Statecharts and Class Diagram XML
A general-purpose textual modelling formalism

Glenn De Jonghe

Supervisor: Prof. Dr. Hans Vangheluwe

Faculty of Science
University of Antwerp
Antwerp, Belgium

March 1, 2014

Abstract

Due to the exponential increase in complexity of video games, writing consistent and re-usable code for game AI has become very hard. In previous research, a new method was presented to obtain intelligent behaviour for Non-Playable Characters (NPCs) based on the visual modelling formalism Statecharts. In this paper we expand on that research, creating a new formalism based on both Statecharts and the actor model. The formalism will incorporate class diagrams to have instantiated classes as actors, which have their behaviour described by a statechart. The modularity that comes forward has as advantage that new models can easily be generated where one or more actors are replaced or adapted. In the case of NPCs this results in models with similar yet sufficiently different behaviour that allows for a more interesting game-play. The proposed state charts and class diagrams XML (SCCDXML) formalism is based on SCXML, a formalism that describes state charts in a human-readable XML format. A compiler is developed to generate code from SCCDXML models, followed by an example tanks game where it is shown that the formalism can easily be used to create intelligent behaviour for computer controlled tanks. Many advantages can be observed : not only does the layer of abstraction result in very simple but powerful designs, the obtained code is also more stable and efficient.

Contents

List of Figures

Code Samples

1 The Formalism

In this chapter our proposed Statecharts and Class Diagram XML (SCCDXML) formalism is described. In the first section the semantics of the formalism are explained using a neutral visual representation. The second section gives a detailed overview of the concrete syntax together with SCCDXML versions of the visual examples.

1.1 Constructs

In this section the semantics of the different constructs that make up the formalism are discussed. We can split the constructs up into two categories, namely those that make up a statechart and those that encapsulate those statecharts into a class and ultimately into a class diagram.

1.1.1 Statechart

The statechart-constructs are based on the Statecharts formalism, first introduced by David Harel [Har87] in 1987. Statecharts is a visual modelling language that represents finite state automata with added hierarchy, parallelism, history and broadcast communication. Harel created the formalism to be able to describe large and reactive systems, as he believed such method wasn't available at that time. We use the visual representation as proposed by Harel in his original paper [Har87], but keep in mind that these can slightly differ across different statechart editors. We end this section with a quick overview of the formal semantics.

Basic State

The basic state acts as the main building block of a statechart and is represented as a labelled (rounded) rectangle. This can be seen in Figure 1.1 where two basic states are depicted. A statechart consisting solely out of basic states, has to have exactly one default/initial state, this is represented by an incoming edge without source node.

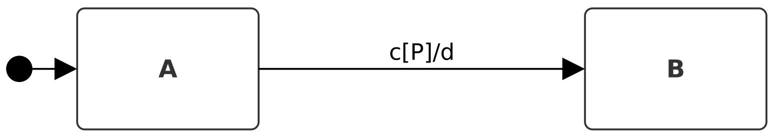

Figure 1.1: Two basic states connected by a transition originating from the initial state on the left.

Transition

The two states are connected by a transition originating from the initial state on the left, with a label of the form *event[guard]/action*. This means that upon reception of the trigger event *c*, the statechart will transition from the initial state *A* to the state *B* if and only if the guard condition *P* evaluates to *true*. Upon firing the transition, the action will be executed which in this case is the casting of the *d* event. All three parts of the label are optional and thus it is for example possible to have a transition without trigger event or guard condition (which is called an unconditional transition) or a transition with only a guard and no action.

Composite State

Composite states add a notion of hierarchy to Statecharts, they allow to group states together into multiple substates. The composite state is also called the XOR state because when such a state is active, exactly one of its substates must be active. Each composite state should have exactly one initial substate and transitions can occur at every level of the state hierarchy.

To illustrate this we look at the example in Figure 1.2. At initialization time only the state *A* is active. Upon reception of the event *h*, the composite state *B* will be entered and consequently its initial substate *C* as well. At this point an event *k* can bring the statechart in the state configuration where *B* and its substate *D* are active, while an event *f* would bring the statechart back to its initial configuration where only state *A* is active.

Parallel State

Besides the XOR decomposition achieved by a composite state, also AND decomposition is available in the Statecharts formalism. These are better known as parallel states and allow for parallelism to be modelled. Upon entering a parallel state, each substate will become active. These substates are always non-atomic states that can in turn contain a state hierarchy.

A parallel state is represented the same way as a composite state, however its substates are depicted by dashed rectangles expressing that they are active at the same time. We see such a state in Figure 1.3 labelled

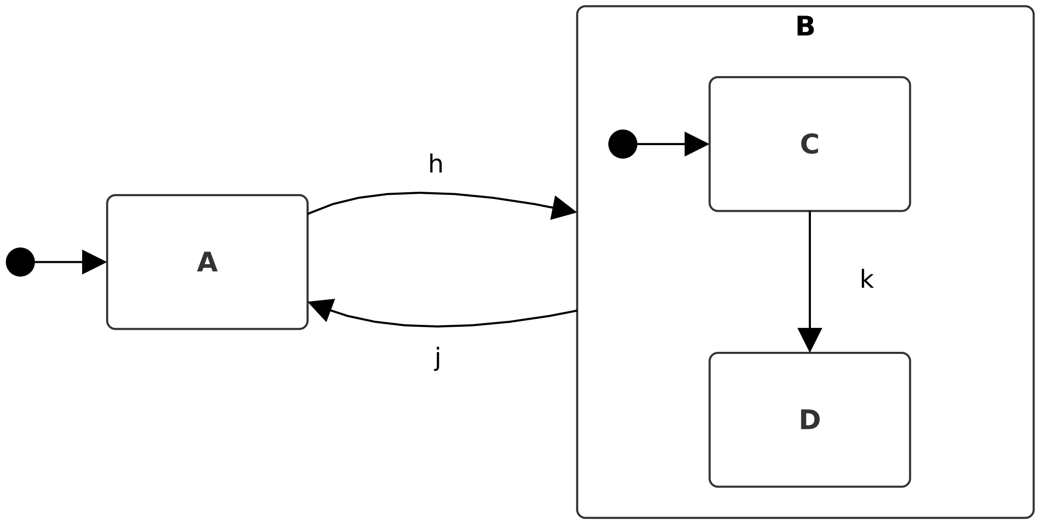

Figure 1.2: On the right we can see the composite state *B* encapsulating the two basic states *C* and *D*.

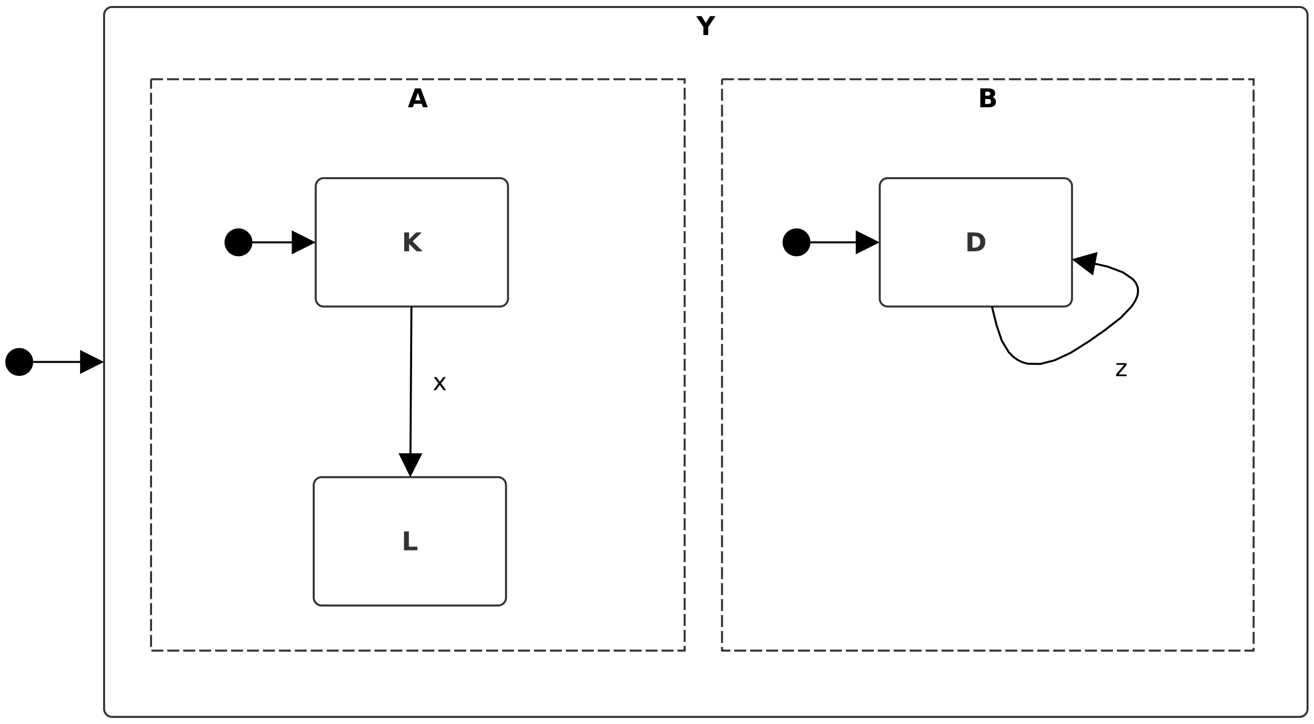

Figure 1.3: A parallel state *Y* with two composite substates *A* and *B*. Upon initialization both inner states *K* and *D* are active.

Y. Since this is the default state at the top level, this will directly be entered upon initialization. Consequently both substates *A* and *B* will be entered, which ultimately results in both inner states *K* and *D* being active at the same time. When the transition of *K* to *L* is triggered by the event *x*, state *D* will still remain active.

History State

A history state, which is depicted by a circle with the label *H*, adds memory to a component. Upon leaving a composite state, a possibly present history state will first save the current state before adapting it. When afterwards that composite state gets re-entered through the history state, the state will restore the retained state instead of using the default state.

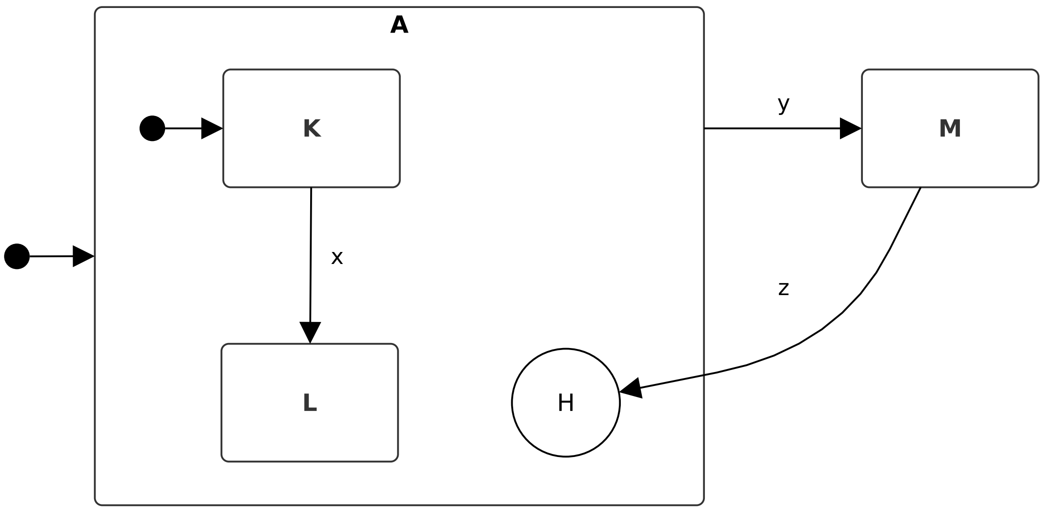

Figure 1.4: When the transition to the history state is fired, the state of *A* will be restored to its last recorded state.

To illustrate this we see in Figure 1.4 a composite state *A* that has a history state and two sub-states of which *K* is the default one. If an event *x* is received after initialization, the composite state will reside in sub-state *L* (*i.e., L* is now active). Upon firing the transition to state M, which is enabled by the event *y*, the current state of *A* is recorded first. When this is followed by an event *z*, the transition to the history state is taken resulting in *A* being reactivated and thus having the saved state, where sub-state *L* is active, restored.

Statecharts offers two types of history. The default *shallow* type only saves one layer of state in a component while the *deep* type saves all descendants of the component. The latter is represented by adding an $*$ to the state label giving $H*$.

Enter and Exit Hierarchy

With Statecharts it's possible to define actions that should be executed on either entering or exiting a specific state. When multiple layers of hierarchy are traversed on firing a transition, these actions are executed in the intuitive way. The exit actions are executed first, from the deepest level up to the first shared parent between the source and target states. This is then followed by executing the enter actions in the opposite direction up to the target states.

Formal Semantics

Statecharts models solely define the behaviour of a system, they do not provide a definition of how to execute or simulate the behaviour. This is where formal semantics come into play. Reliable simulation can only be performed when the syntactic constructs discussed in the previous subsections are associated with formal semantics. Many different semantics have spawned over the years with the most popular being UML statechart diagrams (as specified in UML 2.0 [OMG05]), Statemate [DH96] and Rhapsody [DH04]. The Statemate semantics were the first executable semantics for the modelling language introduced by Harel and Naamad in 1987. In 1996 Harel and Kugler followed this up with executable semantics for object-oriented statecharts known as the Rhapsody semantics. UML statechart diagrams are also an object version of Statecharts, very similar to Rhapsody due to cooperation between the development teams. Our formalism will primarily use the semantic definitions of the latter. For one this means that a transition *may* take time allowing for models with either zero- or fixed-execution time. This is in contrast with Statemate where the synchrony hypothesis holds which states that a system must react immediately to external events and that the corresponding output must occur at the same time. This leads to another important difference. In order for Statemate to adhere to the hypothesis it allows for events to occur simultaneously, and be acted upon simultaneously. This is not the case for our formalism. UML and Rhapsody adhere to the concept of run-to-completion, which means that each event is handled completely before the next event is processed.

Another noteworthy property is the order of execution of actions. While in Statemate multiple actions on a transition are executed in parallel, our formalism will execute them sequentially. As for the priority of conflicting transitions our formalism provides a way of specifying how this should be resolved, as explained in subsection 1.2.3.

1.1.2 Class Diagram

The top level of a SCCDXML model resembles a UML class diagram connecting different classes using relationships. Each class can have methods and attributes just like in UML, but on top of that our formalism defines the behaviour of a class using a statechart. In Figure 1.5 we see the class diagram from the perspective of a single class *ClassD*. We see that the class is associated with the classes *ClassE* and *ClassF* by named unidirectional associations and that a dashed edge with label *«behaviour»* is used to link the class to a statechart.

In the traditional Statecharts formalism, when casting an event, it was obvious that the scope of an event was local to the statechart. Now with the addition of the class diagram, different levels of scope are added. First of all the local scope naturally remains *i.e.,* when an event is raised with local scope only the source statechart will be able to act on it. Next we have a global scope that will make events visible for all currently instantiated statecharts and besides that an event can also be directly send to a statechart using the narrow scope. Similar to the latter a special *class diagram* scope is present that is used for management events like the creation of new instances. And last there is an output scope for attaching events to output ports in order to communicate with components outside of the model.

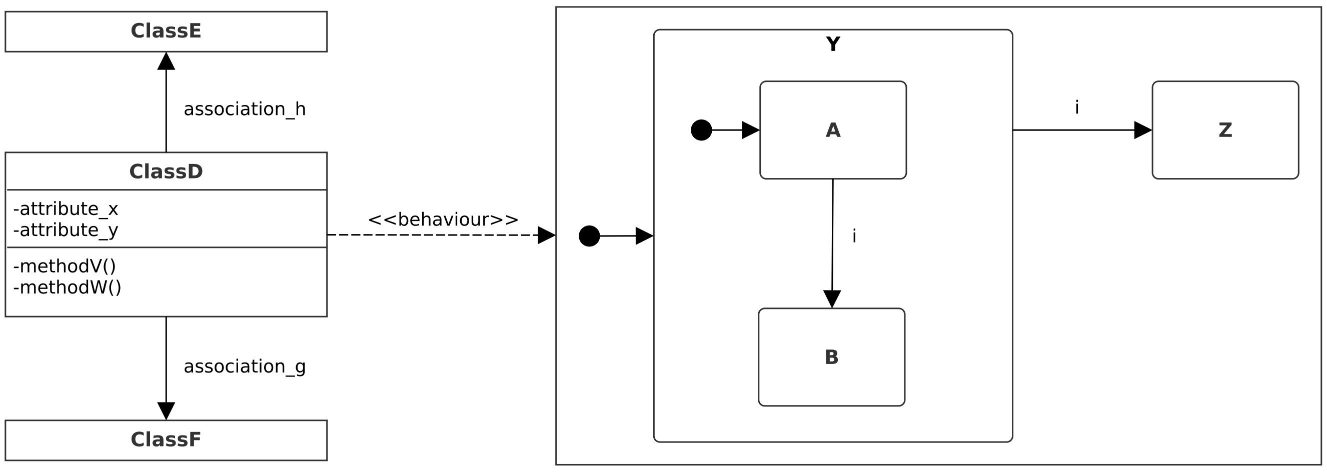

Figure 1.5: This figure illustrates the relation between classes in a class diagram and the statecharts that describe their behaviour.

Actor Model

The formalism can be seen as a case of the actor model [HBS73]. The actor model is a mathematical model for concurrent, fault-tolerant and highly scalable computation that has *actors* as its main building blocks which communicate using messages. In response to a message that it receives, an actor may:

- Make local decisions and changes.
- Create more actors.
- Send messages to other actors it has access to.
- Determine how to respond to the next message received.

The inherently asynchronous nature and absence of a shared state lends itself for highly parallel or distributed implementation. In our formalism instances of classes will be the actors, all having a statechart defining their behaviour. The messages are the events (which can thus also be send between actors) and these events will automatically be forwarded to the statechart of the receiving actor, making it easier to react.

1.2 Concrete Syntax

The concrete syntax of the SCCDXML formalism is based on SCXML, which is a Statecharts variant developed by the W3C and is described in a working draft specification [BAA+13]. SCXML specifies a textual syntax for Statecharts based on eXtensible Markup Language (XML). SCCDXML will adopt most of the SCXML specification and combine it with a simple XML representation to describe class diagram constructs.

We will first collaborate on our motivation for using (SC)XML in subsection 1.2.1. We will then look into the syntactic structures and corresponding XML tags, together with the available attributes and children. We do this for the class diagram and statechart structures seperately in subsection 1.2.2 and subsection 1.2.3 respectively. Next the different executable components available are discussed in subsection 1.2.4. This is then followed by an overview of the different macros in subsection 1.2.5 and an explanation of the syntax for state-referencing in subsection 1.2.6. Note that when talking about the context component (e.g. context statechart, context state) in the explanation of a certain tag, it refers to the component that certain tag belongs to.

1.2.1 Motivation for using XML

There are several reasons why XML has been used for the SCCDXML formalism. First of all we wanted to base ourselves upon some existing format for the concrete syntax instead of creating an entirely new one. Since SCXML[BAA+13] is seen as a standard, this gives us proof of its usability to model state charts.

Next, XML documents are human-readable and human-editable, which means that complex tooling is not needed to develop these. Any text editor can be used to create XML files, and while graphical environments could improve the workflow of designing new models, they are definitely not necessary. Furthermore, XML documents are plain text files and thus can easily be managed using an off-the-shelf version-control system, such as Git or Mercurial.

Also most of the widely used programming languages provide libraries to handle XML documents, thus easing the implementation of tools to handle the formalism. This is not only beneficial for interpreting XML documents used as input for a compiler, but it also greatly facilitates the generation of models made with a visual editor.

Finally, standards are available for querying and transforming XML documents, as well as open-source implementations of these standards. For one there is *XPath* [CD99], a W3C standard which describes a notation for navigating the hierarchical structure of XML documents. Similar to this standard we define a method for referencing states in subsection 1.2.6 and a way of traversing the class hierarchy through associations in subsubsection 2.2.1. Likewise, the W3C standard *XSLT* [Cla99] provides a language for transforming XML documents, which could be used to implement model transformations. The latter however is out of scope for this paper.

1.2.2 Class Diagram

<diagram>

The outermost container element *<diagram>* represents a class diagram and should only occur once in a model. It has two optional attributes *name* and *author*, both giving straightforward extra information about the diagram. The children nodes allowed for a diagram are :

- *<description>* : An optional child element enclosing the description of the model.
- *<class>* : Defines a class as part of the diagram which should at least occurs once.
- *<inport>* and *<outport>* : Respectively defines an input port or output port for the diagram. A port has as mandatory attribute *name*, which should uniquely identify the port. The ports can occur zero

or more times.

- *<top>* : An optional section for imports and definitions that need to be at the top of the generated document and can be referenced anywhere in the diagram.

<**class**>

The *<class>* tag defines a class as part of a class diagram. It has a single flag attribute *default* which defines whether the class is the default class of the diagram or not. As a logical consequence one and only one *<class>* in a single *<diagram>* should have a positive evaluating *default* attribute unless *<diagram>* contains only one *<class>* child. If the latter is the case then the single *<class>* will be used as default. The default class will be instantiated once when the diagram is created. The different children available for a class are *<relationships>*, *<method>*, *<attribute>* and *<scxml>*. The latter represents the dynamic behaviour of the containing class in the form of a statechart.

<**method**>

The *<method>* tag defines a method as part of a class. The mandatory attributes for a method are *name* and *type*, defining the identifier and the return type respectively. A method with the same name as the containing *<class>* will be considered a constructor, the same counts for a destructor where the class name is prepended with a $\sim$. In case of a constructor or destructor, the *type* attribute is ignored. The optional *access* attribute defines the access level of the method with as default value *public*. A method needs to have a *<body>* child containing the code that should be evaluated when the method is called. For proper functioning this code should match the target language of the compiler. Optional formal parameters can be defined with the *<parameter>* tag and have the following attributes :

- *name* : The identifier of the parameter.
- *type* : The type of the parameter.
- *default* : The optional default value for the parameter.

<**attribute**>

The *<attribute>* tag defines a member attribute of a class. The mandatory XML attributes for the element are *name* and *type*, which can be supplemented with the optional *init-value* attribute that defines an initial value of the attribute. Attributes of an instance can be accessed from within that instance using the *SELF* macro. Macros are explained in detail in subsection 1.2.5.

<**relationships**>

To define the different relations between the different classes in the diagram the *<relationships>* tag should be used. To set up an association from one class with another (note the uni-direction of the association) the *<association>* tag should be used inside the source class with the target class set as value for the *class* attribute. Both the minimum and maximum cardinality of the association can be set using the *min* and *max* attributes respectively. The minimum defaults to zero while the maximum has *N* as it's default value meaning that any number of instantiations of this association is possible. A name for the association should be defined using *name* attribute. This name will be used when accessing associated instances as explained in subsubsection 2.2.1.

The <*inheritance*> tag defines a super class for the context class, from which it inherits. The class name of the super class should be set as the *class* attribute and a priority can be set using the *priority* attribute. In case of multiple inheritance the super class with highest priority will be inherited from first. Naturally this *priority* attribute is only available for target languages that support multiple inheritance.

1.2.3 Statechart

In this subsection we zoom in on the elements that construct statecharts. They are illustrated in Code Sample 1.1 to Code Sample 1.4 as the SCCDXML variants of the examples we saw in subsection 1.1.1. Note that we skip the diagram information and thus the resulting XML segments aren't SCCDXML compliant on its own, they should be added as child of a <*class*> element as we saw in subsection 1.2.2.

<scxml>

The <*scxml*> tag defines (the root of) a statechart and is simply a top level composite state sharing the allowed children and attributes of the <*state*> discussed below. Only the *id* attribute is omitted for the root element.

<state>

The <*state*> tag simply represents a state. This can either be a basic state or a composite state depending on its position in the hierarchy. Each state should have an *id* that's unique at sibling level so that each state in the statechart can be uniquely accessed XPath-like using a sequence of state id's (see subsection 1.2.6 for a detailed explanation). For composite states, meaning that the <*state*> encloses other states, two other attributes are available.

First there is *initial*, which should be set to the identifier of the initial child state. Second there is the *conflict* attribute, defining how a transition conflict should be resolved. The possible values for this attribute are :

- *inner* : The inner-most enabled transition will be fired.
- *outer* : The outer-most enabled transition will be fired.
- *inherit* : The *conflict* value of the parent will be used. Consequently this is an invalid value for the root node.

The attribute defaults to *inherit* unless the context state is the *root*, then *outer* will be used. As this setting isn't available in most formalisms based on Statecharts (including SCXML), we'll illustrate the behaviour of the attribute using Figure 1.6.

When state *A* is active and an event *i* is cast, both the transitions to B and Z become enabled but only one transition is allowed to be fired. Since randomly picking a transition would result in indeterminism, either consistent behaviour could be enforced (which restricts the possibilities) or letting the modeller define what

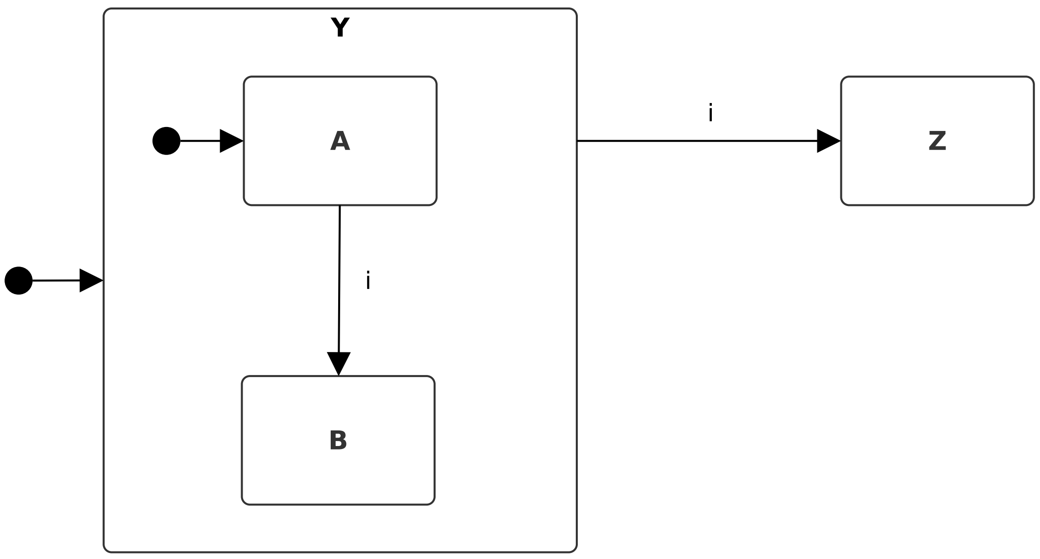

Figure 1.6: Illustrating the use of the *conflict* attribute of a state. When state *A* is active and event *i* is cast, the attribute will decide which transition needs to be fired.

should happen in such a case. Preferring the latter, the *conflict* attribute has been added. If in the example that attribute of the root element would be set to *inner*, the transition to B would be fired. In case of the default *outer*, the transition to Z would be fired.

The optional <*onentry*> and <*onexit*> children of a state hold executable content to be run upon respectively entering and exiting the state. The elements that count as executable are discussed in subsection 1.2.4. Other allowed children for a state are <*state*>, <*parallel*>, <*history*> and <*transition*>.

```
<scxml initial="A">
  <state id="A">
    <transition event="c" cond="P" target="../B">
      <raise event="d"/>
    </transition>
  </state>
  <state id="B"/>
</scxml>
```

Code Sample 1.1: The SCCDXML variant of Figure 1.1

```
<scxml initial="A">
  <state id="A">
    <transition event="h" target="../B"/>
  </state>
  <state id="B" initial="C">
    <state id="C">
      <transition event="k" target="../D"/>
    </state>
    <state id="D"/>
    <transition event="j" target="../A"/>
  </state>
</scxml>
```

Code Sample 1.2: The SCCDXML variant of Figure 1.2

<**parallel**>

The <*parallel*> tag models parallel states and has the same children and attributes as the <*state*> node, except for the *initial* attribute. This is logical since all substates become active, instead of just one, upon entering the context state.

```
<scxml>
  <parallel id="Y">
    <state id="A" initial="K">
      <state id="K">
        <transition event="x" target="../L"/>
      </state>
      <state id="L"/>
    </state>
    <state id="B" initial="D">
      <state id="D">
        <transition event="z" target="."/>
      </state>
    </state>
  </parallel>
</scxml>
```

Code Sample 1.3: The SCCDXML variant of Figure 1.3

<**history**>

The <*history*> tag, representing a history state, only allows the <*onentry*> and <*onexit*> tags as children. Its attributes are limited to an *id* and a *type* attribute. The latter has as possible values *shallow* and *deep*, with *shallow* being the default one.

```
<scxml initial="A">
  <state id="A" initial="K">
    <state id="K">
      <transition event="x" target="../L"/>
    </state>
    <state id="L"/>
    <transition event="y" target="../M"/>
    <history id="H"/>
  </state>
  <state id="M">
    <transition event="z" target="../A/H"/>
  </state>
</scxml>
```

Code Sample 1.4: The SCCDXML variant of Figure 1.4

<transition>

A *<transition>* tag defines a transition originating from its enclosing state. Its *event* attribute defines which event should trigger the transition and the *cond* attribute is a boolean expression that guards the transition. The *target* attribute defines the target state(s) of the transition; the syntax for this is described in subsection 1.2.6. To accommodate for input events sent through ports from the diagram, a special attribute named *port* is available. When set, only matching events from the specified port can enable the transition. For a transition to fire after a number of seconds, the *after* attribute should specify a expression that evaluates to a numeric value. The *after* attribute can not be combined with any of the other attributes except *target*, which is the only mandatory attribute.

The children of a transition exist out of the different available executable components (as listed in the next section) that should be executed when the transition is fired and out of formal event parameters. These parameters are formed with a *<parameter>* tag and have the same attributes as the formal parameters of a method *i.e., name*, *type* and *default*. When the transition is fired these formal parameters take the value of the actual parameters supplied by the trigger event. These formal parameters can then be referenced in the guard and action code using their identifiers as local variables.

1.2.4 Executable Content

Executable content is a series of zero or more executable components wrapped in a containing element like *<onentry>* or *<transition>*. They are processed in document order. The main executable component available is the *<script>* tag. Any text contained by it, will be interpreted as code of the target language to which the compiler is set to. It's possible to obtain the behaviour of all the other executable components by solely using the script component.

Text contained by a *<log>* component will simply be outputted to a console. The *<assign>* component assigns a value to either a local variable, or to a member value if the *SELF* macro is used for referencing. The value is obtained by evaluating the expression of the *expr* attribute, while the *ident* attribute indicates how to access the location to which the value needs to be assigned.

The *<raise>* tag allows for event generation. Its only mandatory attribute *event* defines the name of the event to be cast. As for the optional attributes, not all possible combinations are valid/allowed. The scope of the event, *i.e.,* to which instances the event will be visible, dictates which combinations are valid and it can be explicitly set by using the *scope* attribute. It's possible values are :

- *local* : The event will only be visible for the context statechart.
- *broad* : The event is broad-casted and thus visible for all instances.
- *output* : The event will be send to an output port and is only valid in combination with the *output* attribute.
- *narrow* : The event will be narrow-casted, *i.e.,* sent to certain specified instances only, and is only valid in combination with the *target* attribute.
- *cd* : The event will be processed by the object manager. See subsubsection 2.2.1.

If the *scope* attribute is not explicitly set, it will get a value depending on which of the other optional attributes are set. If none of these has a value, the scope of the event will be local. Having a value for *port*, which defines the name of the output port to which the event should be send, will result in an output scope. Similarly the presence of the *target* attribute will enforce a narrow scope. This last attribute takes a string as value that specifies which association links should be followed to find the target instance(s). This is explained in detail in subsubsection 2.2.1.

1.2.5 Macros

Macros are simple strings, usually capitalized, that will be substituted by an implementation depending on the target language during compilation. Not all macros can be used in every environment and will simply not be replaced if they are used incorrectly. Currently available macros :

- *SELF* : This macro is used to refer to the context instance. To access a member of that instance, *SELF* should be followed by a dot and the identifier of the wanted member. If a reference to the instance is needed, simply providing *SELF* will suffice.
- *INSTATE(x)* : The macro will evaluate to *true* if the state *x* is currently active. How to reference a state is explained in subsection 1.2.6.

1.2.6 State Referencing

For the referencing of one or more states, as needed for example to specify the target of a transition, we created a syntax similar to XPath. To access a certain state, a location path should be formed that ends in the wanted state. This location path is a string consisting of state identifiers separated by forward slashes ('/'), they present the hierarchy to be traversed before reaching the end state.

There is the option to use either a relative path or an absolute path. The latter will start traversing the hierarchy from the *root* and is denoted by prepending the path with a forward slash. The relative path will just start traversing the hierarchy from the context state. To go a level up in the hierarchy (i.e. access the parent of a certain state) two dots ('../') should be used instead of a state identifier. One single dot ('.') will always refer to the current state in the path.

Specifying multiple target states can be done either by a comma separated list of state references, the use of branching, or a combination of both. Branching is done by using brackets to define a common source location path. We illustrate this with the following example :

```
1) A/B/C, A/B/D, A/B/E
2) A/B(C,D), A/B/E
3) A/B(C,D,E)
```

All three expressions denote the same three resulting states. For the third expression we start at the context state where the child with id *A* is selected. This is then followed by going down the hierarchy and looking for a child of *A* with id B. We then encounter a left bracket, meaning that we branch off from the selected *B* node. Enclosed by the left bracket and an ending right bracket, we find a comma separated list of expressions. These expressions are now parsed to obtain (a subset) of the resulting states, using *B* as starting point instead of the original context state. This basically means that our result set will contain the children of *B* with IDs *C*, *D* and *E*.

Note that more complex expressions can be used inside the branches and even nested branching is supported. This can result in more complex expressions as shown below together with its branchless equivalent :

```
1) A/B/C(D/E,F(G,H)), I/J
2) A/B/C/D/E, A/B/C/F/G, A/B/C/F/H, I/J
```

2
Compiler

In this chapter we describe how a SCCDXML model gets transformed into executable code. A compiler library has been developed (in both Python and C#) that can either be used programmatically or through a command line application. The first phase of compilation is transforming the provided SCCDXML model into a abstract syntax structure as discussed in section 2.1. By deploying the visitor pattern, this is then followed by multiple visits that decorate the abstract syntax structure to facilitate code generation. We look into these visits and their corresponding visitors in section 2.3.

The arguments for the compile operation include the path to the file to be compiled and the path to the location where the generated file should be stored. Besides that the target language should be specified, as well as which run-time platform the code should run on. The different platforms and their architecture are described in section 2.2.

2.1 Architecture

The first step in the compilation process is creating an abstract data structure from the input SCCDXML model. The (simplified) class diagram of this structure can be seen in Figure 2.1. At most levels a class in this diagram matches an XML tag in the model. The model gets parsed using a XML library and subsequently the data structure gets created starting from the root. The XML root element gets transformed into a *ClassDiagram* instance, its <*class*> children into *Class* instances, <*method*>s into *Method* instances, and so on.

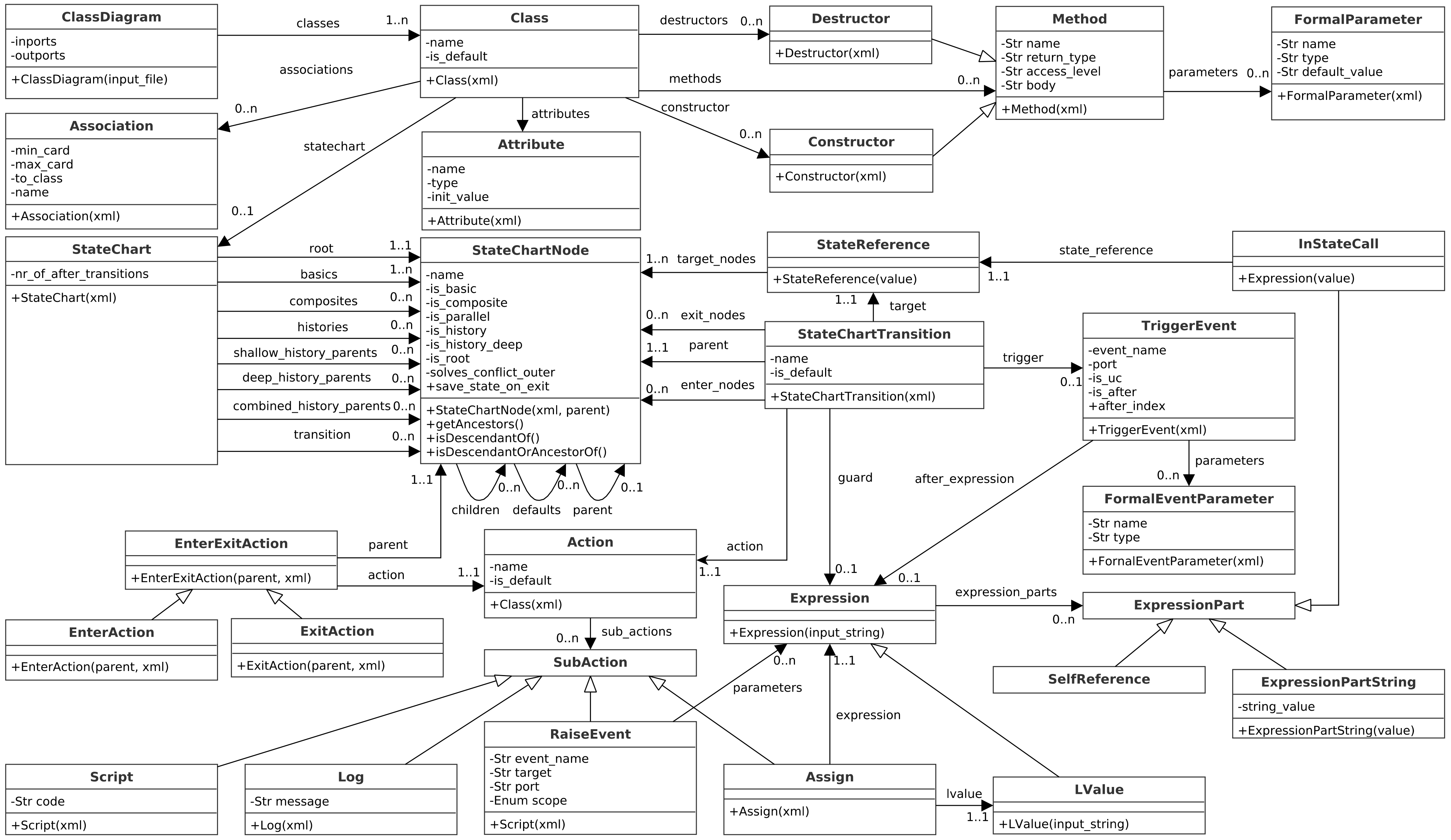

Figure 2.1: Class diagram illustrating the abstract data structure created from an SCCDXML model.

2.2 Runtime Platform

The execution/simulation of SCCDXML models is similar for every single model. To avoid the unnecessary generation and compilation of the duplicated execution code, a runtime library is created that combines all shared code. Multiple execution/simulation methods can be supported by this library, which we will call *runtime platforms*. The wanted execution method should be supplied as parameter to the compiler, which will then generate the correct calls to the matching platform together with the model-specific code.

In subsection 2.2.1 we first look at the entities that are shared by all platforms. We then look at the two runtime platforms supported by our runtime library, *event simulation* and *game loop simulation*, in subsection 2.2.2 and subsection 2.2.3 respectively.

2.2.1 Architecture

The runtime library mainly consists out of base classes that should be inherited by the platform specific classes and by the generated classes. This specific architecture facilitates the addition of new platforms and thus well known simulation methods like headless simulation (as fast as possible) or GUI interleaved execution (using the timing system of the relevant GUI library) can be added easily in the future.

Event Queue

Most of the constructs in the library need to be able to store events and handle them appropriately at the correct time. For this reason a queue-like data structure was created that stores a sequence of events; each event is paired with its remaining time until it should be processed. The available methods on this structure are :

- *add(event, time)* : Pushes an event and its remaining time in the queue.
- *decreaseTime(offset)* : Lowers all the remaining times with the offset value.
- *getEarliestTime()* : Returns the lowest time value present in the queue. If the queue is empty, positive infinity is returned.
- *popDueEvents()* : Returns a list of all the events of which the remaining time is lower than zero. These events are removed from the structure.
- *isEmpty()* : A simple check whether the data structure contains events or not.

Runtime Class

Every class defined in the model gets compiled into a class that inherits from one shared runtime base class. This class contains a series of methods and members that are key for a correct execution of the model. Every instance has its own event queue containing the events that are local to its statechart. Next to that, this class is also responsible for managing the timers for transitions. At compile time, every timed transition is assigned a unique integer that will be used as index for a dictionary that map these transitions to their remaining time. An index of a transition is added to this dictionary upon entering the state from which this transition originates. This basically initializes the timer with the initial value supplied by the *after* attribute. Upon leaving the state, the indexes are removed again from the dictionary.

The *getEarliestEventTime()* method returns the remaining time until the instance needs attention. This is calculated by taking the minimum time from both the event queue and the timer dictionary. Having a zero or negative earliest time is normally followed by a call to the *step()* method (but this is the responsibility of the object manager). In Code Sample 2.1 we see the pseudo code for the step method.

```
public void step(delta){
  if (!this.active)
    return; //Simulation of statechart not started yet

  this.events.decreaseTime(delta); //Decrease the remaining time of all events

  if (this.timers.Count > 0) //Check if there are timers initialized
  {
    next_timers = new Dictionary(); //The dictionary to replace the current one
    foreach ((key,time) in this.timers) //Iterate over all timers
    {
          new_time = time - delta
      if (new_time <= 0.0) //Timer value goes below 0
        this.addEvent(new Event("_" + key + "after")); // transform to event
      else
        next_timers[key] = new_time; //Transition not due
    }
    this.timers = next_timers; //Set new dictionary
  }

  this.microstep();
  while (this.state_changed)
    this.microstep();
}
```

Code Sample 2.1: Pseudo code for RuntimeClassBase's step() method

The method requires one *delta* parameter, which should be the time that has passed since the last call to this instance's *step()* method. As can be seen in the code, meaningful changes are only performed when the instance has already been activated, which is done by calling the virtual *start()* method. In case execution has started, the *delta* parameter is used to update the event times by calling *decreaseTime()* on its event queue. A similar action is then performed on the timers dictionary. The timers are first checked to see if any of them has gone below zero, this evaluation happens using the stored value subtracted by the *delta* value. If a timed transition is due, a new event is added to the queue based on that timer's index which will be used in the generated code to trigger the actual transition.

```
private void microstep(){
  due_events = this.events.popDueEvents();
  if (due.Count == 0)
    this.transition ();
  else
    foreach (event in due)
      this.transition(event);
}
```

Code Sample 2.2: Pseudo code for RuntimeClassBase's microstep() method

The *step()* method ends with repeated calls to *microstep()* until that no more results in changes to the state. In Code Sample 2.2 we see that all the events that are due are first gathered. If at least one event is due, the abstract *transition()* method is called for every event. This method will be implemented by the generated classes. If no events are due, the *transition()* method is still called (but without arguments) as it is possible that unconditional transitions have been enabled that can still affect the state.

Object Manager

The *object manager* is an always present entity responsible for managing the class diagram and coordinating all corresponding actions. This includes the creation and deletion of new instances and handling the communication between the different actors. Commanding the manager happens solely through events (the manager is an actor itself) but for some actions like narrow casting, this is abstracted away by the formalism. To directly message the manager, an event should be cast with *cd* as value for the *scope* attribute. The compiler will automatically set the first parameter of such an event to be a reference to the sender, so that the object manager can reply to the caller if needed. What follows is an overview of the different events the manager acts upon, their parameters, and the replies the manager returns as response to them.

create_instance Upon receiving this event the object manager will create a new instance, if the class diagram allows it. The first user-supplied parameter should be the name of the association for which a new instance should be created, followed by any constructor parameters. If creation succeeded a reply event *instance_created* will be send to the requester with as argument the name of the association for which an instance was created. The same argument is send with the *instance_creation_error* in case creation failed or was not allowed.

delete_instance The *delete_instance* event allows for the deletion of instances. The manager expects the supplied string argument to be an association reference (see subsubsection 2.2.1) referencing the instance(s) to be deleted. All relations with the deleted instance(s) are removed as well, which implies that the deletion needs to conform to the class diagram.

start_instance This event will start the execution of the statecharts of the instances referenced by the sole association reference argument.

associate_instance On creation of an instance, it's associated solely with its creator or with no instance at all in case of the default instance. The *associate_instance* event makes it possible to associate instances with multiple other instances. This event expects two association references as arguments, a source path and a target path. The instance(s) found processing the first reference, will be added to the association that results by processing the second. This implies that the second reference can't have an index specified for the last association name.

Since the object manager is the only entity that has access to all the present instances, it also plays an important role in the correct execution of the diagram. Two important methods are exposed for execution purposes, *stepAll*() and *getWaitTime*(). The latter simply returns the remaining time until the first upcoming event. This can be either an event send directly to the object manager or an event from any of the instances present in the class diagram. *stepAll*() simply calls its own *step*() method (which handles its own events), together with all the *step*() methods of the instances.

Association Structure

The object manager contains a dictionary that maps every runtime class instance to an associations wrapper. This wrapper links the mapped instance to its associations, using another dictionary that maps association names to an *Association* class. This *Association* class encapsulates minimum and maximum cardinality of the association and a list of the linked instances. This list, its size constrained by the cardinality, is used for traversing the association hierarchy. To do this we again created an XPath-like syntax to create association references. Such reference is a string consisting of association names, optionally followed by a square-bracketed index, separated by forward slashes ('/'). They present which associated instances should be traversed to reach the requested instance(s). The algorithm used to translate an association reference into the resulting instances is shown in Code Sample 2.3.

```
current_set = [source_instance];
foreach ((association_name,index) in association_reference) {
  next_set = [];
  foreach ( current in current_set ){
    association = current.getAssociation (association_name);
    if (index >= 0 ) //index was specified
      nexts.append( association.getInstance(index)); //Only add the instance with
   matching index
    else if (index == -1) //index was not specified
      nexts.extend ( association.getAllInstances() ); //Add all instances of this
   association
    else
      throw new AssociationReferenceException("Incorrect index in association
   reference.");
  }
  currents = nexts;
}
```

```
return currents;
```

Code Sample 2.3: Pseudo code for the processing of an association reference

The index after an association name should only be specified if just a single instance is targeted. If omitted, all instances of the association will be added to the current set.

Controller

The controller is the top level element and acts as an interface to the class diagram. It has access to all the instances that make up the diagram through the object manager and takes care of the execution of the state charts. If the default instance needs any constructor arguments, these should by supplied to the constructor of the controller. Each runtime platform is represented by a different controller class inheriting from a common base class. The interface of this base controller exists out of the following methods :

- $start()$: Starts the execution of the class diagram and thus of all present statecharts.
- $stop()$: Ends the execution of the class diagram.
- $addInput(event\_name, port[, parameters, time])$: Adds an input event to the diagram with the specified arguments. Note that the supplied time is relative.
- $addInputEvent(event[, time])$: Adds an already created event with optional relative time.
- $addEventList(event\_list)$: Inputs a list of events.
- $addOutputListener(ports)$: Creates and adds an output listener to the diagram and then returns the reference.

OutputListener

Receiving output from a compiled diagram is done by attaching output listeners to the correct ports. By calling the $addOutputListener()$ method on the controller with an array of ports, a listener gets created and returned. The diagram will attach all output for the listed ports to this listener and thus output can be easily gathered by calling the *fetch()* method on it. The controller is responsible for creating the correct type of output listener by overloading the protected $createOutputListener()$ method of $ControllerBase$. For instance, when using the *event-simulation* platform, listeners of type $ConcurrentOutputListener$ will be created which are thread-safe.

2.2.2 Event Simulation

The run-time platform that approximates real-time simulation the best is the *event-simulation* platform. In Figure 2.2 we see that an event gets processed the moment it gets cast. The platform implements this by running the complete diagram in a separate thread allowing accurate timing of the actions. $ThreadsControllerBase$, the controller class representing the platform, overrides the $start()$ method to start the thread. This thread runs the method depicted in Code Sample 2.4. Most of the timing functionality occurs in the $handleWaiting()$ method. It calculates how long it takes until the next planned action, using its input queue and the $getWaitTime()$ of the object manager, and blocks the thread accordingly.

```
private void run(){
  this.last_recorded_time = now();
  base.start();
  last_iteration_time = 0.0;
  while (true)
  {
    this.handleInput(last_iteration_time);
    this.object_manager.stepAll(last_iteration_time); //Compute the new state based
   on internal events
    this.handleWaiting(); //Blocks the thread until action is required
    acquire();
    if (this.stop_thread)
      break;
    release();
    previous_recorded_time = this.last_recorded_time;
    this.last_recorded_time = now();
    last_iteration_time = this.last_recorded_time - previous_recorded_time;
  }
}
```

Code Sample 2.4: Pseudo code for ThreadsControllerBase's run() method

ThreadsControllerBase overrides the controller's input methods to be thread-safe and to notify the thread of new input. When the thread receives such notification or when the thread-blocking times out, it continues executing and calculates how much time has passed. This time is then used to handle the input and call the *stepAll*() method of the object manager. To properly handle the thread, a *join*() method is added to the controller which blocks the calling thread until the diagram's thread terminates.

2.2.3 Game Loop

The goal of the game platform is to allow easy integration of the generated code with games and game engines that make use of a game loop for execution. The platform adds an extra method *update*() to the controller, which should be called every frame of the loop with as sole argument the time that has passed since the last call. This way the controller is able to execute the actions that should have happened during the last frame, resulting in a decent approximation of continuous time. The method's pseudocode is shown in Code Sample 2.5, and Figure 2.2 illustrates the delayed processing of an event at the beginning of the next frame. The platform supports both fixed and variable time steps.

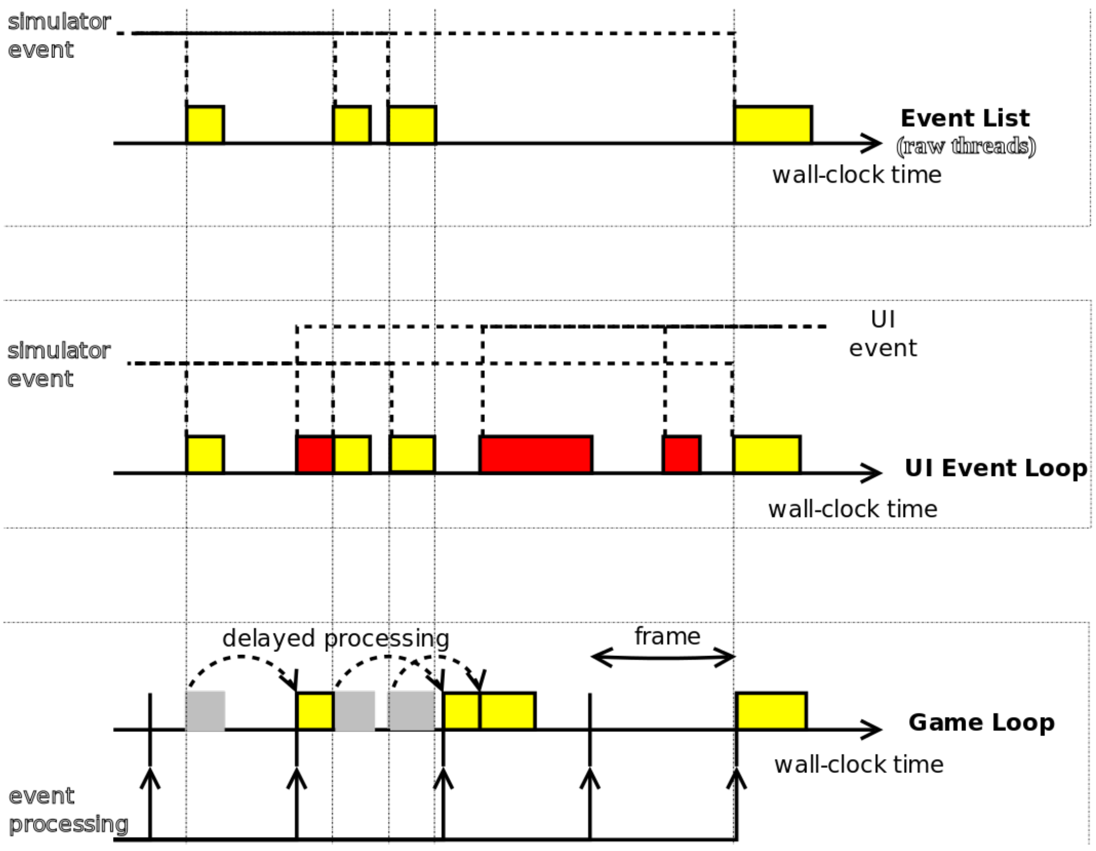

Figure 2.2: Diagram illustrating the different simulation methods.

```
public void update(delta){
  this.input_queue.decreaseTime(delta);
  foreach(event in this.input_queue.popDueEvents())
    this.broadcast(event);
  this.object_manager.stepAll(delta);
}
```

Code Sample 2.5: Pseudo code for GameControllerBase's update() method

2.3 Code Generation

Our code generation makes highly use of the visitor pattern. This design pattern is a way of separating our algorithms from the main datastructure. A practical result of this separation is the ability to add new visitors (for instance one for generating a new target language) without having to modify any of the structure classes. First the initial datastructure gets created as described in section 2.1 by parsing the XML model. Separate visitors then decorate the structure with extra information with as ultimate goal having code generated by the last visitor. All classes that are part of the structure inherit from *Visitable*, which adds an $accept()$ method to the classes' interface to implement double dispatch. This method expects an instance of type *Visitor* as parameter, which should implement $visit()$ methods for each class in the structure it wants to act upon. Finally the *accept*() method will then call the supplied visitor's *visit*() method with its own instance as argument, and polymorphism will select the correct implementation. In the following sections we will go over the different visitors currently present in the implementation of the compiler.

2.3.1 State Linker

The task of the *StateLinker* visitor is to replace any state reference by the *StateChartNode* instance(s) it targets. The AST is traversed in an optimal way to rapidly find all instances of a *StateReference*. The corresponding expressions are first parsed into tokens that are meaningful for our syntax as described in subsection 1.2.6. Next, it takes the context node as current node and starts iterating over the tokens, taking actions accordingly. Finally, the resulting nodes are then added to the *StateReference* instance so that it can be easily used by the following visitors.

2.3.2 Path Calculator

The *PathCalculator* calculates for each transition which sequence of nodes should be exited and entered. An important element in this calculation is the Least Common Ancestor (LCA). The LCA of a set of nodes is the node *n* such that *n* is a proper ancestor of all nodes in the set and no descendant of *n* has this property. Note that there is guaranteed to be such an element since the root is a common ancestor of all states. Note also that since we are speaking of proper ancestor (parent or parent of a parent, etc.) the LCA is never a member of the provided set of nodes. In Code Sample 2.6 we see the method calculating the LCA of a transition, using its source- and target-nodes as input set.

```
calculateLCA(transition){
  //Iterate over all the ancestors of the source node, starting at the bottom
  foreach(anc in transition.parent.getAncestors()){
    all_descendants = true;
    //Check for each target node if it is a descendant of the current ancestor
    foreach (node in transition.target.target_nodes){
      if (!node.isDescendantOf(anc)){
        all_descendants = false;
        break;
      }
    }
    //The first ancestor all target nodes descend of, is the LCA
    if (all_descendants)
      return anc;
  }
  //Since every node has the root as ancestor, this statement is never reached.
  return null;
}
```

Code Sample 2.6: Pseudo code for calculating the Least Common Ancestor (LCA)

This method is then used in the *StateChartTransition* visit depicted in Code Sample 2.7. Determining which exit actions should be executed for a transition is fairly easy once the LCA is known. It's as simple as executing the exit actions of the ancestors up to (but not included) the LCA. Calculating the sequence of enter actions is a bit more complex as a transition can have multiple target nodes. We start off with an empty result set that will hold the sequence of nodes to be entered. Then for each target node, we iterate over its ancestors until we either reach the LCA or a node that is already part of the result sequence. We then add these in reversed order to the result sequence and move on to the next target node. Nodes are added to the result together with a boolean value which specifies whether it's a target node or not. This is used in code generation to determine whether default child states should be entered or not.

```
visit(StateChartTransition transition){
  //Find the scope of the transition (lowest common proper ancestor)
  LCA = this.calculateLCA(transition);
  //Calculate exit nodes
  transition.exit_nodes = [transition.parent];
  foreach(node in transition.parent.getAncestors()){
    if (node == LCA) break;
    transition.exit_nodes.Add(node);
  }
  //Calculate enter nodes
  transition.enter_nodes = [];
  //We iterate over all target nodes and add the nodes to enter accordingly
  foreach (target_node in transition.target.target_nodes){
    to_append = [(target_node,true)];
    foreach (anc in target_node.getAncestors()){
      //If we reach the LCA in the ancestor hierarchy we break
      if (LCA == ancestors[ancesto_index])
        break
      //boolean value to see if the current ancestor should be added to the result :
      to_add = true
      //If we reach an ancestor that is already listed as enter node,
      //we don't add, and break :
      foreach ((enter_node, is_end_node) in transition.enter_nodes){
        if (enter_node == ancestors[ancestor_index]){
          to_add = false;
          break;
        }
      }
      if (to_add)
        to_append.Add((ancestors[ancestor_index], false));
      else
        break;
    }
    to_append.Reverse(); //Reversed order for enter hierarchy
    transition.enter_nodes.AddRange(to_append);
  }
}
```

Code Sample 2.7: Pseudo code for PathCalculator's visit method for StateChartTransitions

2.3.3 Code Generator

The last visitor is responsible for generating actual code. For each target language, a different visitor should be created which traverses the completely decorated structure in a top-down order. The generated code consists out of classes that inherit from the entities present in the matching run-time platform. Each *Class* will first generate a unique integer (ID) for every node in its statechart in the form of an enumeration. The current state is then maintained using a dictionary that maps these IDs to a list of IDs of its children that are currently active. If at least one history state is present in the statechart, then a similar structure is maintained to save the history snapshots of compound states.

The resulting code contains a transition method, accepting an event as argument, for each node in the statechart. When an event occurs, the root's transition method is called first with that event as parameter. The root's transition method will then forward the event to its active children, which will in turn do the same unless they consume the event. Note that the forwarding of the event depends on which conflict resolving method is used. To illustrate this we look at the pseudo code of the transition method of a composite state in Code Sample 2.8 where the *conflict* attribute is set to *outer*.

```
transition_Root_state1(event){
  catched = false; //Boolean value denoting whether the event gets consumed
  enableds = []; //List of enabled transition IDs
  if (event.getName() == "event_x" && event.getPort() == "port_x"){
    enableds.Add(0);
  }
  if (event.getName() == "event_y" && event.getPort() == "port_y"){
    enableds.Add(1);
  }
  if (enableds.Count > 1){
    //Log warning that indeterminism is detected.
    //Only the first in document order enabled transition will be executed.
  }
  if (enableds.Count > 0){
    enabled = enableds[0];//Transition ID to be executed
    if (enabled == 0){
      //Fire transition with ID 0
      this.exit_Root_state1();
      //Action code
      this.enter_Root_state2();
    }
    else if (enabled == 1){
      //Fire transition with ID 1
      this.exit_Root_state1();
      //Action code
      this.enter_Root_state3();
    }
    //A transition was fired so we can set :
    catched = true;
  }
  if (!catched){
    //The event was not consumed so forward to active children
    if (this.current_state[Node.Root_state1][0] == Node.Root_state1_a){
      catched = this.transition_Root_state1_a(event);
    } else if (this.current_state[Node.Root_state1][0] == Node.Root_state1_b){
      catched = this.transition_Root_state1_b(event);
    }
  }
  return catched;
}
```

Code Sample 2.8: Pseudo code for the transition method of a composite state with *outer* conflict resolving

We see that each transition originating from the context node gets assigned a numeric ID. The method first checks for each transition whether they are enabled or not, if so its ID is added to the list *enableds*. If multiple transitions get enabled we warn the user of the detected indeterminism and only fire the first enabled one. In the example we clearly see that before the action code of the enabled transition is executed, a call is made to the exit method of the context node. This method will first execute the user-defined exit actions followed by removing the node from the current state and, in case of a history state, make a snapshot of the state of its children. Only then the action code is executed and a call is made to the enter method. Note that it's possible that multiple exit and enter methods can be called when traversing multiple layers of nodes.

If a transition got fired we set the boolean variable *catched* to *true*. We use this value to determine whether or not we should forward the event to the active children. In case of a negative value we consult the *current_state* dictionary to find the active substate and forward the event accordingly. Checking the current state however is unnecessary for parallel nodes, as is the case in Code Sample 2.9, since all children are active when the parent is active. Also different in this example is that the *conflict* attribute is set to *inner* causing the event to be send to the children first, before checking if the event enables any of the transitions. If (at least one) of the children consumes the event, the transitions at this node aren't considered any more. This is done by returning the value of *catched* to the caller.

```
transition_Root_state1(event){
  catched = false; //Boolean value denoting whether event gets consumed
  //Conflict resolving set to "inner", first check children.
  catched = this.transition_Root_state1_a(event) || catched;
  catched = this.transition_Root_state1_b(event) || catched;
  if (!catched) {
    enableds = [];
    if (event.getName() == "event_x" && event.getPort() == "port_x"){
      enableds.Add(0);
    }
    ...
    if (enableds.Count > 0){
      enabled = enableds[0];
      if (enabled == 0){
        this.exit_Root_state1();
        this.enter_Root_state2();
      }
      catched = true;
    }
  }
  return catched;
}
```

Code Sample 2.9: Pseudo code for the transition method of a parallel state with *inner* conflict resolving

While each *Class* can have multiple constructors defined for certain target languages, a part of the construction will always be the same in order to initialize the different structures. For this reason a *commonConstructor()* is added which will be called first by every (user-defined) constructor. Also, the *start()* method of the runtime classes gets overridden to first call the base behaviour followed by entering the default states and executing the corresponding enter actions.

3 Test Framework

3.1 Requirements

In order to efficiently and correctly verify the functionality of our compiler, the tests and testing framework need to adhere to the following requirements :

1. Automated : We want to be able to (re)run the tests with as little effort as possible.
2. Isolated : A single test needs to be unaffected by the presence, absence, or results of other tests.
3. Easy to write : Writing a test should be easy and cost little effort.
4. Fast execution : Since the tests should be frequently rerun, they should take as little time as possible.
5. Multi-Target : The tests should be able to run on the different target platforms and languages without the need for modifications. This makes it easier to maintain current tests, add new tests, and add new target platforms and languages.

3.2 Architecture

The most optimal set-up of a test would consists out of three SCCDXML models: one model being the model under test (MUT), another generating input events for the MUT, and a last model to receive and act on the output of the MUT. Depending on which state the latter ends up in, the test either succeeds or fails.

However, since we are not sure whether the compiler is functioning correctly, we cannot trust the results of these models and thus this isn't a valid method of testing.

We chose for the tracing alternative where the test includes an input list and an expected output trace. The test suite then compares the actual output to the expected output and announces the result. Since the tests need to be platform and target language independent we include this information in the XML document representing the model to be tested. One (or multiple) <*test*> tags get added as a child of <*diagram*>. Since this is purely extra information the document can still be SCCDXML compliant and we can thus use the compiler without any problem. Only when inserted in the test framework, the test information holds meaning. Note that to maintain the target language independence the tests should not rely on language specific action code, and the test execution code should be ported to every target language.

Each <*test*> tag should contain maximum one <*input*> tag and one <*output*> tag, defining the input and the expected output respectively. The input consists out of a list of events specified with an <*event*> tag and the following attributes :

1. *name* : The name of the input event.
2. *port* : The input port that will be used for the event.
3. *time* : The time offset of the event. If multiple events have the same time offset, the events are added in document order.

The expected output works in a similar manner but has a notion of slots added. Since SCCDXML doesn't have a notion of simultaneous events, we can not know for sure (unless we look into the implementation) which event will be output first in the case two parallel sections cast an event in reaction to the same input event. With the slot structure we would place these two expected events in the same slot and then their order of appearance doesn't matter. As long as both events get generated the test will pass. We see an example of this in Code Sample 3.1. This specific test validates the functionality of a history state in a parallel element.

```
<diagram>
  <description>
    Testing history where the history state is directly inside a parallel element.
  </description>
  <inport name="test_input" />
  <outport name="test_output" />
  <class name="TestClass" default="true">
    <scxml initial="parallel">
      <parallel id="parallel">
        <state id="orthogonal_1" initial="orthogonal_inner_1">
          <state id="orthogonal_inner_1" initial="state_1">
            <state id="state_1">
              <onentry>
                <raise port="test_output" event="in_state_1" />
              </onentry>
              <transition port="test_input" event="to_state_2" target="../state_2"/>
            </state>
            <state id="state_2">
              <onentry>
                  <raise port="test_output" event="in_state_2" />
              </onentry>
            </state>
            <transition port="test_input" event="to_outer_1" target="../outer_1"/>
          </state>
          <state id="outer_1">
            <onentry>
              <raise port="test_output" event="in_outer_1" />
            </onentry>

          </state>
        </state>
        <state id="orthogonal_2" initial="orthogonal_inner_2">
          <state id="orthogonal_inner_2" initial="state_3">
            <state id="state_3">
              <onentry>
                <raise port="test_output" event="in_state_3" />
              </onentry>
              <transition port="test_input" event="to_state_4" target="../state_4"/>
            </state>
            <state id="state_4">
              <onentry>
                  <raise port="test_output" event="in_state_4" />
              </onentry>
            </state>
            <transition port="test_input" event="to_outer_2" target="../outer_2"/>
          </state>
          <state id="outer_2">
            <onentry>
              <raise port="test_output" event="in_outer_2" />
            </onentry>
          </state>
        </state>
        <history id="history_1" type="shallow">
        </history>
        <transition port="test_input" event="exit" target="../next_to_parallel"/>
      </parallel>
      <state id="next_to_parallel">
        <onentry>
          <raise port="test_output" event="outside" />
        </onentry>
```

```
        <transition port="test_input" event="to_history_1"
    target="../parallel/history_1"/>
      </state>
    </scxml>
  </class>
  <test>
    <input>
      <event name="to_outer_1" port="test_input" time="0.0"/>
      <event name="to_outer_2" port="test_input" time="0.0"/>
      <event name="exit" port="test_input" time="0.0"/>
      <event name="to_history_1" port="test_input" time="0.0"/>
    </input>
    <expected>
      <slot>
        <event name="in_state_1" port="test_output"/>
        <event name="in_state_3" port="test_output"/>
      </slot>
      <slot>
        <event name="in_outer_1" port="test_output"/>
      </slot>
      <slot>
        <event name="in_outer_2" port="test_output"/>
      </slot>
     <slot>
        <event name="outside" port="test_output"/>
      </slot>
      <slot>
        <event name="in_outer_1" port="test_output"/>
        <event name="in_outer_2" port="test_output"/>
      </slot>
    </expected>
  </test>
</diagram>
```

Code Sample 3.1: A model that tests the history functionality of a parallel node.

The tests were created in a bottom-up order, first we developed tests that validate each different construct separately and then we step by step merged the different constructs to see if they function properly together. There are also tests that simply validate the XML model. On the one hand valid models are supplied that should throw no errors, while on the other hand faulty models are inserted into the compiler that should generate a specific compiler exception. When the correct exception is caught, the tests is considered to be successful. For this we added an *exception* attribute to the <*test*> tag as depicted in Code Sample 3.2. This test includes a model that has two sibling states with the same ID; since this should not be allowed, the test expects a $CompilerException$.

```
<diagram>
  <description>
    Testing duplicate id's.
  </description>
  <class name="Test1">
    <scxml initial="state1">
      <state id="state1"/>
      <state id="state1"/>
    </scxml>
  </class>
  <test exception="CompilerException"/>
</diagram>
```

Code Sample 3.2: A model that tests the detection of duplicate IDs

4
Example Case : Tank Wars

To proof the usability of our formalism and the correctness of our compiler, a simple game environment has been created where a user-controlled tank has to fight one or more computer-controlled tanks. Both the artificial intelligence and the user-input handling in the game are modelled with the SCCDXML formalism and consequently compiled into executable code. We will see that the resulting code is stable and that the designs are simple but effective. The developed game has been called *Paper Warfare* (See Figure 4.1) and has rather complex controls where the body and the cannon of the tank have to be controlled separately. This makes not only the game-play more interesting but helps us in showing that handling fairly complex input is rather easy with the modelling formalism.

As our goal was to make the game resemble commercial games, a game-loop was used for the simulation of the game. For this reason the models are obviously compiled to run on the game-loop platform and have an input port *engine* which receives events from the game engine. The most important event here is the *update* event which is send for every frame. The input-handling model has an extra input port for the input events generated by the user, and an output port to send events to the game to update the GUI. Both the XML and visual representation of the models are included.

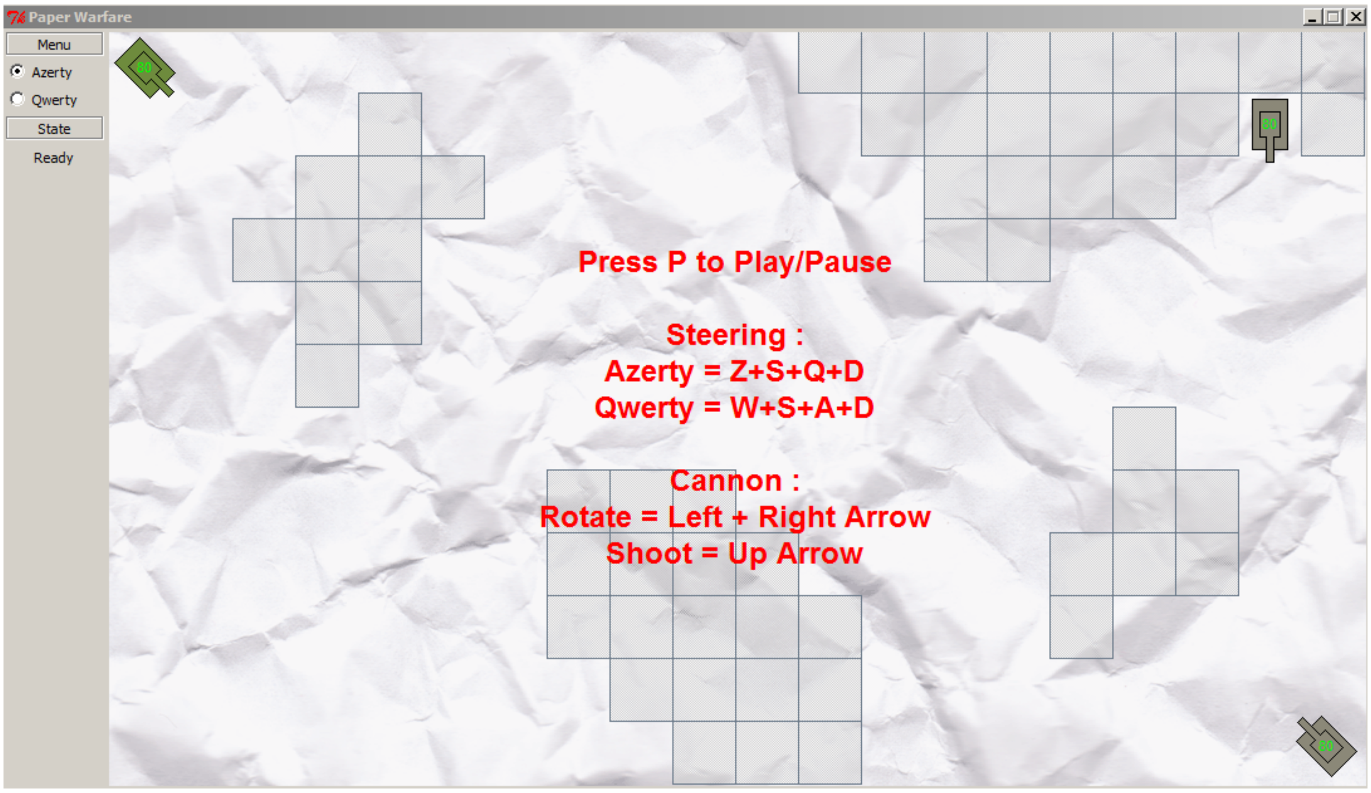

Figure 4.1: Our example game *Paper Warfare* where we demonstrate the usability of the SCCDXML formalism

4.1 User-Controlled Tank

To handle the user-input and the corresponding actions, the body and the cannon of the tank each have their own actor. A separate actor *Main* (as can be seen in Code Sample 4.1) is responsible for the initial instantiation of the other actors and setting up the needed associations.

```
<?xml version="1.0" ?>
<diagram author="Glenn De Jonghe" name="Player Tank">
  <description>
    Handling the player tank.
  </description>
  <inport name="engine" />
  <inport name="input" />
  <outport name="gui" />
  <class name="Main" default="true">
    <attribute name="tank" type="PlayerTank"/>
    <method name="Main">
      <parameter type="PlayerTank" name="tank"></parameter>
      <body>
        self.tank = tank
      </body>
    </method>
    <relationships>
      <association class="Cannon" name="cannon" min="1" max="1"/>
      <association class="Body" name="body" min="1" max="1"/>
    </relationships>
    <scxml initial="state_1">
      <state id="state_1">
        <transition target="../state_2">
          <raise event="create_instance" scope="CD">
            <parameter expr="'cannon'"/>
            <parameter expr="SELF.tank"/>
          </raise>
        </transition>
      </state>
      <state id="state_2">
        <transition event="instance_created" target="../state_3">
           <raise event="create_instance" scope="CD">
            <parameter expr="'body'"/>
            <parameter expr="SELF.tank"/>
          </raise>
        </transition>
      </state>
      <state id="state_3">
        <transition event="instance_created" target="../end">
          <raise event="start_instance" scope="CD">
            <parameter expr="'cannon'"/>
          </raise>
          <raise event="start_instance" scope="CD">
            <parameter expr="'body'"/>
          </raise>
        </transition>
      </state>
      <state id="end"/>
    </scxml>
  </class>
  ...
```

Code Sample 4.1: The *Main* component of the player-controlled tank model.

To have this actor instantiated on creation of the compiled diagrams' controller, the *default* attribute is set to *true*. Since *Main* has a constructor defined with parameters, these parameters should be supplied to the controller's constructor. In our case the sole parameter is *tank*, which should be an object representing the properties and state of the tank to be controlled. It gets assigned to the member attribute with the name *tank* which makes it accessible from anywhere in the class specification (including its state chart).

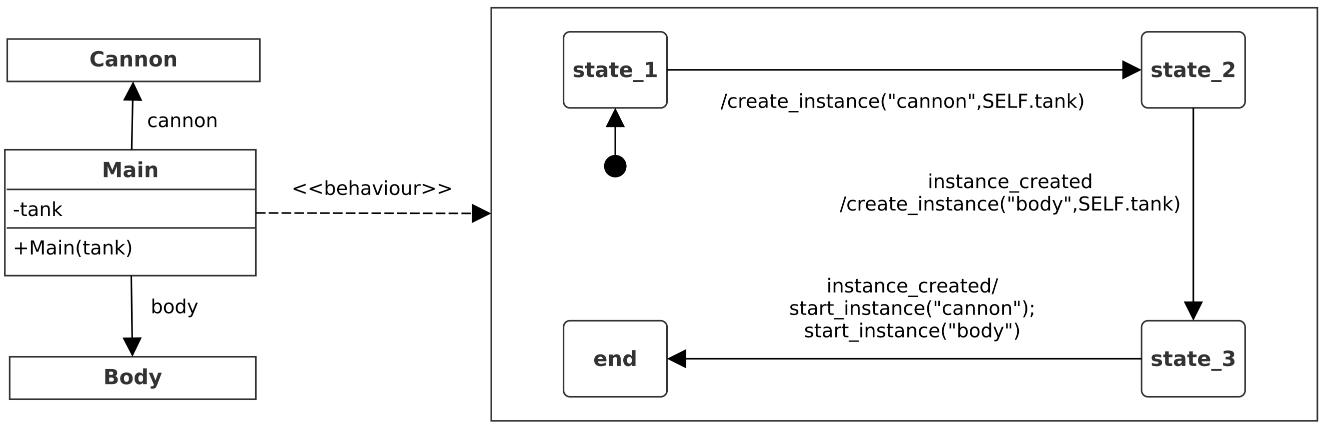

Figure 4.2: Visual representation of Code Sample 4.1

Enclosed by the *<relationships>* tag, we see two associations defined that will allow the instances to be created. When the *start()* method is called on the top-level controller, the model starts execution. *Main* starts in state *state_1* and immediately transitions to *state_2* casting a *create_instance* event to the object manager. The first parameter *cannon* indicates that an instance should be created for the association with the same name. The name indicates an association with *Cannon* which has a user-defined constructor with one parameter, and thus is this parameter also supplied to the *create_instance* event. In our example we just pass on the tank attribute so that the other actors have access to it as well.

In *state_2* an event *instance_created* is expected indicating that the instance is created successfully. After this the process is repeated for the *body* actor. If the corresponding *instance_created* event is received two *start_instance* events are send to the object manager to start execution of both actors.

The *Body* instance doesn't do much more than keeping track of the user input and adjusting the tank's position and angle accordingly. The *Cannon* instance does the same for the angle of the cannon, but additionally enforces reloading time as can be seen in Code Sample 4.2.

```
<class name="Cannon">
  <attribute name="tank" type="PlayerTank"/>
  <attribute name="reload_time" type="float"/>
  <method name="Cannon">
    <parameter type="PlayerTank" name="tank"></parameter>
    <body>
      self.tank = tank
      self.reload_time = tank.getReloadTime()
    </body>
  </method>
  <scxml>
    <parallel id="container">
      <state id="rotating" initial="none">
        <state id="none">
          <transition port="input" event="cannon-left-pressed" target="../left"/>
          <transition port="input" event="cannon-right-pressed" target="../right"/>
        </state>
        <state id="left">
          <transition port="input" event="cannon-left-released" target="../none"/>
          <transition port="input" event="cannon-right-pressed" target="../both"/>
          <transition port="engine" event="update" target=".">
            <script>
              self.tank.turnCannonLeft()
            </script>
          </transition>
        </state>
        <state id="both">
          <transition port="input" event="cannon-left-released" target="../right"/>
          <transition port="input" event="cannon-right-released" target="../left"/>
        </state>
        <state id="right">
          <transition port="input" event="cannon-left-pressed" target="../both"/>
          <transition port="input" event="cannon-right-released" target="../none"/>
          <transition port="engine" event="update" target=".">
            <script>
              self.tank.turnCannonRight()
            </script>
          </transition>
        </state>
      </state>
      <state id="shoot" initial="hold">
        <state id="hold">
          <transition port="input" event="shoot-pressed" target="../shoot">
            <raise event="shoot"/>
          </transition>
        </state>
        <state id="shoot">
          <transition port="input" event="shoot-released" target="../hold"/>
          <transition event="loaded" target=".">
            <raise event="shoot"/>
          </transition>
        </state>
      </state>
      <state id="ammo" initial="loaded">
        <state id="loaded">
          <transition event="shoot" target="../unloaded">
            <script>
              self.tank.shoot()
            </script>
            <raise port="gui" event="reloading"/>
          </transition>
        </state>
        <state id="unloaded">
          <transition after="SELF.reload_time" target="../loaded">
            <raise event="loaded"/>
            <raise port="gui" event="loaded"/>
          </transition>
        </state>
      </state>
    </parallel>
  </scxml>
</class>
```

Code Sample 4.2: The *Cannon* component of the player-controlled tank model.

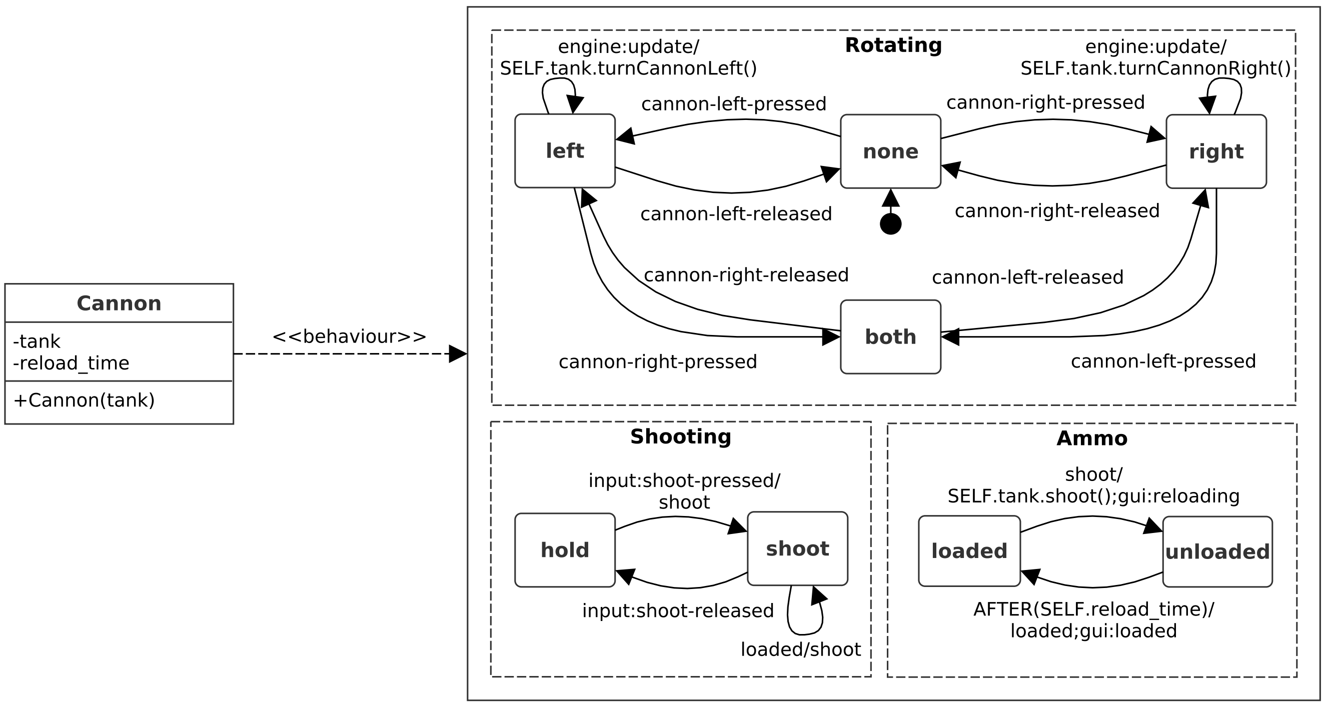

Figure 4.3: Visual representation of Code Sample 4.2

4.2 Computer-Controlled Tank

The model for the NPC is based on the findings in Model-based Design of Computer-Controlled Game Character Behavior [JK07]. They propose a layered architecture, as can be seen in figure 4.4. Input arrives at the sensors layer and output is generated in the actuators. These two layers have the lowest level of abstraction, they closely correspond to the actual components of the NPC. The center layers on the other hand model the high-level goal of the NPC. They adjust state based on the received events from the upper layers, while generating events in order to achieve the current goal.

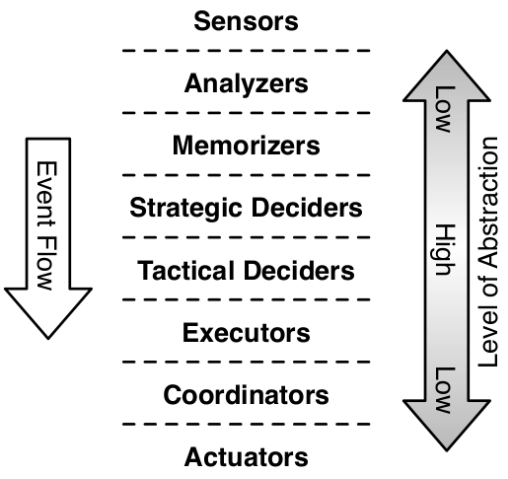

Figure 4.4: The layered architecture used to model the tank NPC.

As in the previous section we'll look at the different classes in the model separately and each time provide the relevant part of the XML file that forms the model. Since the default actor is similar to the one of the player-controlled tank model - it only differs in the number of instances that have to be created and associated - we will skip it and start with the sensors and work our way down the hierarchy to the actuators.

The *radar* component (see Code Sample 4.3) is a sensor that will announce the position of the closest enemy, if there is one in the range of the tank. The very trivial state chart starts in the *no_enemy* state and continuously checks for a present enemy using the defined member function *isEnemyVisible()* as guard. If the latter evaluates positively then the transition to *enemy_in_sight* is triggered. Once in that state, the component will recheck the position of the enemy tank for every *update* it receives from the *engine* port. If an enemy tank is still visible the updated position is send to the *enemy tracker*, otherwise the transition to *no_enemy* is fired.

```
<class name="Radar">
  <attribute name="tank" type="PlayerTank"/>
  <attribute name="range" init-value="2000"/>
  <relationships>
    <association class="EnemyTracker" name="enemy_tracker" min="1" max="1"/>
    <association class="PilotStrategy" name="pilot_strategy" min="1" max="1"/>
  </relationships>
  <method name="Radar">
    <parameter type="AITank" name="tank"/>
    <body>
      self.tank = tank
    </body>
  </method>
  <method name="isEnemyVisible" type="bool">
    <body>
```

```
            sighted_list = self.tank.field.getSightedEnemies(self.tank, self.range)
            if len(sighted_list) > 0 :
                return True
            return False

    </body>
  </method>
  <method name="getEnemyPos" type="bool">
    <body>

            sighted_list = self.tank.field.getSightedEnemies(self.tank, self.range)
            if len(sighted_list) > 0 :
                sighted_list.sort(key=lambda x: x[1])
                return sighted_list[0][0]
            else :
                return (-1,-1)

    </body>
  </method>
  <scxml initial="no_enemy">
    <state id="no_enemy">
      <transition cond="SELF.isEnemyVisible()" target="../enemy_in_sight">
        <raise event="enemy_sighted" target="enemy_tracker">
          <parameter expr="SELF.getEnemyPos()"/>
        </raise>
        <raise event="enemy_sighted" target="pilot_strategy">
          <parameter expr="SELF.getEnemyPos()"/>
        </raise>
      </transition>
    </state>
    <state id="enemy_in_sight">
      <transition cond="not SELF.isEnemyVisible()" target="../no_enemy">
        <raise event="enemy_out_of_sight" target="enemy_tracker"/>
      </transition>
      <transition event="update" port="engine" cond="SELF.isEnemyVisible()"
   target=".">
        <raise event="enemy_pos" target="enemy_tracker">
          <parameter expr="SELF.getEnemyPos()"/>
        </raise>
      </transition>
    </state>
  </scxml>
</class>
```
Code Sample 4.3: The *Radar* actor is responsible for announcing the position of any tank it notices.

In Code Sample 4.4 we see a memorizer responsible for tracking the enemy's position using the infor-

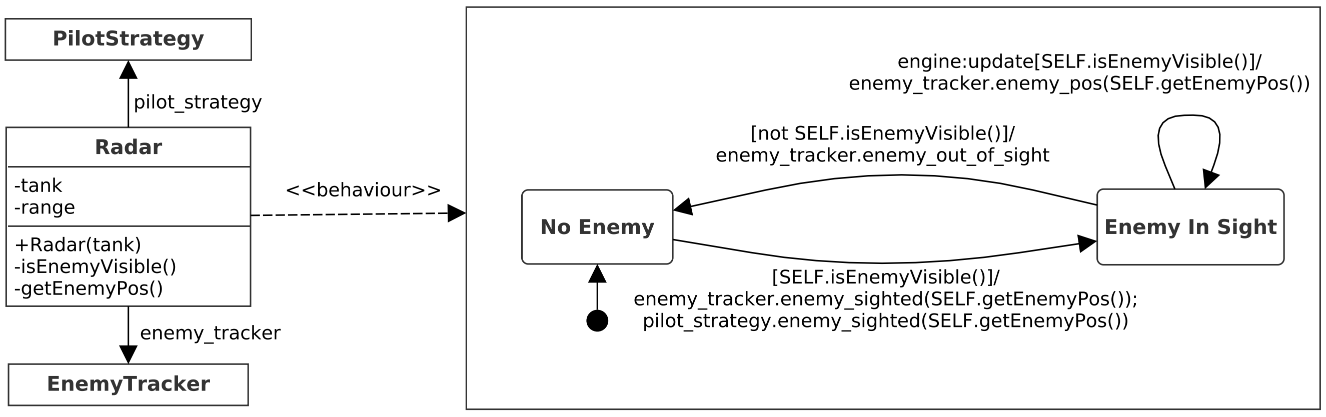
PilotStrategy
pilot_strategy
Radar
-tank
-range
+Radar(tank)
-isEnemyVisible()
-getEnemyPos()
enemy_tracker
EnemyTracker
<<behaviour>>
engine:update[SELF.isEnemyVisible()]/
enemy_tracker.enemy_pos(SELF.getEnemyPos())
[not SELF.isEnemyVisible()]/
enemy_tracker.enemy_out_of_sight
No Enemy
Enemy In Sight
[SELF.isEnemyVisible()]/
enemy_tracker.enemy_sighted(SELF.getEnemyPos());
pilot_strategy.enemy_sighted(SELF.getEnemyPos())

Figure 4.5: Visual representation of Code Sample 4.3

mation received from the *radar*. Once an *enemy_pos* event is received the state chart advances to the *enemy_pos_known* state and saves the position, which is supplied as parameter to the received event, in the *enemy_pos* attribute. For every subsequent *enemy_pos* event, the position is compared using the *hasEnemyMoved()* method. If the position has changed significantly, an *enemy_pos_changed* event is cast to the *attack planner* and the memorized position is updated.

Upon receipt of an *enemy_out_of_sight* from the *radar*, the state *enemy_pos_unsure* is entered. While in this state, the current saved position is the position where the enemy tank was last seen before disappearing. Remembering this position is useful as it can be used to look for and hunt down the enemy. It is however no longer relevant once a *destination_reached* event has been received from one of the upper layers. In this case the *no_enemy* state will be entered after sending an *enemy_lost* event to the *attack planner*.

```
<class name="EnemyTracker">
  <attribute name="enemy_pos" type="Position"/>
  <relationships>
    <association class="PilotStrategy" name="pilot_strategy" min="1" max="1"/>
    <association class="AttackPlanner" name="attack_planner" min="1" max="1"/>
  </relationships>
  <method name="hasEnemyMoved">
    <parameter name="new_position"/>
    <body>
      return new_position != self.enemy_pos
    </body>
  </method>
  <scxml initial="no_enemy">
    <state id="no_enemy">
      <transition event="enemy_sighted" target="../enemy_pos_known">
        <parameter name="enemy_position"/>
        <script>
          self.enemy_pos = enemy_position
        </script>
      </transition>
    </state>
    <state id="enemy_pos_known">
      <transition event="enemy_pos" cond="SELF.hasEnemyMoved(position)" target=".">
        <parameter name="position"/>
        <script>
          self.enemy_pos = position
        </script>
        <raise event="enemy_pos_changed" target="attack_planner">
          <parameter expr="SELF.enemy_pos"/>
        </raise>
      </transition>
      <transition event="enemy_out_of_sight" target="../enemy_pos_unsure">
        <raise event="enemy_out_of_sight" target="attack_planner"/>
      </transition>
    </state>
    <state id="enemy_pos_unsure">
      <transition event="destination_reached" target="../no_enemy">
        <raise event="enemy_lost" target="pilot_strategy"/>
      </transition>
      <transition event="enemy_sighted" target="../enemy_pos_known">
        <parameter name="position"/>
        <script>
          self.enemy_pos = position
        </script>
      </transition>
    </state>
  </scxml>
</class>
```

Code Sample 4.4: The *EnemyTracker* actor is responsible for tracking the enemy's position using the information received from the *radar*.

The *Pilot Strategy* component seen in Code Sample 4.5 is a strategic decider. It chooses which of the two available strategies - exploring or attacking - should be followed. The NPC starts in the *exploring* state (casting an *explore* event to the *Explore Planner* upon entering) and transitions to the *attacking* state upon receiving the *enemy_sighted* event from the *Radar*. The state chart will return to the *exploring* state if the *Enemy Tracker* casts an *enemy_lost* event.

```
<class name="PilotStrategy">
  <relationships>
    <association class="ExplorePlanner" name="explore_planner" min="1" max="1"/>
    <association class="AttackPlanner" name="attack_planner" min="1" max="1"/>
  </relationships>
```

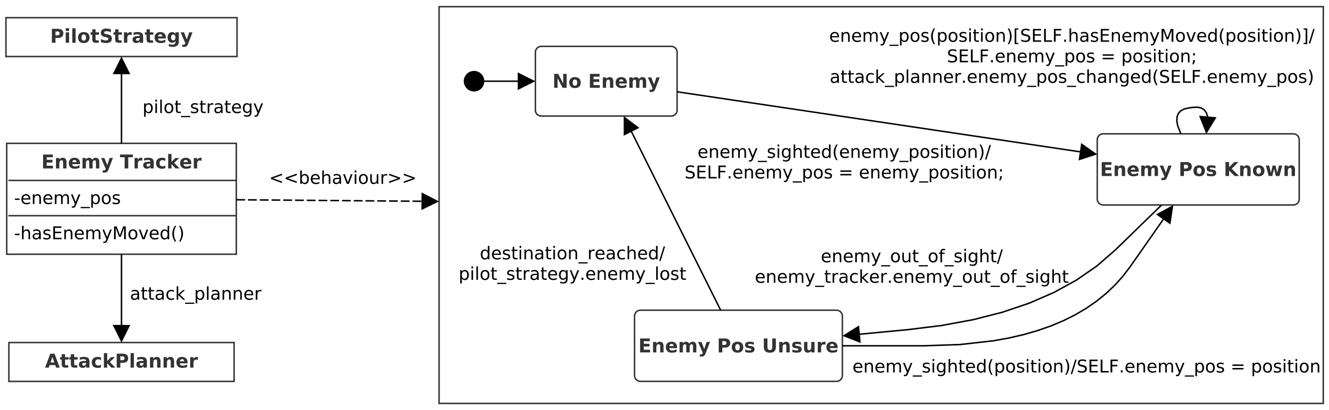
PilotStrategy
pilot_strategy
Enemy Tracker
-enemy_pos
-hasEnemyMoved()
attack_planner
AttackPlanner
<<behaviour>>
No Enemy
enemy_pos(position)[SELF.hasEnemyMoved(position)]/
SELF.enemy_pos = position;
attack_planner.enemy_pos_changed(SELF.enemy_pos)
enemy_sighted(enemy_position)/
SELF.enemy_pos = enemy_position;
Enemy Pos Known
destination_reached/
pilot_strategy.enemy_lost
enemy_out_of_sight/
enemy_tracker.enemy_out_of_sight
Enemy Pos Unsure
enemy_sighted(position)/SELF.enemy_pos = position

Figure 4.6: Visual representation of Code Sample 4.4

```
  <scxml initial="exploring">
    <state id="exploring">
      <onentry>
        <raise event="explore" target="explore_planner"/>
      </onentry>
      <onexit>
        <raise event="stop_exploring" target="explore_planner"/>
      </onexit>
      <transition event="enemy_sighted" target="../attacking">
        <parameter name="position"/>
        <raise event="attack" target="attack_planner">
          <parameter expr="position"/>
        </raise>
      </transition>
    </state>
    <state id="attacking">
      <onexit>
        <raise event="stop_attacking" target="attack_planner"/>
      </onexit>
      <transition event="enemy_lost" target="../exploring">
      </transition>
    </state>
  </scxml>
</class>
```

Code Sample 4.5: The *PilotStrategy* actor decides which strategy the NPC should follow.

For both of the strategies a separate tactical decider is present in our model. On the one hand there's an *explore planner* which will simply generate a new destination (if the NPC resides in exploring mode) each time there's no pending destination for the NPC. On the other hand we have the *attack planner* which is shown in Code Sample 4.6. The planner expects an *attack* event to supply an enemy position. It forwards

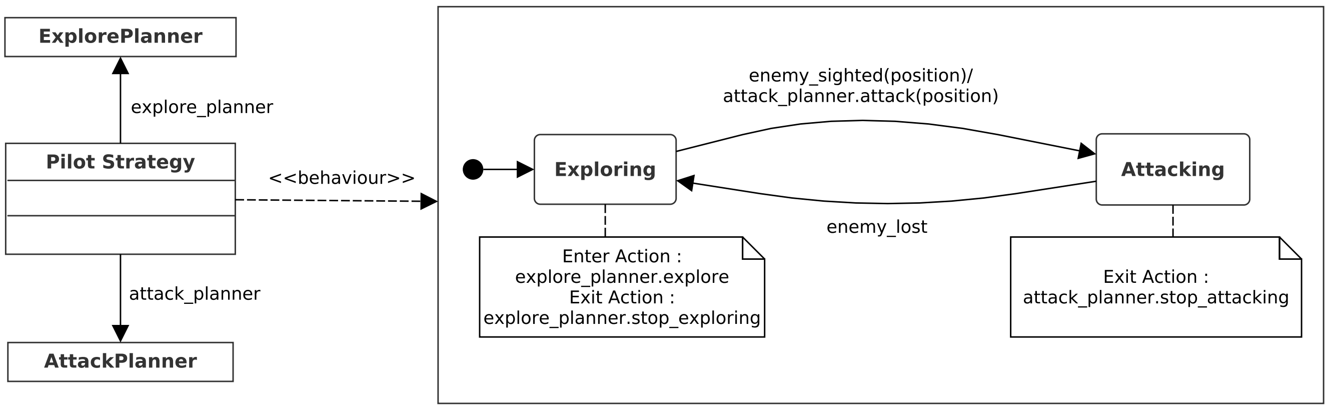

Figure 4.7: Visual representation of Code Sample 4.5

this position to the *path finder* as new destination and to the *turret steering* as target to aim for. In the orthogonal component *action* both the movement of the tank and the shooting of the turret is controlled simultaneously. New destinations are set if needed and when the enemy gets out of sight, the planner informs the *turret steering* component to interrupt aiming. In the *shooting* state we see that after firing a shot, it takes 0.5 seconds to reload.

```
<class name="AttackPlanner">
  <relationships>
    <association class="TurretSteering" name="turret_steering" min="1" max="1"/>
    <association class="PathFinder" name="path_finder" min="1" max="1"/>
    <association class="TurretControl" name="turret_control" min="1" max="1"/>
  </relationships>
  <scxml initial="idle">
    <state id="idle">
      <transition event="attack" target="../action">
        <parameter name="enemy_pos"/>
        <raise event="new_destination" target="path_finder">
          <parameter expr="enemy_pos"/>
        </raise>
        <raise event="aim_at" target="turret_steering">
          <parameter expr="enemy_pos"/>
        </raise>
      </transition>
    </state>
    <parallel id="action">
      <transition event="stop_attacking" target="../idle">
        <raise event="stop_aiming" target="turret_steering"/>
      </transition>
      <state id="movement">
        <state id="following">
          <transition event="enemy_pos_changed" target=".">
```

```
            <parameter name="enemy_pos"/>
            <raise event="new_destination" target="path_finder">
              <parameter expr="enemy_pos"/>
            </raise>
            <raise event="aim_at" target="turret_steering">
              <parameter expr="enemy_pos"/>
            </raise>
          </transition>
          <transition event="enemy_out_of_sight" target=".">
            <raise event="stop_aiming" target="turret_steering"/>
          </transition>
        </state>
      </state>
      <state id="shooting" initial="loaded">
        <state id="loaded">
          <transition event="ready_to_shoot" target="../reloading">
            <raise event="shoot" target="turret_control"/>
          </transition>
        </state>
        <state id="reloading">
          <transition after="0.5" target="../loaded"/>
        </state>
      </state>
    </parallel>
  </scxml>
</class>
```

Code Sample 4.6: The *AttackPlanner* component is responsible for instructing the NPC when in attack mode.

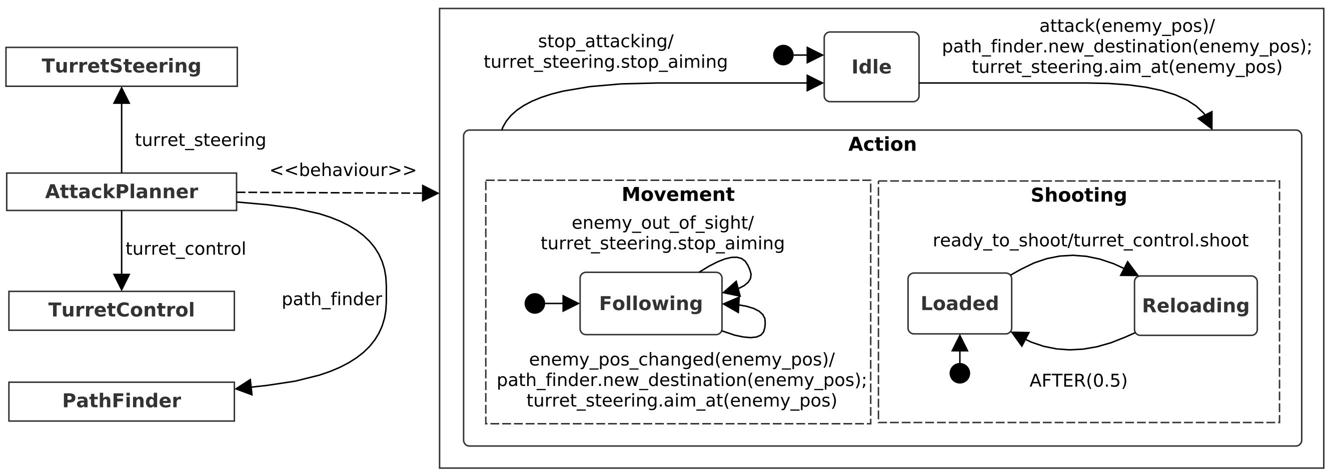
TurretSteering
turret_steering
AttackPlanner
<<behaviour>>
turret_control
TurretControl
path_finder
PathFinder
stop_attacking/
turret_steering.stop_aiming
Idle
attack(enemy_pos)/
path_finder.new_destination(enemy_pos);
turret_steering.aim_at(enemy_pos)
Action
Movement
enemy_out_of_sight/
turret_steering.stop_aiming
Following
enemy_pos_changed(enemy_pos)/
path_finder.new_destination(enemy_pos);
turret_steering.aim_at(enemy_pos)
Shooting
ready_to_shoot/turret_control.shoot
Loaded
Reloading
AFTER(0.5)

Figure 4.8: Visual representation of Code Sample 4.6

Another strategic decider can be seen in Code Sample 4.7. The *PathFinder* translates destinations, requested by one of the strategy planners, into a path of way-points. In the default *idle* state, the component either expects a *waypoint_reached* event or a *new_ destination* event. If the latter is the case and the *PathFinder* decides, using its *requiresNewPath()* method, that the newly received destination is sufficiently different than the last destination (if any) for which a path was set, then a new set of way points is calculated and the *check_points* state is entered. In this state either the next way-point is send to the *Steering* actor or a *destination_reached* event is broadcast if no more way-points are left.

```
<class name="PathFinder">
  <attribute name="waypoints" init-value="[]"/>
  <attribute name="destination" init-value="(-1,-1)"/>
  <attribute name="map"/>
  <attribute name="tank" />
  <relationships>
    <association class="Steering" name="steering" min="1" max="1"/>
  </relationships>
  <method name="PathFinder">
    <parameter type="AITank" name="tank"/>
    <parameter type="AIMap" name="aimap"/>
    <body>
      self.tank = tank
      self.map = aimap
    </body>
  </method>
  <method name="calculatePath">
    <body>
      return self.map.calculatePath(self.tank.getPosition(), self.destination)
    </body>
  </method>
  <method name="requiresNewPath">
    <parameter name="new_destination"/>
    <body>
      return self.map.calculateCell(self.destination) !=
   self.map.calculateCell(new_destination)
    </body>
  </method>
  <method name="morePoints">
    <body>
      return len(self.waypoints) > 0
    </body>
  </method>
  <scxml initial="idle">
    <state id="idle">
      <transition event="waypoint_reached" target="../check_points"/>
      <transition event="new_destination" cond="SELF.requiresNewPath(destination)"
   target="../check_points">
```

```
        <parameter name="destination"/>
        <script>
          self.destination = destination
          self.waypoints = self.calculatePath()
        </script>
      </transition>
    </state>
    <state id="check_points">
      <transition cond=" SELF.morePoints()" target="../idle">
        <script>
          next_waypoint = self.waypoints.pop(0)
        </script>
        <raise event="new_waypoint" target="steering">
          <parameter expr="next_waypoint"/>
        </raise>
      </transition>
      <transition cond="not SELF.morePoints()" target="../idle">
        <raise event="destination_reached" scope="broad"/>
      </transition>
    </state>
  </scxml>
</class>
```

Code Sample 4.7: The *PathFinder* translates destinations into a path of way-points.

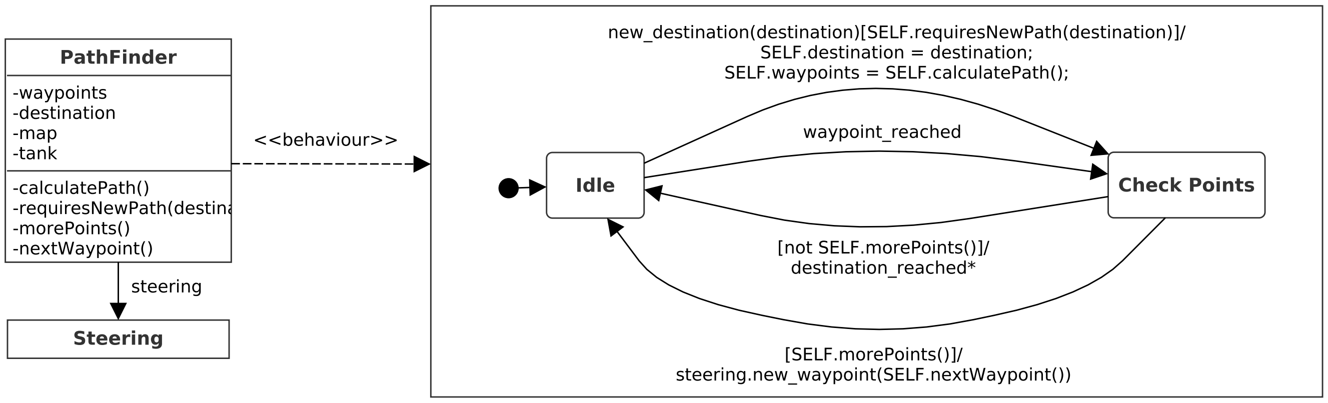

Figure 4.9: Visual representation of Code Sample 4.7

Two executors are present in the model that translate the high level commands received from the strategic layers into events that the actuators can understand. While the *TurretSteering* component is responsible for generating events for the *TurretControl* actuator, the *Steering* executor will cast events mainly to the *MotorControl* component. In Code Sample 4.8 we see that the *TurretSteering* saves the target position it received with the *aim_at* event. For every time interval defined by the NPC's reaction speed, the angle between the tank and target is calculated and events are generated to adjust the tank's angle with as goal to directly face the target.

```
<class name="TurretSteering">
  <attribute name="reaction_time" init-value="0.05"/>
  <attribute name="tank"/>
  <attribute name="margin"/>
  <atribute name="target" init-value="(-1,-1)"/>
  <method name="TurretSteering">
    <parameter type="AITank" name="tank"/>
    <body>
      self.tank = tank
      self.margin = tank.cannonSpeed * D1
    </body>
  </method>
  <relationships>
    <association class="TurretControl" name="turret_control" min="1" max="1"/>
    <association class="AttackPlanner" name="attack_planner" min="1" max="1"/>
  </relationships>
  <method name="pointRight">
    <body>

            goal_angle = self.tank.angleToDest(self.target)
            diff = (self.tank.cannonAngle - goal_angle) % D360
            if diff >= self.margin and diff <= math.pi:
                return True
            return False

    </body>
  </method>
  <method name="pointLeft">
    <body>

            goal_angle = self.tank.angleToDest(self.target)
            diff = (goal_angle - self.tank.cannonAngle) % D360
            if diff >= self.margin and diff <= math.pi:
                return True
            return False

    </body>
  </method>
  <method name="pointCorrect">
    <body>

            goal_angle = self.tank.angleToDest(self.target)
            diff = math.fabs(goal_angle - self.tank.cannonAngle)
            if diff < self.margin or diff > (D360- self.margin):
                return True
            return False

    </body>
  </method>
  <scxml initial="idle">
    <state id="idle">
      <transition event="aim_at" target="../aiming">
        <parameter name="target"/>
        <script>
          self.target = target
        </script>
      </transition>
    </state>

    <state id="aiming" initial="adjust">
      <transition event="stop_aiming" target="../idle">
        <raise event="stop_turning" target="turret_control"/>
      </transition>
      <transition event="aim_at" target=".">
        <parameter name="target"/>
        <script>
          self.target = target
        </script>
      </transition>

      <state id="adjust">
        <transition cond="SELF.pointRight()" target="../wait">
          <raise event="turn_right" target="turret_control"/>
        </transition>
        <transition cond="SELF.pointLeft()" target="../wait">
          <raise event="turn_left" target="turret_control"/>
        </transition>
        <transition cond="SELF.pointCorrect()" target="../wait">
          <raise event="stop_turning" target="turret_control"/>
          <raise event="ready_to_shoot" target="attack_planner"/>
        </transition>
      </state>
      <state id="wait">
        <transition after="SELF.reaction_time" target="../adjust"/>
      </state>
    </state>
  </scxml>
</class>
```

Code Sample 4.8: The *TurretSteering* actor generates events for the *TurretControl* actuator.

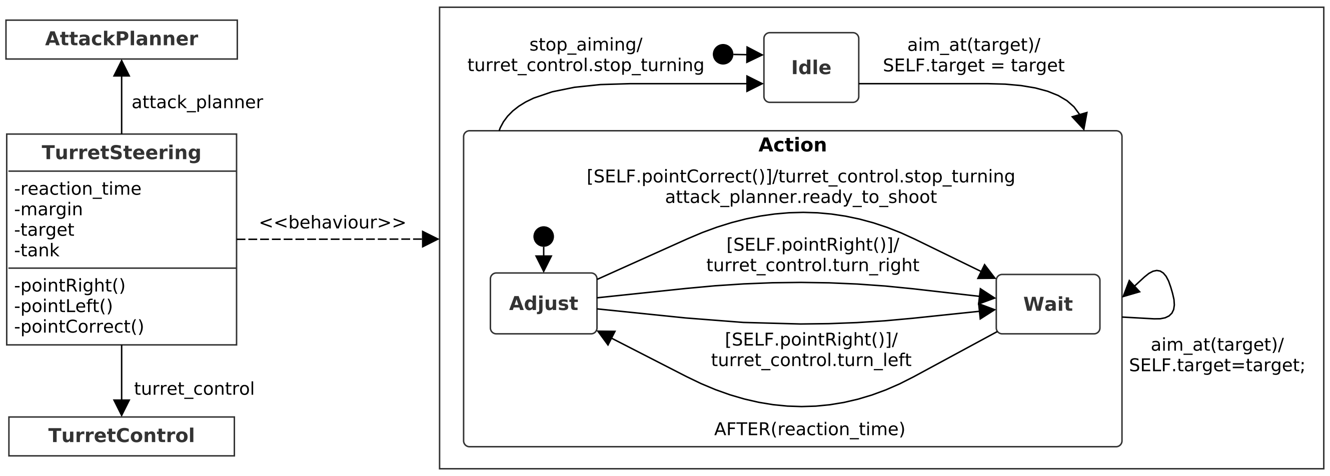

Figure 4.10: Visual representation of Code Sample 4.8

The model contains two actuators, one for the motor and one for the turret. Since they are very similar, we only look into the *TurretControl* class which can be seen in Code Sample 4.9. As the actuators are responsible for executing the actions they received from the upper layers, they should have access to the tank object they need to adapt and hence this object should be supplied as parameter for the constructors. Since movement doesn't happen continuously but in discrete steps, the actuator components only execute a movement action on the receipt of an *update* event from the environment through the *engine* port. What action should be executed is determined by the current state of the components' state chart, which in turn depends on the events they received from the upper layers. For example after processing the event *turn_right*, the current state will be *turning_right*. If subsequently an *update* event is received, the *turnTurretRight()* method is called on the tank which will adapt the angle of the turret.

```
<class name="TurretControl">
  <attribute name="tank"/>
  <method name="TurretControl">
    <parameter type="AITank" name="tank"/>
    <body>
      self.tank = tank
    </body>
  </method>
  <scxml>
    <parallel id="turret">
      <state id="rotation" initial="none">
        <state id="none">
          <transition event="turn_right" target="../turning_right"/>
          <transition event="turn_left" target="../turning_left"/>
        </state>
        <state id="turning_left">
          <transition event="stop_turning" target="../none"/>
          <transition event="turn_right" target="../turning_right"/>
          <transition event="update" port="engine" target=".">
            <script>
              self.tank.turnTurretLeft()
            </script>
          </transition>
        </state>
        <state id="turning_right">
          <transition event="stop_turning" target="../none"/>
          <transition event="turn_left" target="../turning_left"/>
          <transition event="update" port="engine" target=".">
            <script>
              self.tank.turnTurretRight()
            </script>
          </transition>
        </state>
      </state>
      <state id="shooting">
        <state id="polling">
          <transition event="shoot" target=".">
            <script>
              self.tank.shoot()
            </script>
          </transition>
        </state>
      </state>
    </parallel>
  </scxml>
</class>
```
Code Sample 4.9: The *TurretControl* actor which is responsible for executing the actions related to the turret of the tank.

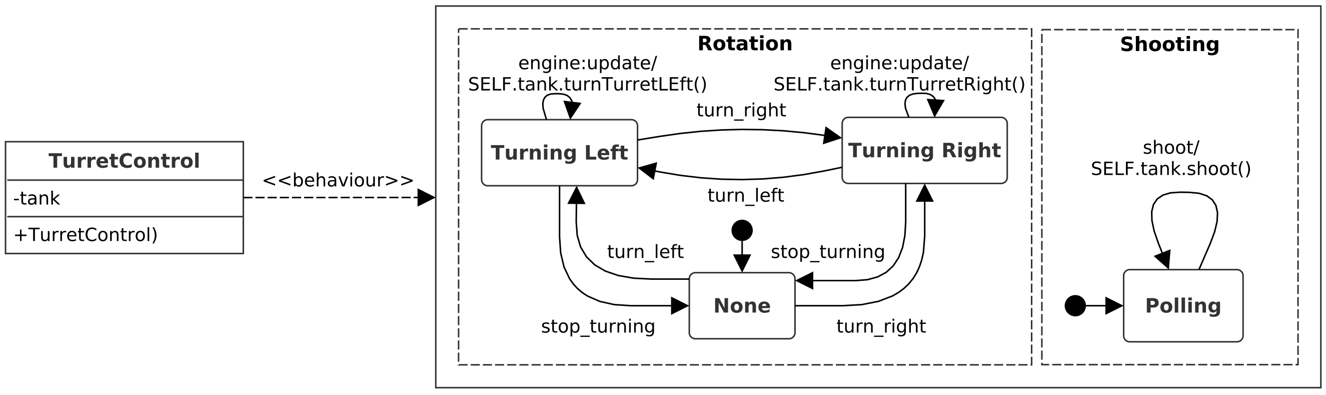

Figure 4.11: Visual representation of Code Sample 4.9

5 Conclusion and Future Work

Our extensive example confirms the statements made in [JK07] and [DKVV11] that Statecharts and its variants are a viable option to define the behaviour of Non-Playable Characters. What's more interesting however are the benefits of the class diagram features added to the formalism. Not only does it make the integration of compiled models into test environments easier and more straight forward, our strictly imposed rules and the conformance to the actor model makes it more reliable and deadlock proof.

As for future work, the most straight forward thing to do is improving the compiler to make Statecharts a viable option performance-wise. Regarding the modelling of AI, our goal is to convince the game industry that this technique is worth investing in. This has to be done by building a proof of concept in a commercial game engine, to show that it is applicable in every environment. Furthermore, it would be a good idea to let two groups of people build the same AI. One group would make use of the SCCDXML formalism while the other group would rely solely on a programming language. This way we can compare the results and see which technique produced the best AI within the same budget.

Bibliography

[BAA$^+$13] Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel C. Burnett, Jerry Carter, Scott McGlashan, TorbjÃűrn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, No'am Rosenthal, and Johan Roxendal. State chart xml (scxml): State machine notation for control abstraction. http://www.w3.org/TR/scxml/, 2013. Accessed: 2013-08-02.

[CD99] James Clark and Steve DeRose. Xml path language. *W3C Recommendation*, 1999.

[Cla99] James Clark. Xsl transformations (xslt). *W3C Recommendation*, 1999.

[DH96] Amnon Naamad David Harel. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293 – 333, 1996.

[DH04] Hillel Kugler David Harel. The rhapsody semantics of statecharts. *Lecture Notes in Computer Science*, 8(3147):325 – 354, 2004.

[DKVV11] Christopher Dragert, Jörg Kienzle, Hans Vangheluwe, and Clark Verbrugge. Generating extras: Procedural ai with statecharts. Technical report, Technical Report SOCS-TR-2011.1, 2011.

[Har87] David Harel. Statecharts : a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[JK07] Hans Vangheluwe Jörg Kienzle, Alexandre Denault. Model-based design of computer-controlled game character behavior. *Lecture Notes in Computer Science*, 4735, 2007.

[OMG05] OMG. Uml 2.0 superstructure specification. Technical report, Object Management Group, 2005.