Code Synthesis for Reactive Systems Using Class Diagrams & Statecharts

David Meunier
Supervised By: Hans Vangheluwe

April 29, 2006

1 Introduction

This paper describes the implementation and design of a code generator for reactive systems modeled using Class Diagrams and Statecharts. First, operational semantics of these two formalisms will be given up to the extent to which their features are implemented. For example, details on Statecharts' orthogonal components are omitted, since these are not supported by the current version of the compiler. Inter-object communication will then be considered and a description of reactivity to synchronous and asynchronous messages is given. This is followed by an overview of the event dispatching system required to handle asynchronous communication and the timing mechanism that governs object instances. Examples showing how to model reactive components are provided towards the end of this paper in the form of a case study. For the most part, the semantics adhered to follow those adopted by I-logix's RHAPSODY tool, which are described in [3].

The environment used to build models using this combination of Class Diagrams and Statecharts was the AToM^3 (A Tool for Multi-Formalism and Meta Modelling) tool developped at McGill University's Modelling, Simultaion and Design lab. The formalisms used to model Class Diagrams and Statecharts within AToM^3 were CD_classDiagramsV3 and DCharts respectively. The implementation of the code generator is written in Python. Currently, the only supported target language is Python, and consequently all details in this paper that refer to the compilation of models into code concern themselves only with that language.

2 Class Diagrams

Class Diagrams are used to describe the structure of an object as well as that of a system of objects. This modeling formalism typically provides a rich panoply of features that can be used to extensively describe object structure, the overall

interconnection of objects within the system, object cardinalities, role names, etc.

The current implementation, however, only supports code compiltation from class entities and associations. A class entity may inlcude an arbitrary amount of attributes and operations and it may be linked to another class entity in order to depict an associtaion from one object to another. Figure 1 shows a Class Diagram model composed of two class entities and an association between them. Due to limitations on the formalism used to model Class Diagrams, operations

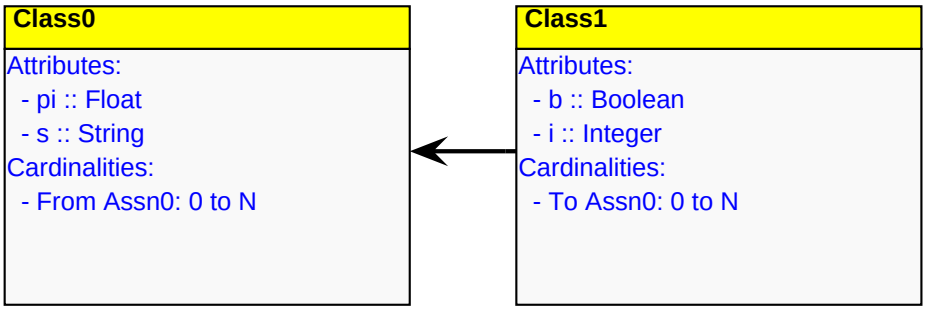

Figure 1: A Class Diagram in AToM^3

are emulated by a special attribute of type "Text" whose initial value is the operation's formal argument list. Optionally, a newline followed by statements implementing an operation may be added after the argument list and they will be executed whenever this operation is called. These statements, however, are executed *before* any action resulting from a state transition. Figure 2 shows the definition of an operation as a Text attribute. Statecharts are associated to

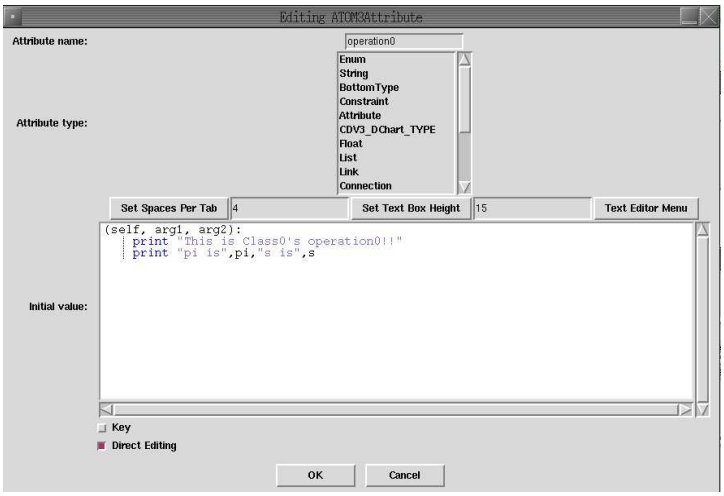

Figure 2: Adding an operation to a class entity

class entities by a giving them a special attribute of type CDV3_DChart_TYPE and setting its initial value to be the path of the statechart model.

The compiler interprets Class Diagrams by compiling them into code having the following structure. For each class entity fond in the Class Diagram, a corresponding class with the same name is generated. This class contains a function definition corresponding to each operation listed in the class entity. That is, the function's name, the specified argument list, and any accompanying code are written to the generated class.

If the class entity does not contain an `__init__` function, then one is created by the compiler. This default `__init__` function takes a single argument (in addition to the first argument, commonly named `self`, which is an implicitly passed reference to the called object) as a reference to the system's Dispatcher object. Otherwise, the `__init__` function is treated like any other operation defined in the class entity. Additionally, if the object had an associated statechart, its `__init__` function declares and initializes `self.CUR_TIME`, `self.CURRENTSTATE`, `self.TIMERS`, `self.DISPATCHER`, and `self.HISTORY`. It is important to note that if the modeler wishes to specify herself the `__init__` operation, then it **must** take the named argument, `dispatcher`, as it will be used in the initialization of `self.DISPATCHER`.

If the class entity has an associated statechart, then the `__init__` function also contains all the code resulting from actions entailed by entering the statechart's initial default state.

Each of these classes also contains a function, `process(self, evtList)`, that is used by the event processing mechanism. This and the aforementionned variables are considered to be reserved words; using them explicitely will most likely result in undesired behaviour from the generated code.

As for associations, they are used at the model level to denote that one object has a reference to another. This is, however, the extent of their significance. Associations do not enforce the navigational constraints they represent, and no restrictions get compiled in the generated code regarding message passing. They are merely used as a visual aid in understanding the structure of the software system.

3 Statecharts

Statecharts are used to model the reactive behavior of a system. They were first introduced in 1987 by Harel [1] but a formal definition of their semantics was not given until 1996 in [2]. These will henceforth be refered to as STATEMATE semantics, and those detailed in [3] will be refered to as the RHAPSODY semantics of Statecharts. Though the differences between RHAPSODY, STATEMATE and other existing semantics are not overwhelming,[1] this compiler tends to adhere to the former one with some variations.

Whereas Class Diagrams are used solely for their contribution to the structure of the generated classes, Statecharts are in fact the essence of practically all

[1] A comparative overview of these differences can be found in [4]

the behavior expected from the system. As a result, much of this paper is dedicated to explaining the syntax and semantics of statecharts and a great deal of attention is given to all of the thinkable ambiguous cases that may arise in their interpretation. This section deals mainly with syntax and basic terminology. The next sections tackle semantics and implementation issues.

3.1 States

States are the building blocks of statecharts. They represent a discrete mode an object may reside in at a given time. States come in two flavors,[2] **Basic** and **Composite**. Basic states are used to denote possible **active** configurations the object can occupy during runtime, while composite states, also known as OR-states, are used to group other states together thereby giving Statecharts a hierarchical structure. An object cannot be in a composite state without being in one of its basic substates. Also, substates within a composite are related to each other by exclusive-or, thus an object resides in either one substate or another. The states at the topmost level in the hierarchy are also related to each other in this manner. The question of which substate is chosen to become active if a transition leads to a composite is settled by introducing **default** states. Any state may be marked as default and any transition whose target is a composite will implicitly lead to that composite's default state, and this notion applies itself recursively until a default basic state is reached. If a composite contains more than one default state, then one is arbitrarily chosen by the compiler.

3.2 Transitions

Transitions are depicted by arrows leading from one state to another (or itself) and are the means by which an object changes its active state. For a transition to be taken (i.e. **fire**), up to three conditions must hold at the same time. Firstly and quite obviously, the object must reside in the state (or a substate of the state) that is the source of the transition. Secondly, the transition's **trigger** must be received by the object and lastly, the transition may have a condition, known as a **guard**, which prevents the transition from firing unless it evaluates to true. The two last requirements are optional. Transitions that do not have a trigger are called **null** transitions.

Triggers are strings that represent messages the object can receive. More on the different types of triggers and their meaning is given in a separate section. Guards, on the other hand, are boolean expressions written in the target language. It is common practice to label transitions *trigger(parameter list)[guard] / action*, where *action* is a set of statements carried out upon taking that transition, but this tends to severly clutter the model.

[2]This is not entirely true. Statecharts also have Orthogonal Components, also known as AND-states, which are not supported by this compiler. See the section on Future Work for more information.

3.3 History Connectors

History connectors are *pseudo*states located within a composite. They hold a reference to the last active basic state within their composite parent. When a transition whose target is a history connector is taken, the result is *exactly* as if the transition led directly to the referenced basic state. Although it is legal to have a history connector that is not contained within any composite state, it does not make much sense to do so since any transition leading to it ultimately results in a self-loop leading back to the source.

4 Transition Scope

Basic and composites states can have actions associated with them that are executed when they are entered (**entry action**) or exited (**exit action**). Transitions may also have an associated action that is performed when it is fired. All actions are written in the target language. When a transition is taken, the exit actions of all the exited states from the lowest to the highest are performed, followed by the transition's action, and lastly the entry actions of each entered state, from the highest to the lowest, are performed.

With this in mind, it becomes important to formally specify which states are exited and which are entered as the result of a firing transition. These states are the ones within the **scope** of the transition. The definition of scope used here is the same as the one given in [2] and states that it is the lowest OR-state in the hierarchy of states that is a proper common ancestor of the source and target states. When a transition is taken, the states exited are those that were proper descendants of its scope and in which the system resided at the beginning of the transition. The states entered are those that are proper descendants of that scope and in which the system will reside after that transtition is taken. For example, in Figure 3, the scope of transition t is Composite2. If t fires, Basic0 and Composite0 will be exited, and the entered states will be Composite1 and Basic1, since it is the default state of Composite1. It is important to realize that

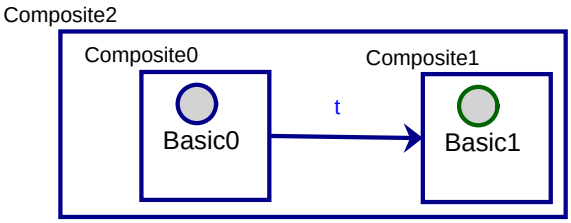

Figure 3: The scope of t is Composite2

it is the immediate source and target of a transition that determine its scope. This detail may affect the entered and exited states in models such as the one shown in Figure 4. Here the basic state Basic0 is active and Basic1 is marked as default. If f's scope was computed using the active state of the system (Basic0)

and the state in which the system will reside after taking f(Basic1), as the source and target of f, respectively, then f's scope would have been Composite0. In reality, the scope is the entire model, i.e. the Statechart's **root**.

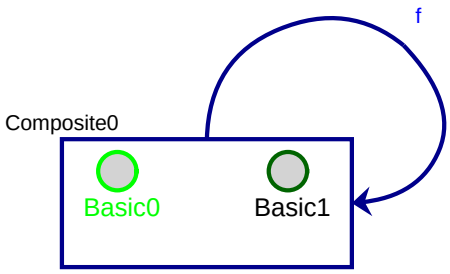

Figure 4: The scope of f is the entire model

5 Semantics of History Connectors

The syntax surrounding basic states bears similarities to that of history connectors: their graphical appearance is comparable, they may be marked as default, and they can serve as the source or target of a transition. Their semantics, however, express blatant differences. Most notably, history connectors do not represent a possible active configuration achievable by some object. They are merely a shorthand notation for what would otherwise become a combinatorial expansion in the size of a statechart's graphical layout.

Before looking at how they function, it is helpful to define the scope of a history connector. Quite simply, if C is a composite state containing the history connector H, then all of C's descendants are within H's scope, and H's scope encompasses no more than these states. This type of history is known to STATEMATE semantics as "deep history". Statemate also supports "shallow history" connectors, whose scope includes only the immediate children of the composite in which it resides. No further attention is given to shallow history connectors as they are currently not supported by this implementation, nor by RHAPSODY.

When a transaction fires, any history connector that has the object's active basic state within its scope is updated to hold a reference to that state. Conversely, when a transition leads to a history connector, it must first be resolved to a basic state, and only afterwards can the scope of the transition be computed.[3] The process of resolving a history connector is detailed in the following algorithm.

The bulk of the procedure worries itself about what goes on when a history connector does not yet have a reference to another state in memory. The idea

[3]This may appear to contradict what was said in the previous section, but keep in mind that history connectors are not real states!

Algorithm: resolve-history(H, S)
Input: A history connector, H, the active basic state, S.
Output: The actual target of a transition leading to H.

```
if H is not contained or S is in the scope of H
    return S
else
    C=composite containing H
if ref(H) is None
    if tr is a null, guardless transtion leaving H
        T=final target of tr
        if H is a history connector
            return resolve-history(T, S)
        else
            return T
    else
        for all null transitions, tr, leaving H
            if tr's guard evaluates to True
                T=final target of tr
                if T is a history connector
                    return resolve-history(T, S)
                else
                    return T
        return the default basic state of C
else
    return the basic state referenced by H
```

then is to pick some null transition leaving H and to use its target instead. If such a transtition exists, and does not have a guard, then it is chosen. If not, then other null transitions are considered and the first one found whose guard evaluates to true is chosen. If there are no null tansitions leaving it then the target is defined to be the default basic state of the history connector's composite parent. The reason that only null transitions are considered is that it does not make sense to wait for a trigger to be received by the object because this would imply the object's active state at that point is the history connector, which is not allowed. If the chosen transition has associated actions, then they will be performed after those of the transition that led to the history connector in the first place.

Figure 5 shows a statechart model involving a history connector. The system is in state Basic0 and when the transition e fires, its target will be Basic1 because History0 does not yet have a state reference in memory, and there are no null transitions leaving it so the default basic state of Composite1 is used. Afterwards, if f fires, followed by g, the system's active state will be Basic2,

since it was the last active state in History0's scope.

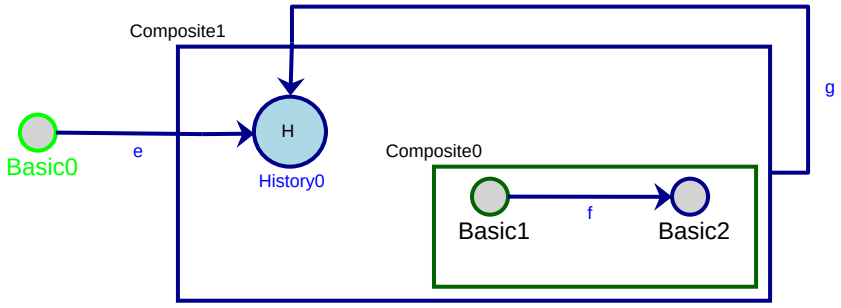

Figure 5: A history connector, History0

6 Object Reactivity

Up to now we have seen that an object may change its active state by means of a transition and we know precisely what effects it will have. That is, the task of computing the set of states that will be exited and entered, as well as the order of execution of their associated actions is well understood. This section investigates what causes a transtition to fire, more specifically, the types of messages that can be sent to objects as triggers.

6.1 Synchronous Messages

A message may be sent to an object in a synchronous manner simply by calling an operation defined in its class entity. There is no special syntax to be used by the caller. The modeler, for example, would simply write a method call within some action code as she would do so if she were programming directly in the target language.

The receiving object may use this operation to induce a state change by using the operation's name as a trigger. This is known as a **triggered operation**. The operational semantics applied when an object's triggered operation is called is to first execute the initial code, if any, specified in its definition (from its class entity). The exectution then follows with all of the actions resulting from the transition that gets fired as a consequence of the triggered operation. If a triggered operation is called on an object but its active state does not react to it, then the operation's initial code, if any, will still be executed, but no further actions will be taken and no state change will ensue.

If a triggered operation is to return a value, say x, then it must do so within the action code of the transition by using *REPLY(x)*. Using the target language's return statement will pre-empt the operation before it can perform the necessary entry actions and internally update the active state. The statement

REPLY(x) gets translated by the compiler into `RETVAL=x`, which simply stores the return value into a variable that is local to the function. This value is then returned to the calling object after the appropriate actions are performed. This detail is important to keep in mind when writing the actions implementing a triggered operation. For example,

```
x=5
if x==5:
    return True
return False
```

will return `True` whereas

```
x=5
if x==5:
    REPLY(True)
REPLY(False)
```

will return `False`.

In order to avoid confusion, it is worthwhile to mention that not all operations listed in an object's class entity need be used as triggered operations in the object's statechart. An operation's implementation may be contained entirely in the code found in the class entity's definition for that operation. Such operations are referred to in [3] as **primitive operations**. When the modeler choses to write a primitive operation, she makes the design decision that invoking this operation should not influence the object's state. Because primitive operations have no connection with the object's statechart, the compiler does not bother with *REPLY()* statements and these should not be used in this context. Primitive operations that must return a value do so using the return statement provided by the target language.

6.2 Asynchronous Messages

The second type of trigger a transition can respond to is called an **event**. Events are sent as asynchronous messages by an object through the use of the *GEN(targetObj, e, p1, p2)* statement. This sends the event *e* with parameters *p1* and *p2* to the object referred to as *targetObj*. In this example, two parameters were sent with the event but there may be an arbitrary number of them (or none). This statement gets compiled as a command that adds the event (as a string) along with any parameter to an event queue specific to the target object. These event queues are held and maintaned by a Dispatcher object, which will be the focus of a separate section.

6.3 Timed Transitions

It is possible to schedule a transition to fire after a certain timeout. This so-called **timed transition** is achieved by giving it a special trigger named *AFTER(dt)*. This transtion will then fire on its own provided the object remains

in the source state for at least dt seconds. If the source of a timed transition happens to be a composite state, then the transition will fire after the specified time as long as it has not exited that composite prior to the timeout. That is to say, state changes that keep one of this composite's substates active will not cancel or otherwise affect the countdown of the timed transition.

The generated code internally maintains a list of special events named `_iafter` where `i` is an integer. Each of these events are associated with a single timed transition within a statechart. When an object's active state remains the source of a timed transition for the required time, the object generates the appropriate special event to be dispatched to itself.

7 Event Processing

The following algorithm describes the steps taken by the `process(self,evtList)` function defined in each object. This function is invoked by the Dispatcher on each object at roughly regular time intervals. The argument `evtList` is a list of all the events and their parameters accumulated by the Dispatcher for a particular object. One thing to note is that when an event is sent to an object,

Algorithm: process(*evtList*)
Input: The list of events to be processed *evtList*.
Output: None.

```
C=current active state
generate events resulting from a timeout
while evtList not empty
    evt=evtList.pop()
    bind evt's actual parameters to local variables
    if active state reacts to evt
        t=transition reacting to evt whose guard
          evaluates to true
        S=scope of t
        update all appropriate history connector
          references to S
        if t leads to a history connector, H
            T=history-resolve(H, S)
            S=lowest common ancestor of {C, T}
        N=last (basic) state in S to be entered
        execute required exit actions
        execute required transition actions
        execute required entry actions
        update active state to N
        if N has an outgoing null transtion
             process([None])
```

it is effectively consumed regardless of whether or not the object is in a state reacting to it. Events are not "remembered" throughout the execution of the processing function or any time afterwards in the life of the object.

Though this procedure applies to processing events, the same steps are taken in the case of a triggered operation. Obvious exceptions being that there is no list of events to deal with and the triggered operations' variables are already bound by the target language.

7.1 Nondeterminism

In order to decide whether a transition can react to the event being processed or the invoked triggered operation, that is, whether it is **enabled**, its name alone (i.e. the type or amount of parameters or arguments passed along are not used) is compared to the triggers of the transitions leaving the active state and all its composite ancestors. The reacting transition is then chosen among those with matching triggers provided its guard evaluates to true if it has one. It may be possible that multiple transitions are enabled at a given time. There are two such types of situations. First, there is the case where the enabled transitions have different sources. This is resolved by prioritizing the "innermost" transition, i.e. the transition whose source is the closest ancestor to the active basic state. The second case occurs when there are multiple enabled transitions, none of which are closest to the active basic state. This results in nondeterminism and the code generator deals with this by taking the first transtion it comes across. These two cases are depicted in figures 6 and 7, where the labels on the transtions represent their triggers.

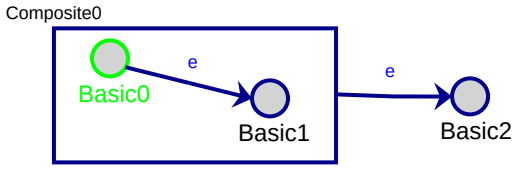

Figure 6: The transition to Basic1 is prioritized.

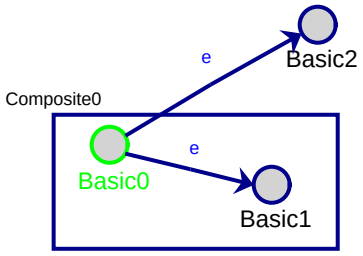

Figure 7: A nondeterministic situation.

7.2 Dealing With Null Transitions

Null transitions are always taken instantaneously. The last lines of the above algorithm accomplish this by recursively invoking the process function with a list containing only the event None as its argument. Null transitions have no triggers and thus are not compared to the event popped from the list. Thus the transtion is taken immediately if there is no guard or if it evalutates to true. The issue of loops of null transitions arises, and at a first glance it may seem that any compiler allowing this is badly designed. However, the generated code's behavior does indeed remain faithful to the model being compiled. It is, in this case, the statechart model that is flawed.

8 Event Dispatching

In order to achieve asynchronous messaging by means of events, a mechanism is needed that will allow one object to process an event while another one proceeds with its regular execution. The design adopted was to have the compiler produce a special Dispatcher class of which a single instance would exist throughout the system. All objects receive a reference to the Dispatcher upon instantiation and the creation of the Dispatcher object itself is intended to be done by the application using the classes generated from the model. The Dispatcher provides an interface allowing objects to be attached and detached to it in order to maintain distinct event queues for each respective object instance in the system. When the *GEN(targetObj, e, p1, p2)* statement appears in some action code, it gets compiled into a command that invokes the Dispatcher and appends the event and its parameters to the appropriate queue.

The Dispatcher emulates concurrency among object instances much like an operating system allocates computing resources to various running processes. The event queues are considered one at a time, treating each object instance in a round-robin fashion. The Dispatcher invokes `process(evtList)` on each instance and passes its queue of accumulated events as the argument. It is also the invokation of the process function that allows objects to examine the current system time and possibly generate special events in order to fire timed transitions. These special events are immediately appended to the list to be processed in order to avoid delays incurred by having to wait until the next time the Dispatcher calls process for the transition to fire.

9 Timing

Intimately linked to dispatching and also necessary to achieve asynchronous communication between objects is a timing mechanism governing the entire system. The approach taken here was to use a time-slicing technique that would give the Dispatcher a fixed amount of time for it to dispatch events to objects and have them process them accordingly. This approach is one commonly taken in

embedded systems and computer games, the latter's computations being driven by the need to execute within a somewhat fixed frame rate.

This timing scheme can be implemented as part of the Dispatcher class. The following algorithm covers this functionality. Essentially all object queues

```
Algorithm: processAllLoop()
Input: None.
Output: None.

while there exist non empty event queues
    O=next object with non empty queue
    Q=event queue of O
    O.process(Q)
reschedule processAllLoop after 2ms
```

are processed until they are all empty at which point the function reschedules itself after a fixed amount of time. It is this self-rescheduling that creates the time-sliced behaviour of the system. The function itself will typically take much less than 2ms to consume all the event queues. The time gap between the end of the processAllLoop function and its next scheduled execution is intended to give other processes the chance to take control of the CPU. This relinquishment of system resources has proven to be much more efficient than keeping the Dispatcher in a busy loop.

The processAllLoop function also has another important role. It emulates the system's main loop. The application using the generated code is intended to instantiate a Dispatcher object along with any other object it requires, and then make a single call to processAllLoop, which then takes control of the system and acts on its own.

The development of a time-sliced environment avoids the complications and potential errors related to threading. In some cases it is more efficient to implement the timing-slicing directly within the application. For this reason, the Dispatcher also provides a function named processAll that is identical to processAllLoop except that it does not reschedule itself automatically. For example, the case study presented next uses Tkinter primitives to schedule calls to processAll at fixed time intervals. This achieves significantly greater overall CPU efficiency.

10 Case Study: A Wristwatch Application

In order to demonstrate the compiler's correctness and to illustrate the concept of modeling a reactive system, it was used to generate the code implementing the behavior of a wristwatch.

10.1 Requirements

Given is the GUI (figure 8) for a wristwatch and the interface it provides. The task is to the part of its backend describing its behavior such that the following requirements are met.

- The time value should be updated every second, even when it is not displayed (as for example, when the chrono is running). However, time is not updated when it is being edited.
- Pressing the top right button turns on the Indiglo light. The light stays on for as long as the button remains pressed. From the moment the button is released, the light stays on for 2 more seconds, after which it is turned off.
- Pressing the top left button alternates between the chrono and the time display modes. The system starts in the time display mode. In this mode, the time (HH:MM:SS) and date (MM/DD/YY) are displayed.
- When in chrono display mode, the elapsed time is displayed MM:SS:FF (with FF hundreths of a second). Initially, the chrono starts at 00:00:00. The bottom right button is used to start the chrono. The running chrono updates in 1/100 second increments. Subsequently pressing the bottom right button will pause/resume the chrono. Pressing the bottom left button resets the chrono to 00:00:00. The chrono will keep running (when in running mode) or keep its value (when in paused mode), even when the watch is in a different display mode (for example, when the time is displayed).
- When in time display mode, the watch will go into time editing mode when the bottom right button is held pressed for at least 1.5 seconds.
- When in time display mode, the alarm can be displayed and set on or off by pressing the bottom left button. If the bottom left button is held for 1.5 seconds or more, the watch goes into alarm editing mode. Initially, the alarm time is set to 12:00:00. The alarm is activated when the alarm time is equal to the time in display mode. When it is activated, the screen will blink for 4 seconds, then the alarm turns off. Blinking means switching to/from highlighted background (Indiglo) twice per second.The alarm can be turned-off before the elapsed 4 seconds by a user interrupt (i.e.: if any button is pressed). After the alarm is turned off, activity continues exactly where it was left-off.
- When in (either time or alarm) editing mode, briefly pressing the bottom left button will increase the current selection. Note that it is only possible to increase the current selection, there is no way to decrease or reset the current selection. If the bottom left button is held down, the current selection is incremented automatically every 0.3 seconds. Editing mode should be exited if no editing event occurs for 5 seconds. Holding the

bottom right button down for 2 seconds will also exit the editing mode. Pressing the bottom right button for less than 2 seconds will move to the next selection (for example, from editing hours to editing minutes).

The interface provided by the GUI has the following functions.

- getTime() Returns the current clock time.
- getAlarm() Returns the alarm time set.
- refreshTimeDisplay() Redraws the time with the current internal time value. The display does not need to be cleaned before calling this function. For instance, if the alarm is currently displayed, it will be deleted before drawing the time.
- refreshChronoDisplay See refreshTimeDisplay()
- refreshDateDisplay() See refreshTimeDisplay()
- refreshAlarmDisplay() See refreshTimeDisplay()
- resetChrono() Resets the internal chrono to 00:00:00.
- startSelection() Selects the leftmost digit group currently displayed on the screen.
- increaseSelection() Increases the currently selected digit group's value by one.
- selectNext() Selects the next digit group, looping back to the leftmost digit group when the rightmost digit group is currently selected. If the time is currently displayed on the screen, selects also the date digits. If the alarm is displayed on the screen, doesn't select the date digits.
- stopSelection() Turns off selection.
- increaseTimeByOne() Increase the time by one second. Note how minutes, hours, days, month and year will be modified appropriately, if needed (for example, when increaseTimeByOne() is called at time 11:59:59, the new time will be 12:00:00).
- increaseChronoByOne() Increase the chrono by 1/100 second.
- setIndiglo() Turn on the display background light
- unsetIndiglo() Turn off the display background light
- setAlarm() Flag the alarm to be on or off.

When a button is pressed or released, one of the following events is set to the statechart (as strings): topRightPressed, topRightReleased, topLeftPressed, topLeftReleased, bottomRightPressed, bottomRightReleased, bottomLeftPressed, bottomRightReleased, alarmStart.

10.2 Design

The design was chosen to consist of a Watch class associated to TimeDisplay, ChronoDisplay, AlarmDisplay and Indiglo components. An instance of each object is created and the instance of the Watch class serves to receive events from the GUI and send them to the appropriate component. The resulting Class Diagrams and Statecharts are shown in the following figures.

Figure 8: The wristwatch GUI.

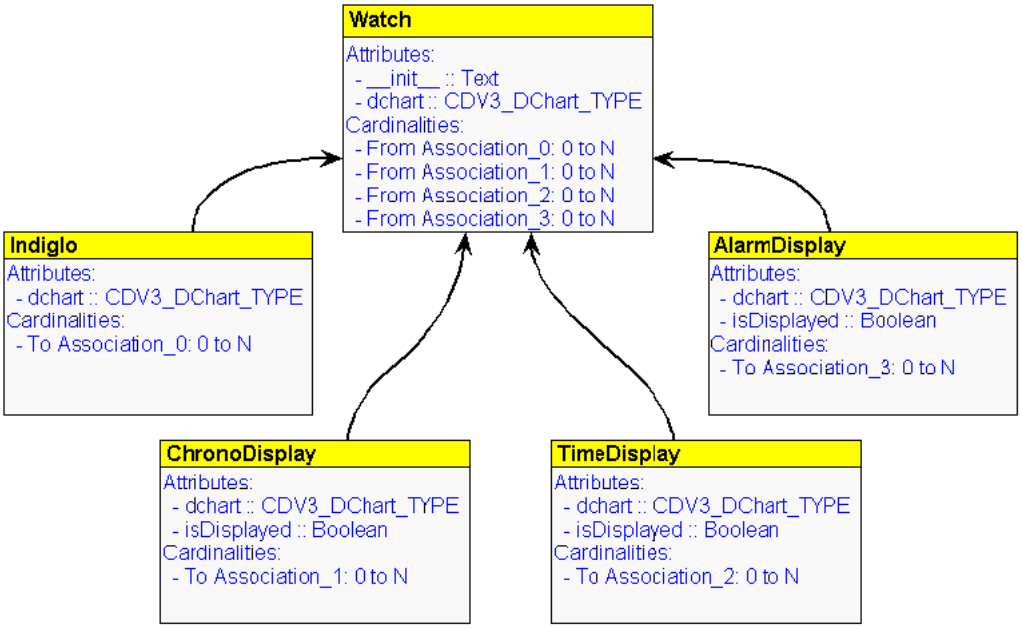

Figure 9: Class Diagram for behavioral components.

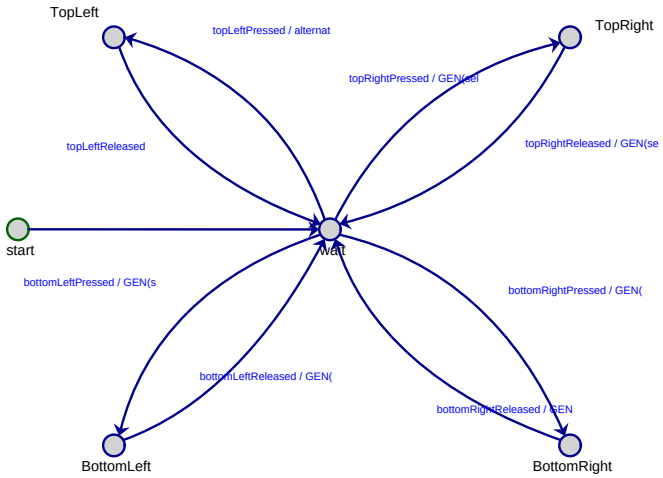

Figure 10: Statechart for the Watch class.

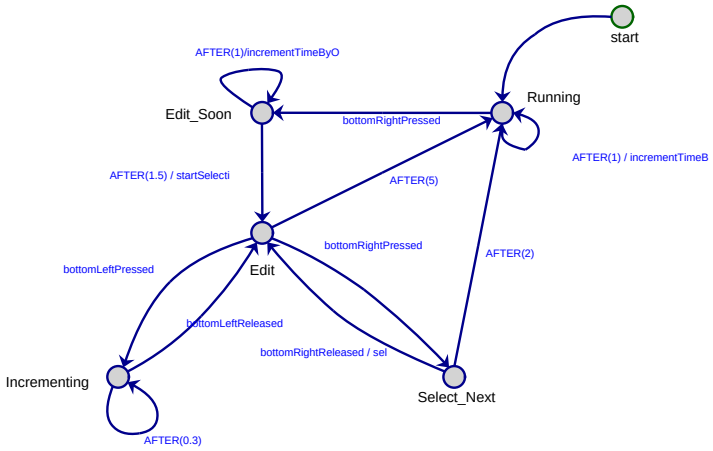

Figure 11: Statechart for the TimeDisplay class.

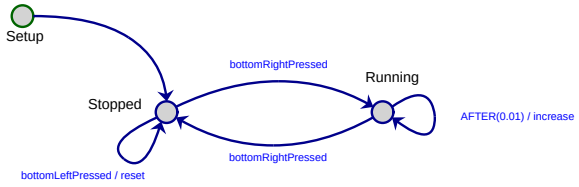

Figure 12: Statechart for the ChronoDisplay class.

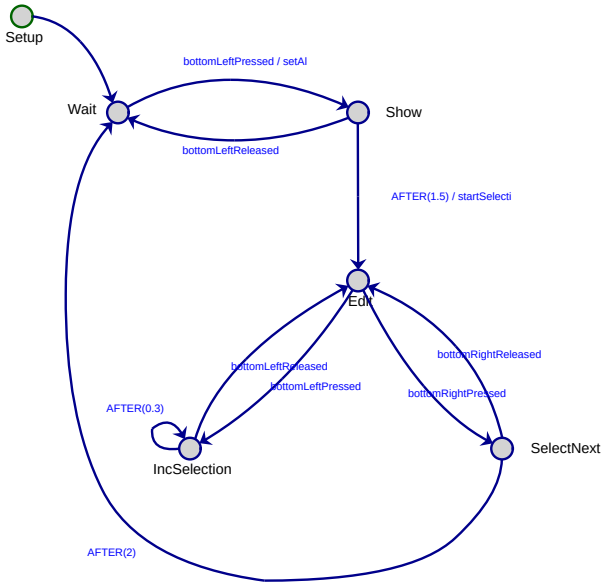

Figure 13: Statechart for the AlarmDisplay class.

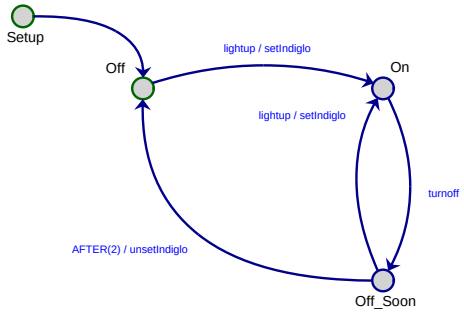

Figure 14: Statechart for the Indiglo class.

11 Future Work

This section contains various implementation details, variations on the RHAPSODY semantics, design decisions, and suggests alternate possibilites or potential extensions to this compiler.

Orthogonal Components are a feature of Statecharts that allow the system to be in multiple *orthogonal* states at once. This enables concurrency within one Statechart. This compiler achieves orthogonality between object instances but not within an object itself. It is therefore possible to emulate each orthogonal component by an object instance. However, in practice this is not only

cumbersome but leads to unintuitive segregation of software components that really belong in the same class. This alone is enough reason to add support for orthogonal components.

History is accompished by keeping a variable for each history connector to hold the name of the last active state in its scope. For each transition ultimately leading to a history connector, the compiler generates the code for necessary exit, entry, and transition actions *for every possible state referenced*. Although this causes the generated code to be extremely efficient, it can easily explode in length and be difficult to read. Another solution would be to have the model itself stored in the generated code. Then actions associated with states and transitions could be stored as attributes of the model and accessed the same way regardless of what state is in the memory of the history connector. This would abstract away the details of what actions get executed depending on what state is referenced.

Loop detection is another important feature that is worth adding. It is acceptable (and indeed a requirement) for the generated code to contain infinite loops if it was intended by the model. However, loops caused by null transitions or *AFTER(0)* triggers are usually unintentional errors or simply bad design and can be easily detected and reported by the compiler.

Static Reactions are supported by the RHAPSODY tool but not by this compiler. They are similar to transitions in that they react to triggers and can produce actions. The difference is that they are associated with a state and do not cause the active configuration of the object to change. Static reactions can easily be emulated with orthogonal components, but their support can easily be implemented within the compiler.

A new formalism for Class Diagrams is definitely needed. This would allow for many features such as class attributes, templates, role names, etc. to be used. Also, enforcing navigational constraints expressed by associations would be tremendously beneficial.

Support for other languages would also, naturally, be a great thing to have.

12 Acknowledgments

I would like to thank my supervisor, Prof. Hans Vangheluwe, for his continued support and great advice, and also Denis Dubé for creating the CDV3_DCHART_TYPE attribute type and for his help on adding it to the CD_Class-Diagrams_V3 formalism.

References

[1] David Harel.Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, volume 8. 1987. pp.231-274.

[2] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Enginneering and Methodology (TOSEM)*, volume 5, issue 4 (October 1996). pp.293-333

[3] David Harel and Hillel Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). *Lecture Notes in Computer Science*, volume 3147, January 2004. pp.325-354.

[4] Michelle L. Crane and Juergen Dingel. UML Vs. Classical Vs. Rhapsody Statecharts: Not All Models Are Created Equal.*Lecture Notes in Computer Science*, volume 3713, November 2005. pp.97 - 112.