String *enter* and String *exit*. The same goes for history states and transitions, classes **History** and **Hyperedge** are respectively given and need only to be instantiated for use. History classes have the modifiers Boolean *default* and Boolean *deep*. A **Hyperedge**'s modifiers are described above. The Modelica statecharts library also supplies base classes for **Composite** and **Orthogonal** components, but these cannot be at the lowest level of the hierarchy and thus need to be modelled. Hence, a user will extend these classes to model her own composite and orthogonal components as user-defined classes. Composite components contain the same modifiers as Basic states and orthogonal components have no modifiers.

All of the above classes, except for Hyperedge, extend the generic base Modelica class **BaseStatechart**. The use of a base Modelica class was to group together relevant classes. For example, a hyperedge's start and destination nodes may either be a basic state, history state, composite component or an orthogonal component. To easily allow for this, we say that the start and destination nodes must be of type BaseStatechart. Also note that some notion of the root node of the statechart is needed. We chose to allow the root user-defined class to extend BaseStatechart. When a class that extends BaseStatechart is encountered, it is stored as the root of the model. This would be regarded as the canvas in some visual modelling tool and as such, there can only be one root.

In our example statechart, Figure 2, the root component contains two basic states Setup and Stopped, a composite component NORMAL_MODE and two hyperedges. NORMAL_MODE in turn only contains two orthogonal components, MODE and CHRONO. MODE contains only basic states and hyperedges, while CHRONO contains basic states, hyperedges and a composite component RUNNING. Within RUNNING, there are only basic states and hyperedges. Now we attempt to textually describe this model in Modelica.

Our use-before-declaration approach allows us to model the statechart's components in any order. Thus, a modeller can pick any component she wishes, model it and move onto the next. For each component, we first figure out if it is a composite, orthogonal or root and extend the proper base class. Now we declare, in any order, the basic states, hyperedges and other subcomponents of the component with the appropriate modifiers to the class constructors. Hence, at this point, we must know the class type of the other subcomponents that we will model so that we can use those specific types in our declarations. This also allows for component reuse. Suppose we model a component C, then anywhere in our model we can instantiate C, as many times as one wishes.

A simple yet concrete example While describing the textual modelling process, please refer to Appendix C for a description of the partial[3] statechart in Figure 2. We first choose to model the MODE orthogonal component. We describe the class that models the component and later instantiate it, so we give the class the name Mode in camel-case. When it is instantiated, it will be given the identifier MODE (the name and identifier can be anything, but in order to coincide with the naming in the visual statechart of Figure 2, we stick to this method). Since Mode is an orthogonal, we first extend the Orthogonal base class.

[3] For the purpose of a simple example, this is a stripped down version of the entire DigitalWatch, only depicting the running time and chrono mode. The entire DigitalWatch is included with the distribution of the μModelica package, under the *example* folder, available at `http://moncs.cs.mcgill.ca/people/reehan/15_projects.dtml`

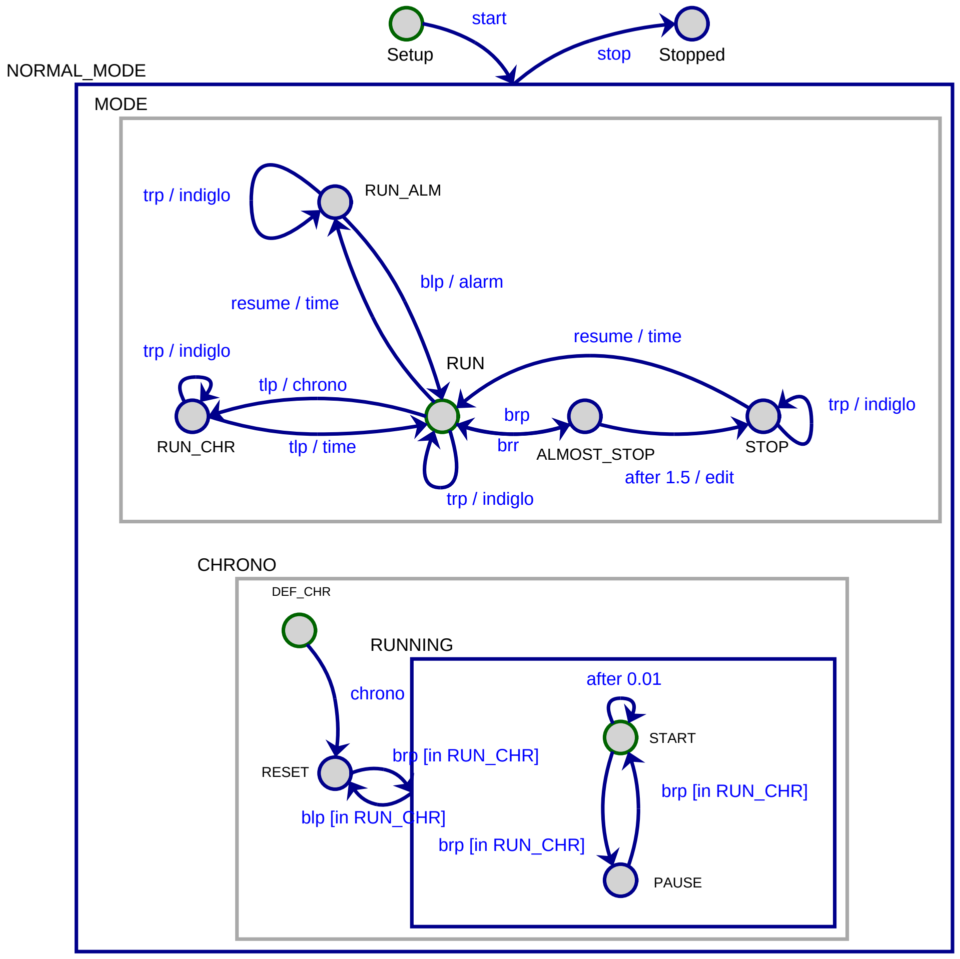

Fig. 2. A simple example statechart

Within Mode, there are 5 basic states which are instances of the Basic class and have the same names as in Figure 2. The basic state RUN is the default state of this component, so we pass the modifier *default*=true to the constructor. There are also 12 hyperedges. Since hyperedges don't have an explicit name in our visual tool, we choose the general naming convention of *h* followed by a number. Once again, you can choose to name them as you wish. For each hyperedge, we pass the appropriate modifiers to the constructor as well. The same thing is done for the RUNNING component by defining a class Running.

Now we wish to model the CHRONO component. Exactly the same steps as above are done except that CHRONO uses the Running component. So, along with instantiating the Basic states and Hyperedges, we also instantiate the Running component with the identifier RUNNING (to follow the visual statechart naming). The same is done for modelling the NORMAL_MODE component. Finally, we model the root of the statechart. We give this class a name, in our example we call it Statechart, extend BaseStatechart and follow the same steps of instantiating its children. Note that we model containment of children by some component via explicitly instantiating those children within the given class of that component. Also note that if a transition is required from a child A of a component C to a child B of C's subcomponent D, it is possible to accomplish this by separating each name in the hierarchy with a "**.**". Thus, we would have a transition within C from A to D**.**B. An example of this situation is included with the complete DigitalWatch, in the RING component.

Compiling to DES We now describe the compiling process. The compiler first parses the given file and produces an AST, then performs scoping analyses and builds symbol tables. Finally, the AST is given to the statechart visitor where the primary objective is to build an easy-to-manipulate data structure holding the hierarchical information of the given statechart. No naming verification is done yet since we allow use-before-declaration. We only permit *package*, *class*, *model* or *partial model* declarations at the top-level. In our example, we chose *package*. Within this top-level, there can only be *class* declarations that extend the appropriate Modelica library class and conform to the previous rules of specification.

The hierarchical information is stored as follows. For each user-defined class, a Python dictionary (equivalent to a hashmap) entry is created where the key is the name of the class and the value is a list. The first element of the list is either a C for composite, an O for orthogonal or an R for root. The second and subsequent entries are all the declared children and hyperedges of that class. Each entry is a list, where the first element is the type of the declaration, the second entry is the identifier of the declaration and the third entry is a list of modifiers for that declaration. These modifiers are stored as [modifierName, value] tuples. Once the hierarchical information is complete, we verify the hierarchy for correct statechart semantics.

First we make sure that if, in a given component, an orthogonal component is present, then only orthogonal components are present within that component.

Second, we verify that each Hyperedge declaration has *src* and *des* modifiers. We also check if the given node in the modifier is within the scope of the component. Third, we make sure that all user-defined components that are instantiated within other components are actually declared. Finally, we verify that each component has a valid default child and format the output for the compiler to return.

Appendix D gives the produced DES[2] code of the statechart in Figure 2. The format of the DES specification file is as follows: each section is preceded by a descriptor. There are many descriptors, but those of importance to us are STATECHART, ENTER, EXIT and TRANSITION. Under the STATECHART descriptor, the node hierarchy is given. If node B is a child of node A, then B must be written under and to the right of A. Moreover, **all** of A's children must have the same number of spaces from the left. If a node is a default node, the annotation [DS] (for Default State) is written next to the node. If the node is an orthogonal component, the annotation [CS] (for Concurrent State) is written. By the nature of orthogonal sections, they are always both default and concurrent. If a node is a history state, [HS] (for History State) is written. For example, in our DES code, START and PAUSE are children of RUNNING, where START is the default state. There is always only one STATECHART descriptor.

The ENTER (EXIT) descriptor is used to keep track of the enter (exit) actions of nodes. For each node with an enter (exit) action, an ENTER (EXIT) descriptor is written, followed by an N (S) to depict that we are entering (exiting) the state, the state's fully qualified name (separating each hierarchical parent with a "**.**"), an O to symbolize that there is an output action and finally the action itself. There can be more than one action, written on separate lines under the first O. The O is only written once. There can be zero or more ENTER (EXIT) descriptors, even more than one per node. In our example, there is only one ENTER action and no EXIT actions.

Finally, we prepare to write the hyperedge output. For each hyperedge found in the hierarchy, we write out the descriptor TRANSITION. This is followed by an S with the fully qualified name of the source node and an N with the fully qualified name of the destination node. Then either an E or a T is present. If the trigger is an event of type string, then E is used with the event. If there is no event, then T is used and a time of 0 with the annotation [RTT] is written out. The RTT stands for *repeated timed transition*, meaning that the transition is taken repeatedly at the time interval, in our case, 0. This is useful in the case of self-loops; in any other case, the transition will never repeat since the destination node will not be same as the source node, meaning the system will end up in a state other than the one slated for a repeated transition. If the event is a special macro AFTER(x), meaning that the transition should be taken every x seconds, then we write out a T followed by x and [RTT]. We always output a C for the guard. If there is no guard or the guard is *true*, we output a 1, otherwise we output the given guard. If there are many guards, they are written each on a separate line and, in entirety, the AND of all the guards is taken. Finally, if there is an action, we output an O with the action. Once again, there can be more than one action and they are separated one per line. The O is the only optional piece of information, all others are mandatory.

4 Conclusion

With the ultimate goal of standardization, a possible method by which modellers may model statecharts in Modelica is given. A compiler that takes such a statechart model and produces DES specification code is the major contribution of this work. The widespread use of the statechart formalism and the Modelica language further promoted this research.

Short-term future work includes cleaning up the code and documenting it, executing SCC automatically behind the scenes so that the user doesn't have to manually do this task and is solely left with the executable code. Long-term goals include rewriting the front-end so that is uses sableCC[4] and possibly allow declarations of meta-models so that when a model is specified, it can be checked against some meta-model.

References

[1] Bolduc, J and Vangheluwe, H., A modelling and simulation package for classical hierarchical DEVS. MSDL technical report MSDL- TR-2001-01, McGill University, (2001)

[2] Feng, H., DCHARTS, A FORMALISM FOR MODELING AND SIMULATION BASED DESIGN OF REACTIVE SOFTWARE SYSTEMS Master's thesis. McGill University. School of Computer Science. (2004)

[3] Song, H., Infrastructure for DEVS Modelling and Experimentation. Master's thesis. McGill University. School of Computer Science. (2006)

[4] Song, H., Infrastructure for DEVS Modelling and Experiment. Presentation. McGill University. School of Computer Science. (2006)

[5] Xu, W., The Design and Implementation of the μModelica Compiler. Master's thesis. McGill University. School of Computer Science. (2005)

[6] Xu, W., The Design and Implementation of the μModelica Compiler. Presentation. McGill University. School of Computer Science. (2005)

[7] Modelica Language Specification. Version 3.0, `http://www.modelica.org/documents/ModelicaSpec30.pdf` (2007)

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

Appendix A

```
package queue
  import devs.*;
  import externalfunctions.*;
  import simulator.*;
```

[4] www.sablecc.org

```
class GeneratorState
  Generator.SeqStates seqState(start=Generator.SeqStates.G_IDLE);
end GeneratorState;

class Generator
  extends AtomicDEVS;
  parameter Integer ia=0;
  parameter Integer ib=0;
  parameter Integer szl=0;
  parameter Integer szh=0;
  parameter String name="a";
  output DevsPort g_out;
  GeneratorState state();
  type SeqStates = enumeration(G_IDLE, G_GENERATING);

  function intTransition
  algorithm
    if( state.seqState == SeqStates.G_IDLE ) then
      state.seqState := SeqStates.G_GENERATING;
    elseif( state.seqState == SeqStates.G_GENERATING ) then
      state.seqState := SeqStates.G_IDLE;
    end if;
  end intTransition;

  function outputFnc
    DevsEvent evt = null;
  algorithm
    if( state.seqState == SeqStates.G_GENERATING) then
      evt := Job(szl, szh);
      poke(g_out, evt);
    end if;
  end outputFnc;

  function timeAdvance
    output Integer timespan;
  algorithm
    if( state.seqState == SeqStates.G_IDLE) then
      timespan:=randint(ia, ib);
    end if;
    if( state.seqState == SeqStates.G_GENERATING) then
      timespan := 0;
    end if;
  end timeAdvance;

end Generator;

class ProcessorState
  Job currentJob = null;
  list queue();
  Integer queueSize = 0;
```

```
    Real timeElapsed = 0.0;
    Processor.SeqStates seqState(start=Processor.SeqStates.P_IDLE);
  end ProcessorState;

  class Processor
    extends AtomicDEVS;
    parameter Integer qSize=0;
    parameter String name="a";
    input DevsPort p_in;
    output DevsPort p_out;
    output DevsPort p_discard;
    ProcessorState state();
    type SeqStates = enumeration(P_IDLE, P_BUSY, P_DISCARD);

    function initialization
    algorithm
      state.queueSize := qSize;
    end initialization;


    function extTransition
    algorithm
      if( state.seqState == SeqStates.P_IDLE ) then
        state.currentJob:=peek(p_in);
        state.timeElapsed:=0;
        state.seqState := SeqStates.P_BUSY;
      elseif( state.seqState == SeqStates.P_BUSY ) then
        state.timeElapsed:=state.timeElapsed+elapsed;
        if( len(state.queue)<state.queueSize ) then
          state.queue.append(peek(p_in));
          state.seqState := SeqStates.P_BUSY;
        elseif( len(state.queue)==state.queueSize ) then
          state.seqState := SeqStates.P_DISCARD;
        end if;
      end if;
    end extTransition;


    function intTransition
    algorithm
      if( state.seqState == SeqStates.P_BUSY ) then
        if( len(state.queue)==0 ) then
          state.timeElapsed:=0;
          state.currentJob:=null;
          state.seqState := SeqStates.P_IDLE;
        elseif( len(state.queue)>0 ) then
          state.timeElapsed:=0;
          state.currentJob:=state.queue.pop(0);
          state.seqState := SeqStates.P_BUSY;
        end if;
      elseif( state.seqState == SeqStates.P_DISCARD ) then
        state.seqState := SeqStates.P_BUSY;
      end if;
    end intTransition;

    function outputFnc
      DevsEvent evt = null;
    algorithm
      if( state.seqState == SeqStates.P_BUSY) then
        poke(p_out, state.currentJob);
        state.currentJob:=null;
      end if;
      if( state.seqState == SeqStates.P_DISCARD) then
        poke(p_discard, peek(p_in));
      end if;
    end outputFnc;

    function timeAdvance
      output Integer timespan;
    algorithm
      if( state.seqState == SeqStates.P_IDLE) then
        timespan:=INFINITY;
      end if;
      if( state.seqState == SeqStates.P_BUSY) then
        timespan := state.currentJob.size - state.timeElapsed;
      end if;
      if( state.seqState == SeqStates.P_DISCARD) then
        timespan:=0;
      end if;
    end timeAdvance;

  end Processor;

  class Root
    extends CoupledDEVS;
    parameter Integer ria;
    parameter Integer rib;
    parameter Integer rszl;
    parameter Integer rszh;
    parameter Integer rsize;
    parameter String name;
    output DevsPort r_out;
    output DevsPort r_discard;
    Generator g1(ia=ria, ib=rib, szl=rszl, szh=rszh);
    Processor p1(qSize=rsize);
    Processor p2(qSize=rsize);
    Processor p3(qSize=rsize);
  equation
    connect( g1.g_out, p1.p_in);
    connect( p1.p_out, r_out);
```

```
    connect( p1.p_discard, p2.p_in);
    connect( p2.p_out, r_out);
    connect( p2.p_discard, p3.p_in);
    connect( p3.p_out, r_out);
    connect( p3.p_discard, r_discard);
  end Root;

  class Job
    extends DevsEvent;
    parameter Integer szl;
    parameter Integer szh;
    Integer id = 0 "ID" ;
    Integer size = randint(szl, szh);
  end Job;

  class RootExperiment
    extends DevsExperiment;
    Root rootModel(ria=2, rib=5, rszl=3, rszh=10, rsize=2,
                   name="RootExperiment");
    Simulator sim(simModel=rootModel);
  end RootExperiment;

end queue;
```

Appendix B

```
package statecharts
  class BaseStatechart
  end BaseStatechart;
  class Hyperedge
    input BaseStatechart src;
    input BaseStatechart des;
    input String trigger = "";
    input Boolean guard = true;
    input String action = "";
  end Hyperedge;
  class Composite
    extends BaseStatechart;
    input Boolean default = false;
    input String enter = "";
    input String exit = "";
  end Composite;
  class Orthogonal
    extends BaseStatechart;
  end Orthogonal;
  class History
    extends BaseStatechart;
    input Boolean default = false;
    input Boolean deep = false;
  end History;
  class Basic
    extends BaseStatechart;
    input Boolean default = false;
    input String enter = "";
    input String exit = "";
  end Basic;
end statecharts;
```

Appendix C

```
package DigitalWatch
  import statecharts.*;

  class Mode
    extends Orthogonal;
    Basic RUN(default=true);
    Basic RUN_CHR();
    Basic RUN_ALM();
    Basic ALMOST_STOP();
    Basic STOP();
    Hyperedge h1(src=RUN, des=RUN, trigger="topRightPressed",
                 action="[EVENT('indiglo')]");
    Hyperedge h2(src=RUN, des=RUN_CHR, trigger="topLeftPressed",
                 action="[EVENT('chrono')]");
    Hyperedge h3(src=RUN, des=RUN_ALM, trigger="bottomLeftPressed",
                 action="[EVENT('alarm')]");
    Hyperedge h4(src=RUN, des=ALMOST_STOP, trigger="bottomRightPressed");
    Hyperedge h5(src=RUN_CHR, des=RUN_CHR, trigger="topRightPressed",
                 action="[EVENT('indiglo')]");
    Hyperedge h6(src=RUN_CHR, des=RUN, trigger="topLeftPressed",
                 action="[EVENT('time')]");
    Hyperedge h7(src=RUN_ALM, des=RUN_ALM, trigger="topRightPressed",
                 action="[EVENT('indiglo')]");
    Hyperedge h8(src=RUN_ALM, des=RUN, trigger="resume",
                 action="[EVENT('time')]");
    Hyperedge h9(src=ALMOST_STOP, des=RUN, trigger="bottomRightReleased");
    Hyperedge h10(src=ALMOST_STOP, des=STOP, trigger="AFTER(1.5)",
                  action="[EVENT('edit')]");
    Hyperedge h11(src=STOP, des=RUN, trigger="resume",
                  action="[EVENT('time')]");
    Hyperedge h12(src=STOP, des=STOP, trigger="topRightPressed",
                  action="[EVENT('indiglo')]");
  end Mode;

  class Running
    extends Composite;
    Basic START(default=true);
    Basic PAUSE();
    Hyperedge h1(src=START, des=START, trigger="AFTER(0.01)",
                 action="controller.increaseChronoByOne()");
    Hyperedge h2(src=START, des=PAUSE, trigger="bottomRightPressed",
                 guard="[INSTATE('NORMAL_MODE.MODE.RUN_CHR')]");
    Hyperedge h3(src=PAUSE, des=START, trigger="bottomRightPressed",
                 guard="[INSTATE('NORMAL_MODE.MODE.RUN_CHR')]");
  end Running;

  class Chrono
    extends Orthogonal;
    Basic DEF_CHR(default=true);
    Basic RESET(enter="controller.resetChrono()");
    Running RUNNING();
    Hyperedge h1(src=DEF_CHR, des=RESET, trigger="chrono");
    Hyperedge h2(src=RESET, des=RUNNING, trigger="bottomRightPressed",
                 guard="[INSTATE('NORMAL_MODE.MODE.RUN_CHR')]");
    Hyperedge h3(src=RUNNING, des=RESET, trigger="bottomLeftPressed",
                 guard="[INSTATE('NORMAL_MODE.MODE.RUN_CHR')]");
  end Chrono;

  class Normal_Mode
    extends Composite;
    Mode MODE();
    Chrono CHRONO();
  end Normal_Mode;

  class Statechart
    extends BaseStatechart;
    Basic Setup(default=true);
    Basic Stopped();
    Normal_Mode NORMAL_MODE();
    Hyperedge h1(src=Setup, des=NORMAL_MODE, trigger="start",
                 action="[DUMP('Starting the Digital Watch')]
                          \ncontroller=[PARAMS]");
    Hyperedge h2(src=NORMAL_MODE, des=Stopped, trigger="stop");
  end Statechart;

end DigitalWatch;
```

Appendix D

```
# Reehan Shaikh's Modelica Statechart Compiler
# Source: /home/reehan/src/cs763/project/muModelica/
#         example/digitalWatchStatechart/partial.mo
# Date:   Wed May 28 11:51:01 2008

CONNECTIONS:

STATECHART:
  Setup [DS]
  Stopped
  NORMAL_MODE
    MODE [DS] [CS]
      RUN [DS]
      RUN_CHR
      RUN_ALM
      ALMOST_STOP
      STOP
    CHRONO [DS] [CS]
      DEF_CHR [DS]
      RESET
      RUNNING
        START [DS]
        PAUSE

ENTER:
  N: NORMAL_MODE.CHRONO.RESET
  O: controller.resetChrono()

TRANSITION:
  S: NORMAL_MODE.MODE.RUN
  N: NORMAL_MODE.MODE.RUN
  E: topRightPressed
  C: 1
  O: [EVENT('indiglo')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN
  N: NORMAL_MODE.MODE.RUN_CHR
  E: topLeftPressed
  C: 1
  O: [EVENT('chrono')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN
  N: NORMAL_MODE.MODE.RUN_ALM
  E: bottomLeftPressed
  C: 1
  O: [EVENT('alarm')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN
  N: NORMAL_MODE.MODE.ALMOST_STOP
  E: bottomRightPressed
  C: 1

TRANSITION:
  S: NORMAL_MODE.MODE.RUN_CHR
  N: NORMAL_MODE.MODE.RUN_CHR
  E: topRightPressed
  C: 1
  O: [EVENT('indiglo')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN_CHR
  N: NORMAL_MODE.MODE.RUN
  E: topLeftPressed
  C: 1
  O: [EVENT('time')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN_ALM
  N: NORMAL_MODE.MODE.RUN_ALM
  E: topRightPressed
  C: 1
  O: [EVENT('indiglo')]

TRANSITION:
  S: NORMAL_MODE.MODE.RUN_ALM
  N: NORMAL_MODE.MODE.RUN
  E: resume
  C: 1
  O: [EVENT('time')]

TRANSITION:
  S: NORMAL_MODE.MODE.ALMOST_STOP
  N: NORMAL_MODE.MODE.RUN
  E: bottomRightReleased
  C: 1

TRANSITION:
  S: NORMAL_MODE.MODE.ALMOST_STOP
  N: NORMAL_MODE.MODE.STOP
  T: 1.5 [RTT]
  C: 1
  O: [EVENT('edit')]

TRANSITION:
  S: NORMAL_MODE.MODE.STOP
  N: NORMAL_MODE.MODE.RUN
  E: resume
  C: 1
  O: [EVENT('time')]

TRANSITION:
  S: NORMAL_MODE.MODE.STOP
  N: NORMAL_MODE.MODE.STOP
  E: topRightPressed
  C: 1
  O: [EVENT('indiglo')]

TRANSITION:
  S: NORMAL_MODE.CHRONO.RUNNING.START
  N: NORMAL_MODE.CHRONO.RUNNING.START
  T: 0.01 [RTT]
  C: 1
  O: controller.increaseChronoByOne()

TRANSITION:
  S: NORMAL_MODE.CHRONO.RUNNING.START
  N: NORMAL_MODE.CHRONO.RUNNING.PAUSE
  E: bottomRightPressed
  C: [INSTATE('NORMAL_MODE.MODE.RUN_CHR')]

TRANSITION:
  S: NORMAL_MODE.CHRONO.RUNNING.PAUSE
  N: NORMAL_MODE.CHRONO.RUNNING.START
  E: bottomRightPressed
  C: [INSTATE('NORMAL_MODE.MODE.RUN_CHR')]

TRANSITION:
  S: NORMAL_MODE.CHRONO.DEF_CHR
  N: NORMAL_MODE.CHRONO.RESET
  E: chrono
  C: 1

TRANSITION:
  S: NORMAL_MODE.CHRONO.RESET
  N: NORMAL_MODE.CHRONO.RUNNING
  E: bottomRightPressed
  C: [INSTATE('NORMAL_MODE.MODE.RUN_CHR')]

TRANSITION:
  S: NORMAL_MODE.CHRONO.RUNNING
  N: NORMAL_MODE.CHRONO.RESET
  E: bottomLeftPressed
  C: [INSTATE('NORMAL_MODE.MODE.RUN_CHR')]

TRANSITION:
  S: Setup
  N: NORMAL_MODE
  E: start
  C: 1
  O: [DUMP('Starting the Digital Watch')]
     controller=[PARAMS]

TRANSITION:
  S: NORMAL_MODE
  N: Stopped
```

```
E: stop
C: 1
```