A Visual Modelling and Simulation Environment for State-Charts Class-Diagram (SCCD) Formalism
Research Internship

Addis Gebremichael

addis.gebremichael@student.ua.ac.be

Abstract

SCCD is a formalism that combines Harel Statecharts with UML Class-diagrams. It allows users to model complex, timed, autonomous, reactive, and dynamic-structure systems. Moreover, a textual concrete syntax, i.e. SCCDXML, was defined and a compiler was developed that generates executable code from models expressed in SCCDXML for different platforms and target languages. Nonetheless, Statecharts, and its extension, SCCD, is a visual topological formalism. Hence, a visual notation is better suited to express SCCD models, and this work brings together the visual concrete syntax of UML's Class-diagram with Harel's Statecharts and builds an interface that supports a modelling and simulation environment for SCCD. SCCD models can be exported to SCCDXML notation which can then be compiled. This environment is bootstrapped, to create and model the editor and its UI behaviour within itself.

Keywords: Modeling languages, Statecharts, Class-diagram, SCCD, Executable models, Model-driven Engineering.

1. Introduction

Statecharts, first introduced by David Harel [1], is a visual modeling language that is a higraph-based extension of standard state-transition diagrams that is used to aid the specification of complex, reactive, timed, autonomous, interactive discrete-event systems. It's appropriate for describing large and reactive' systems as it naturally adds the notion of depth, orthogonality and modularity,to 'normal' Finite State Automata (FSAs) [2]. However it lacks the facilities for specifying the structure of a system in addition to creating, deleting and communicating multiple Statecharts instances at runtime.
Class-diagram, in the Unified Modeling Language (UML) [3], is a visual modeling language that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among classes. SCCD [4] is a hybrid formalism which combines the structural object-oriented expressiveness of Class-diagrams with the behavioural discrete-event characteristics of Statecharts. This, in turn, facilitates the specification

of dynamic-structure systems that are timed, autonomous and reactive. Classes model both structure and behaviour-structure in the form of attributes and relations with other classes, behaviour in the form of methods, which access and change the values of attributes of the class, and a Statecharts model, which describes the modal behaviour of the class, modelling it's control flow. At runtime, a class can be instantiated, which creates an object. Objects are initialized according to the classs constructor, and can be deleted, invoking the classs destructor. After initialization, an object is controlled by its Statecharts through changes in the object's state caused by triggers like events and operation invocation. The relationships modelled between classes are instantiated at runtime in the form of links. These links serve as communication channels, over which objects can send and receive events.
A concrete textual syntax, i.e. the SCCDXML notation, was developed for representing SCCD models and a compiler was developed that generates executable code from SCCDXML models for different platforms. However, Statecharts, and its extension, SCCD, is a visual formalism. Although SCCDXML can be used as a serialization format for the developed compiler, it is not very well suited as a concrete syntax to better express SCCD models. Hence, the overall goal of this research is to bring together the visual concrete syntax of Statecharts and Class-diagrams and build an interface to obtain a user-friendly modelling and simulation environment where a complete visual editor for the SCCD formalism can be developed. The editor application is to be modelled in SCCD, since its interface has a behavior that is reactive, timed, autonomous and dynamic-structure. The visual editor should also support for code generation, i.e. transforming SCCD models to their corresponding SCCDXML notation which can then generate executable code.
The next section gives an overview of the SCCD formalism, followed by a thorough discussion of the visual editor requirements, design choices made and finally a bootstrapping procedure to create and model the editor and its UI behaviour within itself.

2. The SCCD Formalism

In this section the Statecharts and Class-diagram (SCCD) formalism is discussed. Many parts of this section are taken from where the formalism is first defined [4] and a documentation for the SCCDXML notation [5]. The various available constructs of Statecharts and their semantics are discussed first using a neutral visual representation, followed by a discussion of Class-diagram constructs supported by an example illustrating a combination of both Statecharts and Class-diagram in a single SCCD model. Finally, a brief introduction to how events can be used for communication among instances in SCCD models is discussed followed by a highlight of the object manager, which is in charge of the management of objects at runtime.

2.1. Constructs

In this section the semantics of the different constructs that make up the formalism are discussed. For readability purposes, the constructs are categorized as either Statecharts constructs or a concept that encapsulate the Statecharts into a class, and ultimately into a Class-diagram.
The constructs of Statecharts formalism discussed in this section are based on a neutral visual representation as proposed in Harel's definition published in 1987 [1]. Statecharts is a visual modelling language that extends 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 there was no such method available at the time.

2.1.1. Basic State

The basic state acts as the main building block of a Statecharts and is represented by a rounded rectangle as in figure 1, where two basic states are depicted. A basic state, like that of composite and parallel states, represents a mode the system can be in. A state can be entered (which executes an optional block of executable content) and exited (which executes an optional block of executable content) using transitions. A Statecharts consisting solely of basic states has to have exactly one default/initial state, this is represented by an incoming edge with a black-dot as a source, as shown in figure 1.

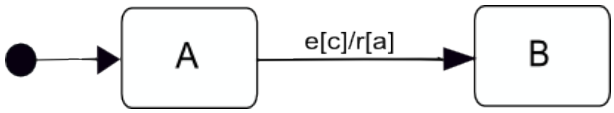


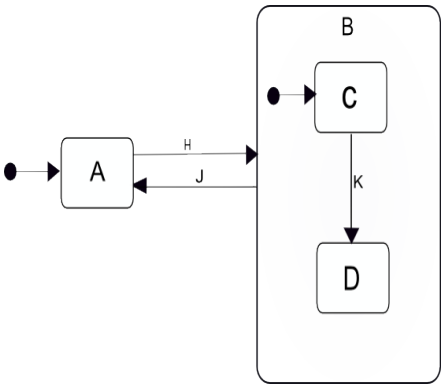
Figure 1: Two basic states connected by a transition originating from the initial state on the left.

2.1.2. Transition

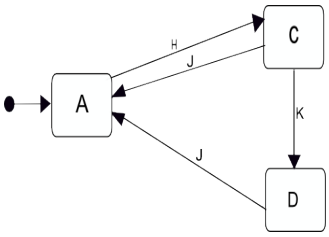
In figure 1, the two states are connected by a transition originating from the initial state on the left, with a label of the form event[guard]/action. This means that, if the current state is A, upon reception of the trigger event e, the Statecharts will transition from state A to state B if and only if the guard condition c is satisfied. This guard condition can reference parameter values received by the transition which catches the event. Upon firing the transition, an action will be executed which in this case is raising an event r and an executable content a. Events in SCCD are strings that are accompanied by a number of parameter values: the sender is obliged to send the correct number of values, and the receiver declares the parameters when catching the event. Each parameter has a name, that can be used as a local variable in the action associated with the transition that catches the event. Thus, transitions are triggered by an event or a timeout, or can be spontaneous. They can optionally specify a condition, an action and raising of an event.

2.1.3. Composite State

Composite states add a notion of hierarchy to Statecharts. 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 and between every level of the state hierarchy.



(a) Composite state B encapsulating the two basic states C and D.



(b) Semantics of Statecharts in (a) by "flattening".

Figure 2: Depth/hierarchy in Statecharts.

To illustrate this we look at the example in Figure 2a. At initialization time, 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 Statecharts in the state configuration where B and its substate D are active, while an event J would bring the Statecharts back to its initial configuration where state A is active. Figure 2b shows the semantics of the Statecharts in figure 2a by "flattening" it. We can see that both substates C and D have an outgoing transition with event J. Thus, either one of the active substates, i.e. C or D can change the Statecharts configuration back to state A. Nonetheless, in case of non-determinism, i.e. if both the parent state and one of its substates have a transition leaving it on the same event, Statecharts will handle this by either enabling the transition associated with the inner substate or the transition associated to the parent state.

As mentioned before, with Statecharts it is possible to define actions and output events, that should be raised on either entering or exiting a specific state. When multiple layers of hierarchy are traversed on firing a transition, these actions are raised in an intuitive way. The exit actions are raised first, from the deepest level up to, but excluding, the first shared parent between the source and target states. This is then followed by executing the enter actions in the opposite direction down to the target states.

2.1.4. Parallel State

Besides the XOR composition achieved by a composite state, also AND composition is available in the Statecharts formalism. These are better known as parallel states or orthogonal components and allow for parallelism to be modelled. Upon entering a parallel state, each of the orthogonal regions (substates) will become active.

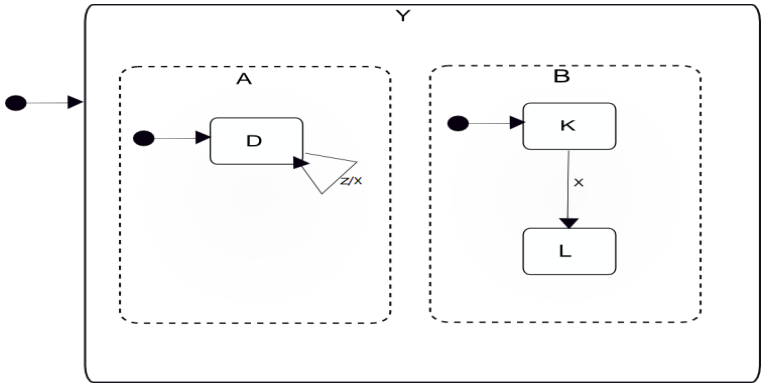


Figure 3: Parallel state Y with two orthogonal regions A and B.

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 3 labelled 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 D and K 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. In addition, an action appearing along a transition in Statecharts is not merely sent to the outside world as an output. Rather, it can affect the behavior of the Statecharts itself in orthogonal components and it is known as *broadcasting*. This can be achieved by a simple broadcast mechanism, as in the Statecharts shown in figure 3, where if an external event Z occurs, a transition labelled Z/X in orthogonal component A is taken, the action X of the transition is immediately activated and regarded as a new event, possibly causing further transitions in other components, in this case transition labelled X in orthogonal component B.

2.1.5. History State

A history state, which is depicted by a circle with the label H, adds memory to a component. A history state keeps track of the current configuration when its parent state is exited. If a transition has the history state as a target, the configuration that was saved is restored. If no configuration was saved yet, the

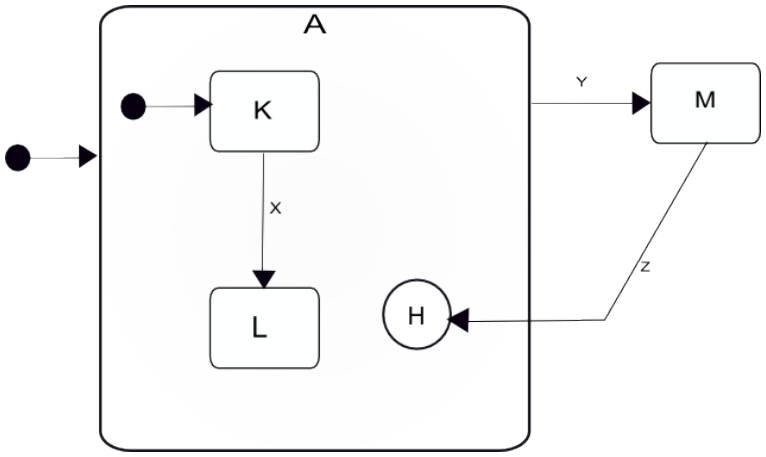


Figure 4: When the transition to the history state is triggered, the state of A will be restored to its last recorded state.

default state is entered instead.
This can be illustrated further as in Figure 4 where there is 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 triggering the transition to state M, which is enabled by the event Y, the current sub-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 restored, where sub-state L is active again. 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 'asterisk' (*) to the state label resulting in H*.

2.1.6. Classes

The top level of a SCCD model resembles a UML Class-diagram with classes and edges connecting them. Classes are the main addition of the SCCD language. They model both structure and behaviour - structure in the form of attributes and relations with other classes, behaviour in the form of methods, which access and change the values of attributes of the class, and a Statecharts model, which describes the modal behaviour of the class modelling it's control flow. At runtime, a class can be instantiated, which creates an object. Objects are initialized according to the classs constructor, and can be deleted, invoking the classs destructor. After initialization, an object is controlled by its Statecharts through changes in the objects state caused by triggers such as events and operation invocations. An SCCD model can also have an *external* class, denoted by a dotted rectangular box, which can be referenced from outside and thus can not be linked to a Statecharts for its behavioural specification. The

relationships modelled between classes are instantiated at runtime in the form of links. They serve as communication channels, over which objects can send and receive events.

2.1.7. Class Relationships

Classes can have relationships with other classes. There are two types of relationships: associations and inheritance. An association is defined between a source class and a target class, and has a name. It allows instances of the source class to send events to instances of the target class by referencing the association name. An association has a multiplicity, which defines the minimal and maximal cardinality. They control how many instances of the target class have to be minimally associated to each instance of the source class, and how many instances of the target class can be maximally associated to each instance of the source class, respectively. Each time an association is created, it results in a link between the source and target object. This link gets a unique identifier, allowing the source object to reference the target, for example to send events.
An inheritance relation results in the source of the relation to inherit all attributes and methods from the target of the relation. Specialisation of modal behaviour (i.e., (parts of) the SCXML model of the superclass) is currently not supported. Inheritance edges have a priority attribute which allows to specify in which order classes need to be inherited (in case of multiple inheritance). Inheritance relations with higher priority are inherited from first.
In Figure 5 we can see the Class-diagram from the perspective of a single class, i.e. ClassD. This class is related with classes ClassE and ClassF by an inheritance edge and a named unidirectional association edge - *assocation_g*, respectively. Finally, a dashed edge with label <<behaviour>>is used to link the class to its corresponding Statecharts.

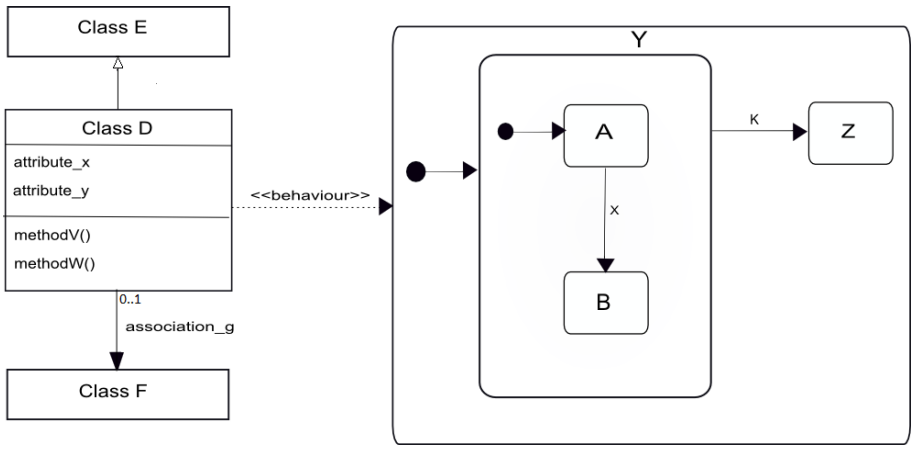


Figure 5: This figure illustrates the relation between classes in a Class-diagram and the Statecharts that describe their behaviour.

2.2. Events in SCCD

In the traditional Statecharts formalism, when casting an event, it was obvious that the scope of an event was local to the Statecharts that generates it. Now SCCD adds the ability to transmit events to class instances and to output ports with the addition of a public input/output interface using ports, as well as classes and associations. Thus, different levels of scope are added as described below by the different scope names used in the action associated with a transition that catches an event.

- *local*: The event will only be visible for the sending instance.
- *broad*: The event is broadcast to all instances.
- *output*: The event is sent to an output port and is only valid in combination with the *port* attribute, which specifies the name of the output port.
- *narrow*: The event is narrow-cast to specific instances only, and is only valid in combination with the *target* attribute, which specifies the instance to send the event to by referencing a link.
- *cd*: The event is processed by the object manager. See the next section for more details.

2.3. Object Manager

At runtime, a central entity called the object manager is responsible for creating, deleting, and starting class instances, as well as managing links (instances of associations) between class instances. It also checks whether no cardinalities are violated: when the user creates an association, it checks that the maximal cardinality is not violated, and when the user deletes an association, it check whether the minimal cardinality is not violated. As mentioned previously, instances can send events to the object manager using the *cd* scope. The object manager can thus be seen as an ever-present, globally accessible object instance, although it is implicitly defined in the runtime, instead of as a SCCD class.
When the application is started, the object manager creates an instance of the default class and starts its associated Statecharts model. From then on, instances can send several events such as *associate_instance* and *delete_instance* to the object manager to control the set of currently executing objects.

3. Visual Editor Requirements

In this section we look into the various requirements of developing the visual editor application. The SCCD formalism, using its textual concrete syntax representation (i.e. SCCDXML), can be chosen as an appropriate language for modelling such an application that has high interactive features with complex user interface requirements. The overall goal is to obtain a user friendly

environment where a complete SCCD model can be created with the functionality to compile the model and generate executable code from it. In addition, the editor should be able to save a model and retrieve it for further editing. The application should also provide a way to set model information such as its name, author, description and imported packages, together with the possibility of defining input and output ports.
As an SCCD model consists out of a Class-diagram in which each class, if not an external class, has a corresponding Statecharts, the editor should provide functionality to create and manage both these types of components in a scalable manner, while still making it clear which class corresponds to which Statecharts. Thus, the requirements of a part of the editor that takes care of a Class-diagram is discussed next, followed by the Statecharts part of the editor.

3.1. Class-diagram Editor

The essential feature of the Class-diagram editor is to be able to provide a top-level view of the classes that make up the SCCD model by having the user create classes on canvas and position them as desired. The user then should be able to create edges to connect the different classes as a means of defining their different relationships. The user should also be able to edit settable properties of a class including its *name* and optionally its *attributes*, *methods*, *constructor* and a *destructor*, and whether the class is the *default* class, an *external* class, or neither. For each attribute, ways to set the name and type should be provided as well as the possibility to define an initial value. Methods require input fields for a return type, name and a function body, in addition to ways to set formal parameters defining their name, type and an optional default value for each method.
While an edge is created to connect different classes, the user should have the option to add movable multiple in between angle points (control points) in order to obtain an optimal diagram layout. An edge among classes can represent two different relationships, namely an association and inheritance. Settable properties for an association include its name, minimum and maximum cardinality, where input fields should be provided. The name of the association has to be displayed as a label. Furthermore, for inheritance edges, input fields for setting its priority and actual parameter values for the target class constructor shall be given.
Finally, there has to be a possibility of associating each class with their corresponding Statecharts. This can be diagrammatically represented using a behaviour edge, with no settable properties.

3.2. Statecharts Editor

The requirements of the Statecharts editor is similar to that of Class-diagram. States can be drawn on canvas and connected via an edge, namely a *Transition* edge. Since Statecharts have four modelling constructs, it's essential that each state type has a different visual representation on canvas in order to have a clear distinction. Moreover, Statecharts also add a notion of hierarchy, where there

has to be a way to represent child-parent relationship. The most intuitive way to do this is by dragging a state into another one and drop it. When doing so, the parent state should be able to automatically re-size itself to accommodate for a child with larger size. Additionally, the parent state should possibly draw a semi-transparent line connected to the child state, just to emphasise the child-parent relationship.
Settable properties for a state should include its *name* and whether or not it is a *default* state. The latter should only be available if the parent is not a parallel state. In case of a history state there should also be an option to change the history type between *shallow* and *deep*. Lastly the user should have the option to define enter and exit actions for a state. As an action consists of an unlimited number of scripts and raise events, the user should be able to add as many of these as needed. Each raise event has an *event*, *scope* and *target* attributes as well as the option to add parameters to be sent with the raised event.
The user should also be able to define actions for transitions. Additionally, input fields should be provided to set a transition's *event*, *guard*, *cond* and *after* attributes. Furthermore, there should be an option to add formal event parameters to a transition where the name and type attributes can be set. Similar to class edges, a transition should also have a label displaying a limited number of characters of its properties in the format T[C] / R[A], where T is the trigger, C a condition, R raised events, and A the action.
Finally, the class and Statecharts editor canvas should enable for zoom capabilities to fully utilize the diagram layout and make the editor scalable for large models.

4. Visual Editor Design Choices

This section thoroughly discusses the design choices made to develop the SCCD visual modeling and simulation environment. The application architecture for the SCCD editor can be summarized as in figure 6.
The front-end application module, which includes all the SCCD classes and their Statecharts for modeling the behaviour of the application user-interface, is discussed first. Furthermore, this module makes use of two interfaces, namely the *model repository API*, which is explained next, and the *simulation interface* that is included in the front-end module. The former is used by the application as an Application Programming Interface (API) to perform CRUD (Create, Read, Update, Delete) operations of model information in a repository, while the latter can be used to interact with the external simulation engine for simulation of Statecharts models. This external simulation engine makes use of the generated SCCDXML to access model instances.
Finally discussed is a third module that the application front-end utilizes, i.e. the *code generator*, which simply makes use of the model information from a repository, via the repository API, to generate SCCDXML code.

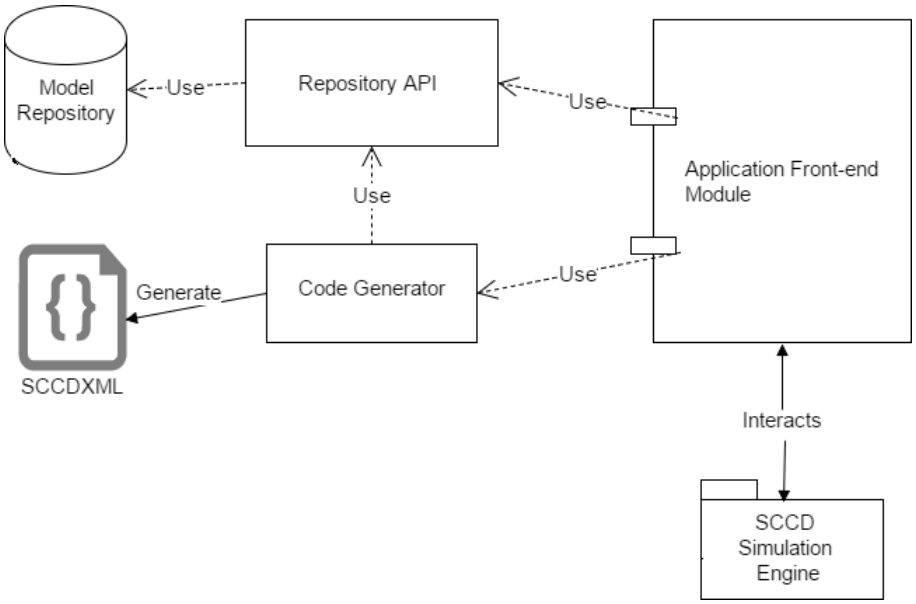


Figure 6: An overview of the application architecture of the SCCD editor.

4.1. Application Front-end

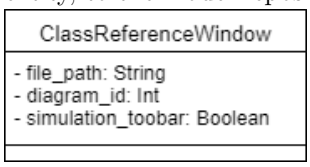
The application front-end includes all the SCCD classes involved in rendering the user-interface and its behaviour. An overview of these classes can be represented in a Class-diagram as in figure 7. For discussion purpose, we classify the Class-diagram into different functional components, namely- the *class-reference* editor, the *class-behaviour (Statecharts)* editor, the *simulation* classes, and other *external* classes used. The following subsections thoroughly discuss the essential classes involved in the aforementioned functional components of the front-end module, along with their corresponding UI appearance and possibly their interaction with other classes depicted in sequence diagrams.

4.1.1. Class-reference editor

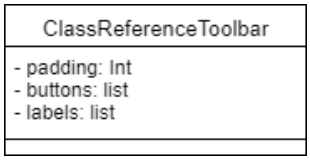
The class-reference editor provides a top-level architectural view of SCCD classes and their corresponding relationships. The main classes involved and their responsibilities are briefly discussed as follows:

- *ClassReferenceWindow* : This class has the role of providing a window widget for an instance of a new SCCD model, where a toolbar with menu items and a canvas can be created and contained in, as shown in figure 9. In addition, it receives and handles toolbar menu events accordingly; creates the corresponding "pop-up" window for editing a modeling construct; and finally makes CRUD operational calls, related to a "Diagram"

entity, to the model repository via the *Repository API*.



- *ClassReferenceToolbar* : The main purpose of this class is to provide a toolbar for menu items in the class-reference editor. Menu items are composed of buttons and labels, as in figure 9. Buttons include icons for various operations such as saving, opening (new or saved models), validating, exporting, and compiling a given model. In addition, there is a button for editing model information, as well as for loading the simulation toolbar. Labels are used to select which type of model construct to use on canvas, in this case a Class-diagram element.



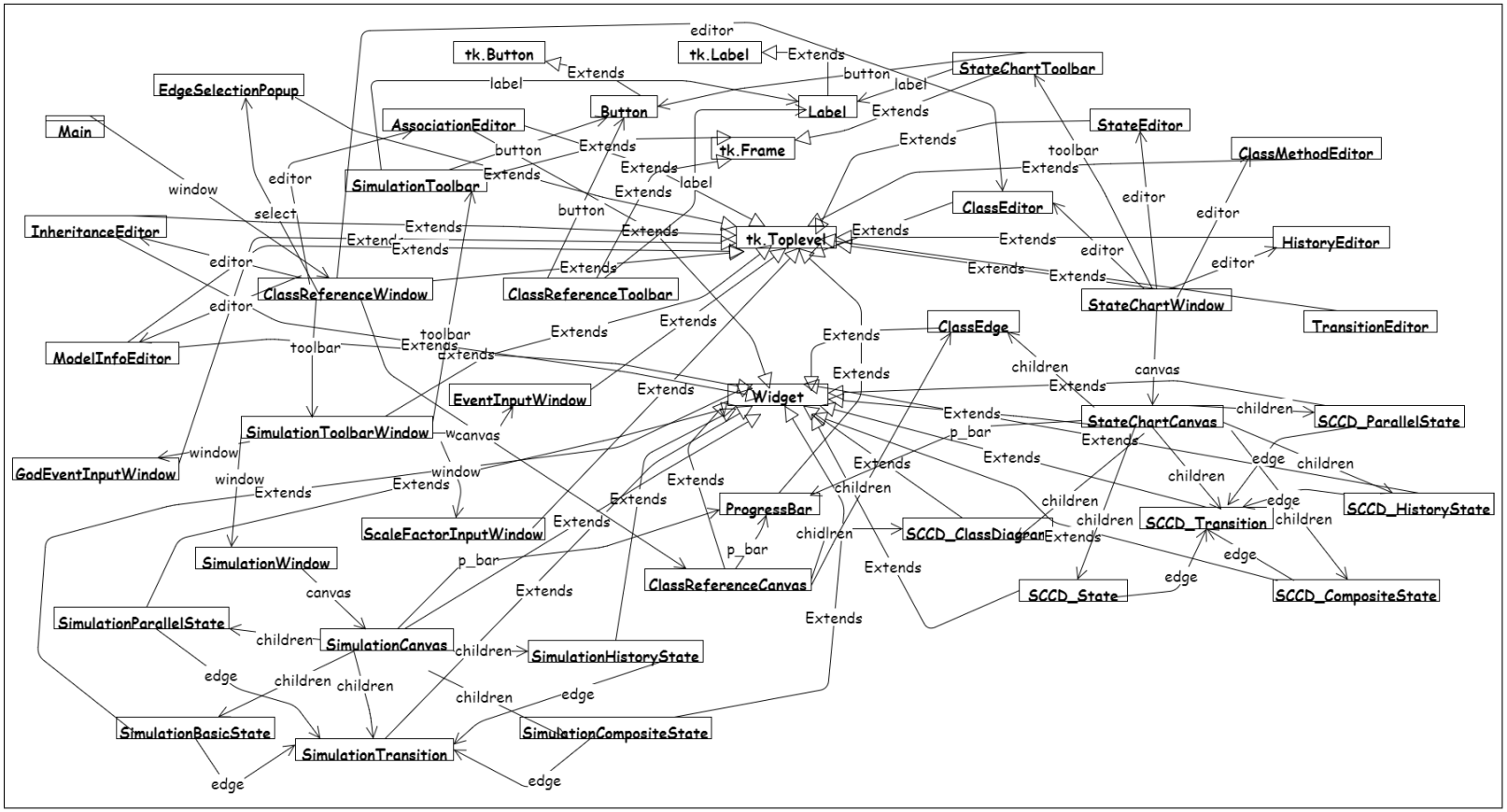
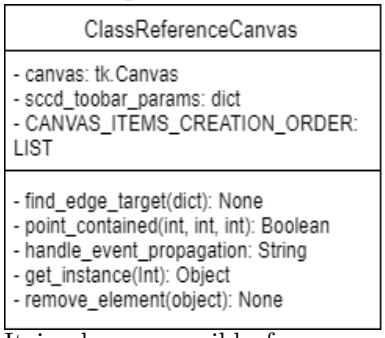


Figure 7: **A Class-diagram depicting the classes involved and their relationship in the application front-end module.**

- *ClassReferenceCanvas*: This class provides a canvas environment for creating/deleting Class-diagram elements and edges representing their different relationships.



It is also responsible for re-generating a saved top-level class-reference model on canvas. In addition, while handling the save button, it sends a "broadcast" message to all elements on canvas to store their current location, as shown in the sequence diagram in figure 8. Moreover, since this

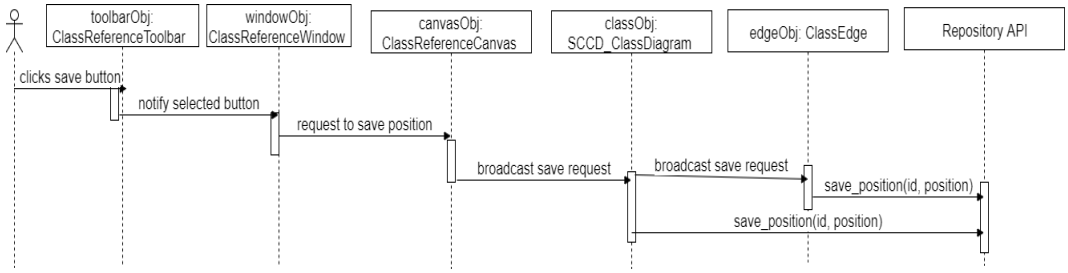


Figure 8: A sequence diagram depicting object interaction at run-time while saving a model on the class-reference editor.

class owns the keyboard focus, keyboard events are sent to this class and subsequently broadcasted to all elements on canvas so that the element on "selected" state will only act up on the received keyboard event. Finally, this class is also responsible for identifying a target class during the creation of an edge connecting class elements. Figure 10 shows the behaviour of this class modelled in the SCCD editor. We can observe that many of the above mentioned roles can be executed concurrently since they're nested inside the parallel state "running".

- *SCCD_ClassDiagram*: This class is responsible for drawing the visual concrete syntax of SCCD Class-diagram on canvas, as shown in figure 9.

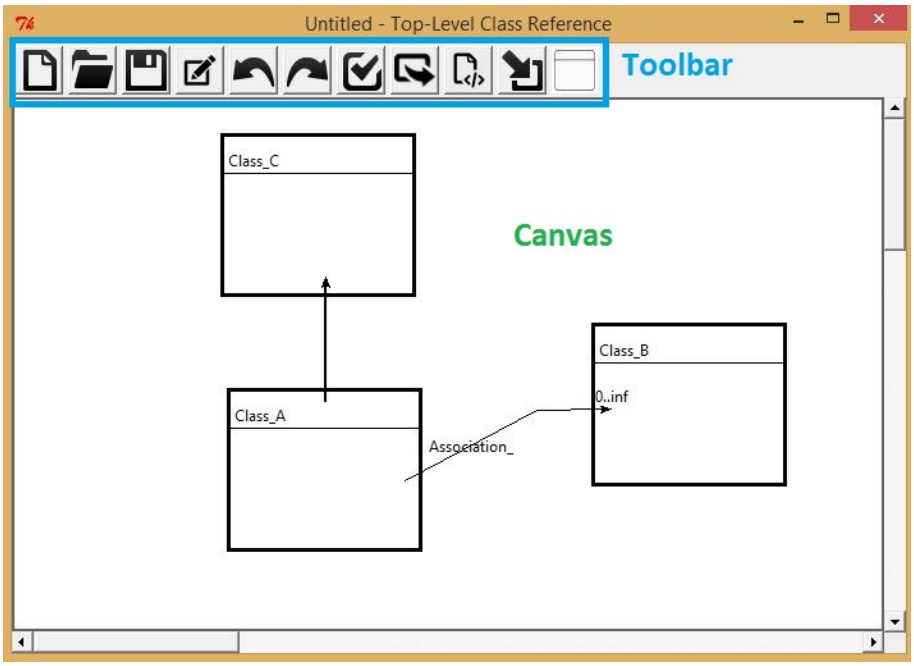


Figure 9: A simple model showing the User-interface of the Class-reference editor.

SCCD_ClassDiagram
- canvas: tk.Canvas - type : String - line_offset: int - text_offset_x: int - text_offset_y: int - tag: String - id: Int - external: Boolean - _drag_data: dict - repository_id : Int
- join_parent(Int): None - exit_parent() - update_hierarchy_position():None

In addition, it listens to and handles all user interaction events accordingly. These interactions are summarized in table 1. While invoking CRUD operational calls of the *Repository API* for a "ClassDiagram" entity, this class listens to any changes to it's corresponding edges (both incoming and outgoing) to also make the appropriate call for updating its edge property.

- *ClassEdge*: There are three types of edges that can be constructed having a Class-diagram as a source element, i.e. *association, inheritance,* and *behaviour* edges. The top-level class reference editor, however, supports for

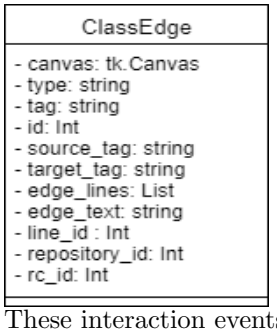
Event ID	Event	Action
1	left-click	Select
2	1 + move	Drag start
3	2 +left-release	Drag end
4	right-click + move	Start edge creation
5	4 + left-click	Create edge control-points
6	4 or 5 + right-release	End edge creation
7	double-click	Display Statecharts model
8	1 + return key	Display attribute editor

Table 1: User interaction events and actions of the SCCDClassDiagram class.

Event ID	Event	Action
1	left-click	Select
2	1 + left-shift	Start editing edge position
3	1 control point + move	Adjust control point
4	1 anywhere on canvas	End editing edge position
5	1 + return key	Display attribute editor

Table 2: User interaction events and actions of the ClassEdge class.

only an *association* and *inheritance* edges, as depicted in figure 9 between "Class_A" and "Class_B", and "Class_A" and "Class_C" respectively. This class is thus responsible for drawing the visual concrete syntax of an edge on canvas. Moreover, it listens to and handles all user interaction events on the element accordingly.



These interaction events and their actions are summarized in table 2.

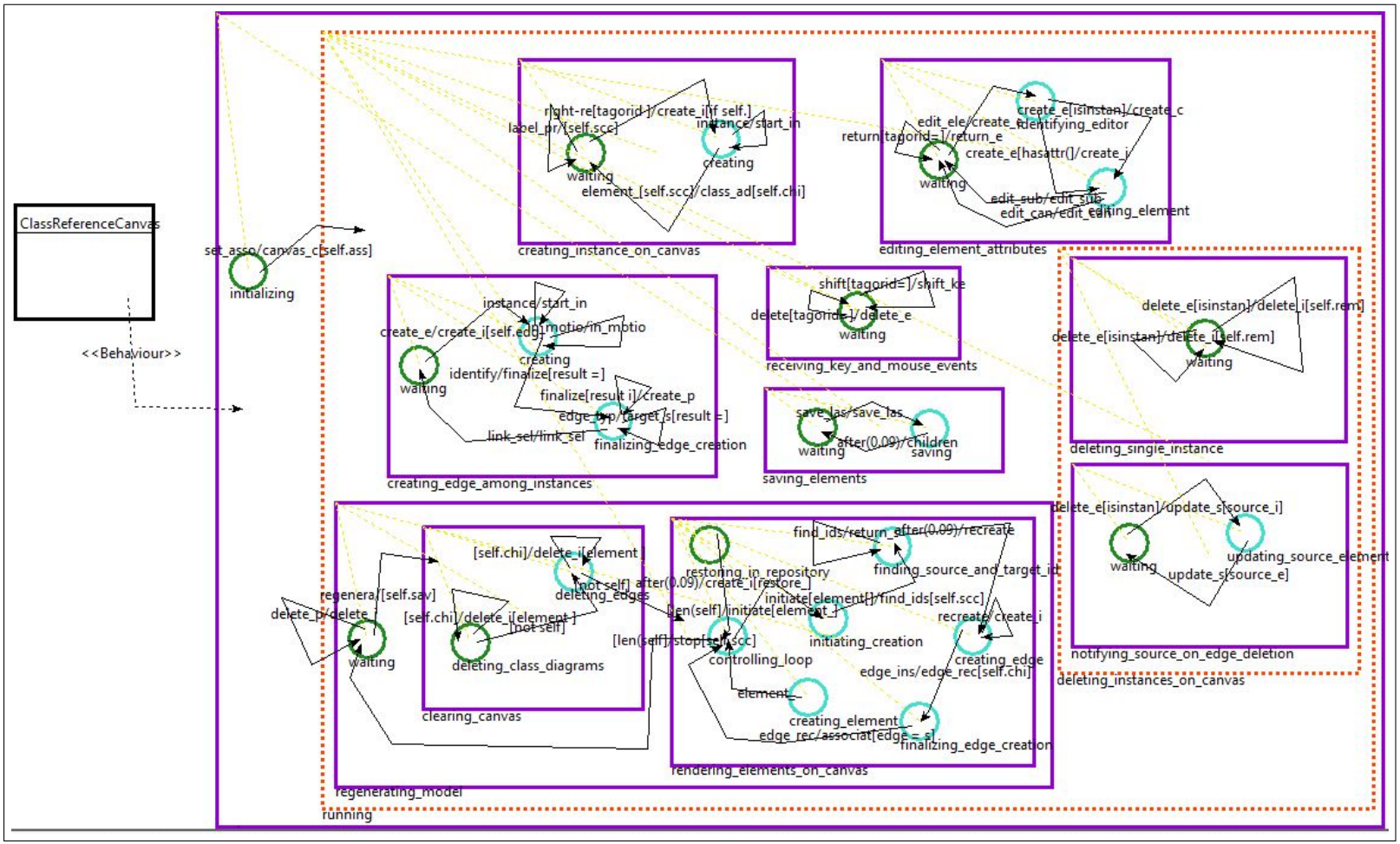


Figure 10: **The behaviour of *ClassReferenceCanvas* class modelled in the SCCD editor.**

- *Element editor classes*: The modeling constructs in the *class-reference* editor include only Class-diagrams and edges connecting them. Hence, these classes, such as the *ClassEditor*, *InheritanceEditor*, etc. create a "pop-up" window for editing, validating and submitting an element's attributes. For example, in figure 11 we can see a "pop-up" window generated by the class *AssociationEditor* for editing an association edge attributes such as its name and cardinality attributes.

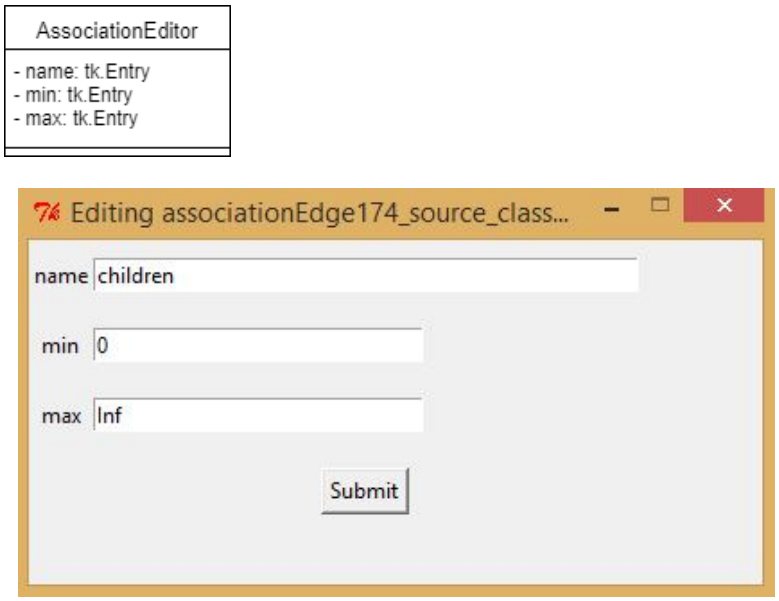
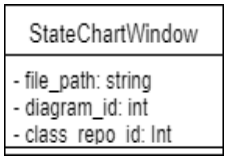


Figure 11: A "pop-up" window for editing an association edge.

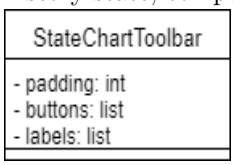
4.1.2. Class-behaviour (Statecharts) editor

The class-behaviour (Statecharts) editor provides an environment for modelling the behaviour of a selected class and has all the Statecharts constructs of SCCD. Hence, the main classes involved and their responsibilities are briefly discussed as follows:

- *StatechartsWindow*: Similar to other window classes, this class provides the role of a window widget where menu items and a canvas can be created and contained in, for the class-behaviour editor as shown in figure 12. Moreover, it receives and handles toolbar menu events accordingly, and creates the corresponding "pop-up" window for editing the attributes of a selected Statecharts modeling construct.



- *StatechartsToolbar*: The main purpose of this class is providing a toolbar for menu items that include buttons and labels as shown in figure 12. Buttons include icons for operations such as saving and validating a Statecharts model. Labels are used to select which type of Statecharts modelling construct to use on canvas, and include icons for a basic state, history state, composite state and a parallel state.



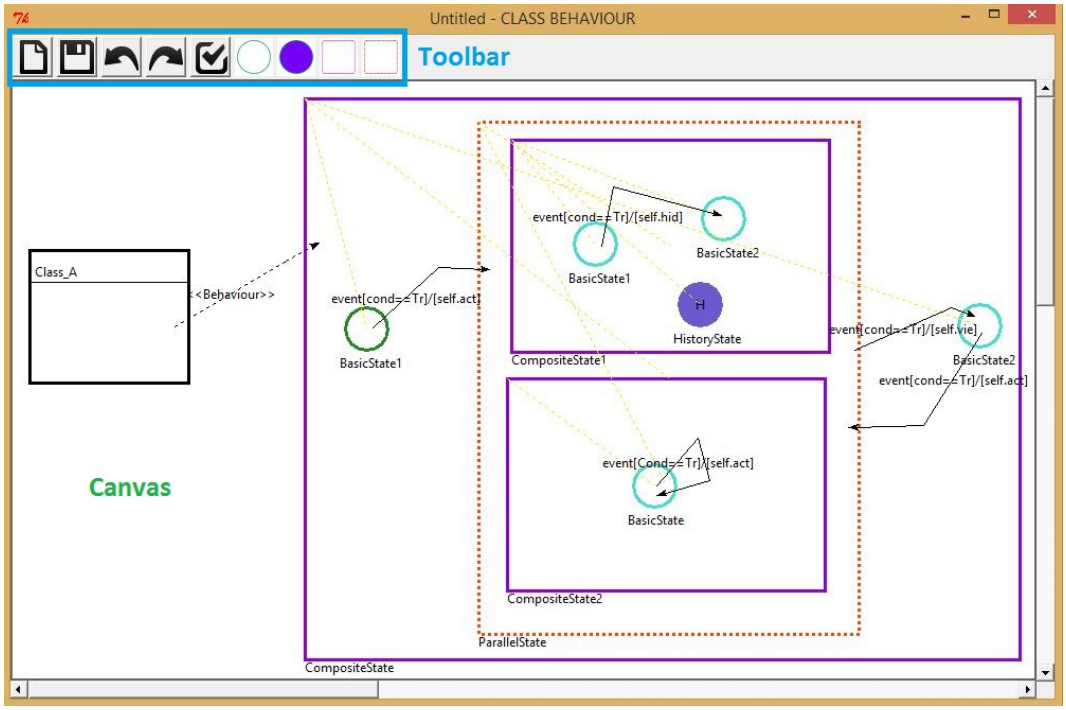
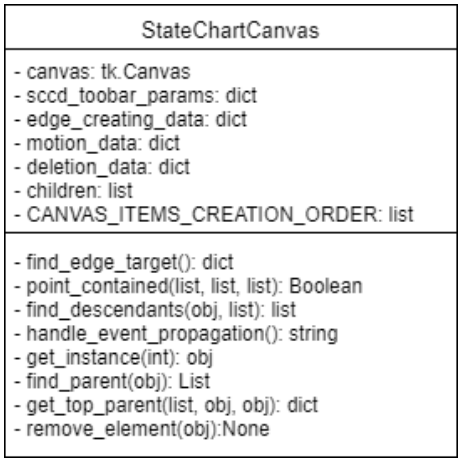


Figure 12: A simple model showing the User-interface of the Class-behaviour (Statecharts) editor of "Class_A" in figure 9.

- *StatechartsCanvas*: This class provides a canvas environment for creating/deleting Statecharts elements and transition edges connecting them.

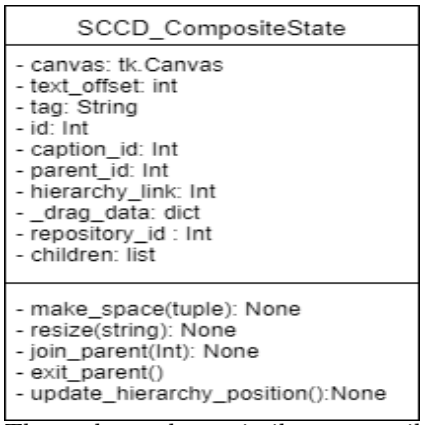


It is also responsible for re-generating a saved class-behaviour (Statecharts) model on canvas. Similar to the class-reference editor, while handling the save button, this class receives a request to save location from *StatechartsWindow* class and sends a "broadcast" message to all elements on canvas to store their location. In addition, since this class owns the keyboard focus, keyboard events are sent to this class and subsequently broadcasted to all elements on canvas so that the element on "selected" state will only act up on the received keyboard event. Moreover, in support of Statecharts's hierarchical needs, this class provides the necessary functions in finding the parent element (if any) for an element being dragged. Likewise, when an element is looking to create a transition edge, this class takes the responsibility of identifying the target class (if any). Essentially, this class also serves as a message exchange platform between hierarchical elements on canvas. This is because SCCD has an *associate_instances* event to create an association of a 'parent-children' relationship at run-time, which can in turn serve as a message exchanging mechanism. However, the parent-children relationship can be temporary since a child can leave its parent, thereby making SCCD's *associate_instances* event impossible to use. Thus, an event to "dissociate_instances" at run-time was proposed and implemented as an enhancement of SCCD to meet such requirements.

- *Statecharts element class*: This class can be any one of the classes representing the Statecharts modeling constructs, i.e. *SCCD_State*, *SCCD_HistoryState*, *SCCD_CompositeState*, *SCCD_ParallelState*.

Event ID	Event	Action
1	left-click	Select
2	1 + move	Drag start
3	2 +left-release	Drag end
4	right-click + move	Start edge creation
5	4 + left-click	Create edge control-points
6	4 or 5 + right-release	End edge creation
7	1 + return key	Display attribute editor

Table 3: User interaction events and actions of any one of the Statecharts element classes.



These classes have similar responsibilities which mainly include drawing the visual concrete syntax of the corresponding SCCD element on canvas, as depicted in figure 12, and listening to and handling all possible user interaction events on the element accordingly. These common interactions are summarized in table 3. The visual concrete syntax for the Statecharts elements look slightly different than Harel's first representation. For example, composite and parallel states are not visualized by a *rounded* rectangle since the developed editor uses Tkinter, a graphical user interface development toolkit, and it does not provide an interface for creating rounded rectangles. Similarly for inheritance edges, Tkinter does not provide an option for leaving line arrowheads unfilled. Inheritance edges in this editor are visualized with a thick line instead. As an alternative choice, initial states in the editor are also visualized by highlighting the border of the initial state in green color. Moreover, each class invokes the necessary CRUD operational calls of the *Repository API* to keep its information updated in the repository. In support of Statecharts hierarchical needs, each class listens to its parent to update its position on canvas when the parent moves, or deletes itself when the parent get deleted. Furthermore, *SCCD_CompositeState* and *SCCD_ParallelState* classes have additional roles of being up to date of children contained in them and managing their requested services such as enlarging their size to accommodate for

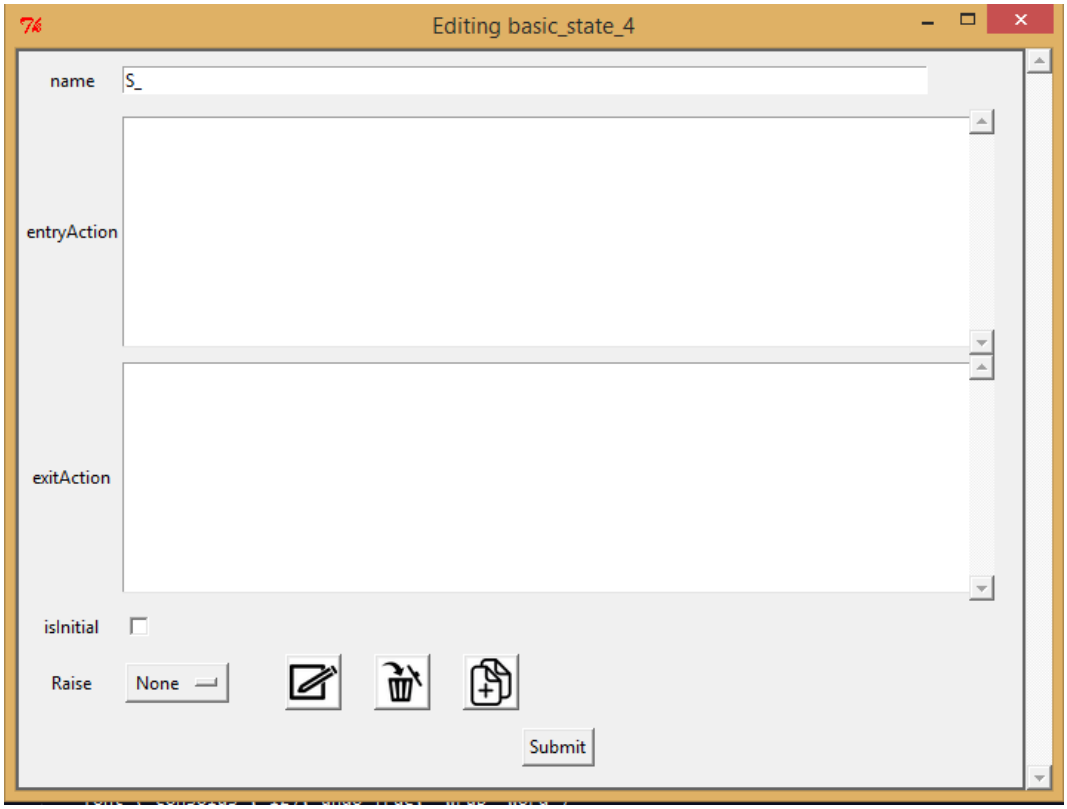


Figure 13: A "pop-up" window generated for editing state attributes.

a child. They also listen to and handle user interaction events to allow manual resizing.

- *Element editing class* : These classes can be any one of the window classes used for editing the attributes of a Statecharts modeling construct , such as *StateEditor*, *TransitionEditor*, etc. Thus, the editor class creates a "pop-up" window for editing, validating and submitting its corresponding element properties. Figure 13 depicts a "pop-up" window generated by *StateEditor* class for editing attributes of the initial state in Figure 12.

- *SCCD_ClassDiagram and ClassEdge*: The *class-behaviour* editor also provides a way to access and edit the selected Class-diagram proprieties. Thus, these classes provides similar roles as mentioned in the *class-reference* editor, except the edge type used in the class *ClassEdge* is a "behaviour" one connecting the Class-diagram to its Statecharts, as depicted in figure 12.

4.1.3. Simulation classes

The Simulation classes included in the front-end module are used by an external Statecharts simulation engine and provide an interface for communication. These classes can receive certain parameters from the simulation engine, such as the current state of a given class instance at run-time, so as to create a "pop-up" window (if not already created) which displays the class's Statecharts model and also highlight the current state in the model. Furthermore, the simulation classes can also display a menu consisting of the various debugging operations supported by the simulation engine and send to it the input parameters received while a user selects one of the menu options. Hence, the main classes and their responsibilities are briefly discussed as follows:

- *SimulationToolbarWindow*: This class provides a "floating" toolbar window, as shown in figure 14, and is responsible for creating menu items contained in it. The menu items are buttons representing the various debugging operations supported by the external simulation engine. Figure 15 depicts the behaviour of this class modelled in the developed visual SCCD editor. The three main behaviours of this class are represented by the three composite states contained in the "running" orthogonal component. Thus, it has the main role of listening to and handling user interaction events on the menu items. Concurrently, it listens to incoming parameters, such as a list of state-reference-ids (see section 4.3) of the "current states" of a specified class instance at run-time, from the simulation engine. This interaction of objects at run-time for highlighting a state of a given class is depicted in a sequence diagram in figure 17. In addition, it manages the creation and deletion of windows used for displaying the simulation; and receive user inputs as a result of handling events of menu items in the toolbar shown in figure 14.

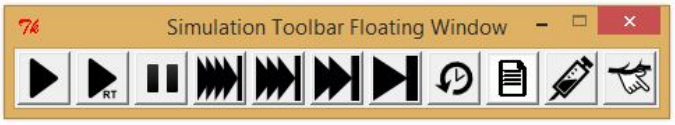
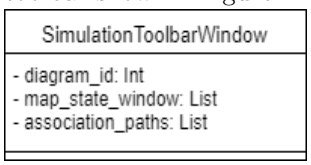
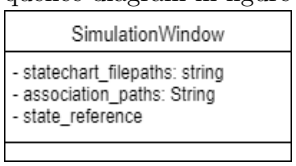


Figure 14: Menu items of a simulation toolbar "floating" window.



- *SimulationWindow*: This class creates a canvas environment used for simulation and contains it in, as depicted in figure 16. In addition, it receives simulation parameters from the *SimulationToolbarWindow* and passes it

on to the canvas for exhibiting the simulation effects, as shown in a sequence diagram in figure 17.



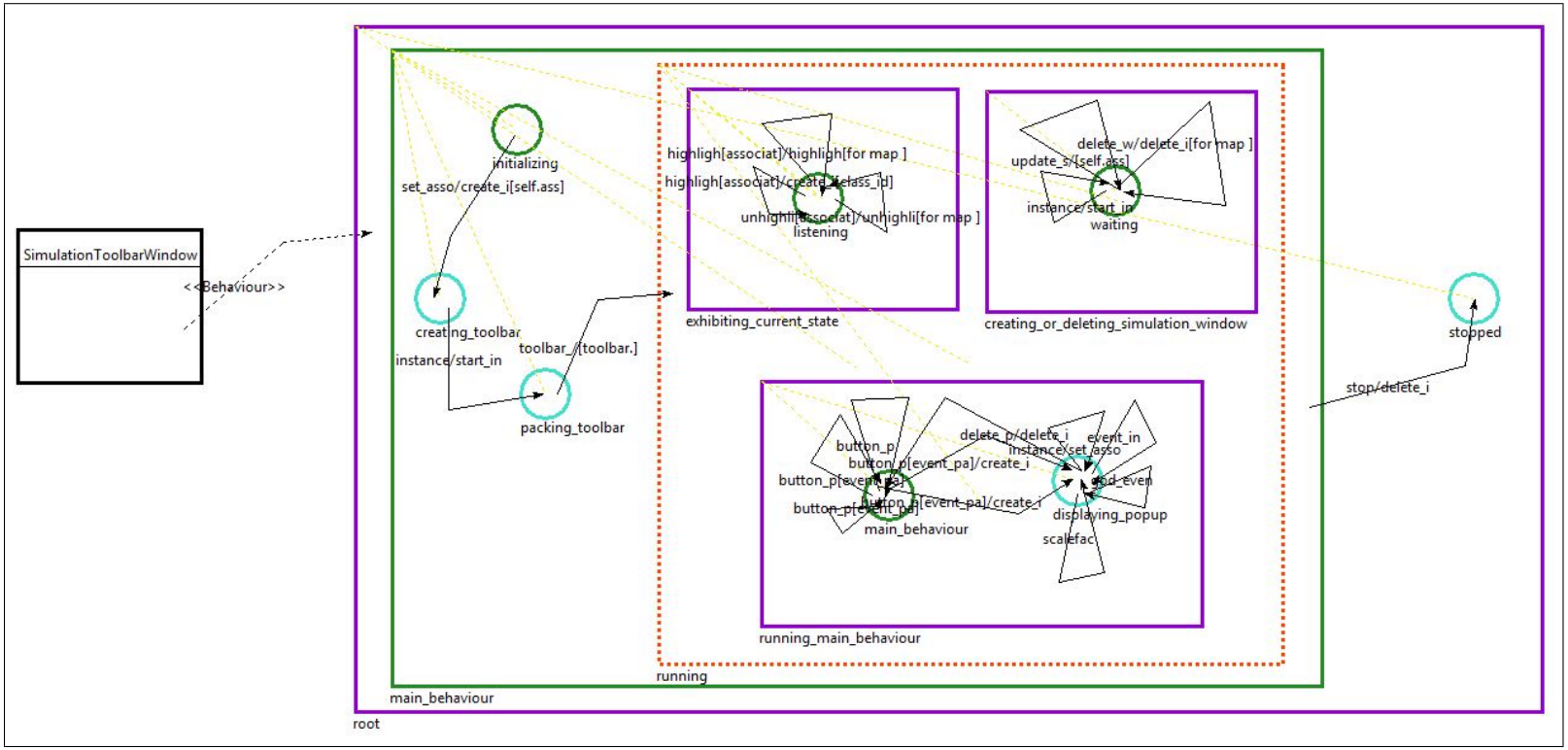
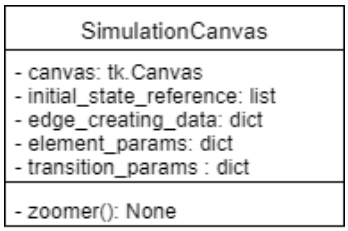
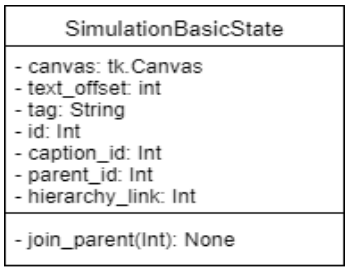


Figure 15: **The behaviour of *SimulationToolbarWindow* class modelled in SCCD.**

- *SimulationCanvas* : This classes uses the simulation parameters it receives from the *SimulationWindow* class and generates the corresponding Statecharts model for the selected class instance. It also broadcasts the received parameters to all Statecharts elements on canvas so as to "highlight" and/or "un-highlight" themselves as a result of the simulation effect, as depicted in a sequence diagram in figure 17.



- *Simulation Statecharts classes*: These classes represent the Statecharts constructs used by the *SimulationCanvas* class to create a Statecharts model for a given class, and includes classes such as *SimulationBasicState*, *SimulationHistoryState*, *SimulationCompositeState*, *SimulationParallelState*, and *SimulationTransition*. Thus, besides drawing the corresponding visual concrete syntax of the Statecharts element, the only role of any of these classes is to listen to the simulation parameters received from the canvas and evaluate itself to be the state for "highlighting" or "un-highlighting" itself. Figure 16 exhibits the Statecharts of an SCCD_CompositeState class instance and shows the initial states as the current states highlighted with a green colour.



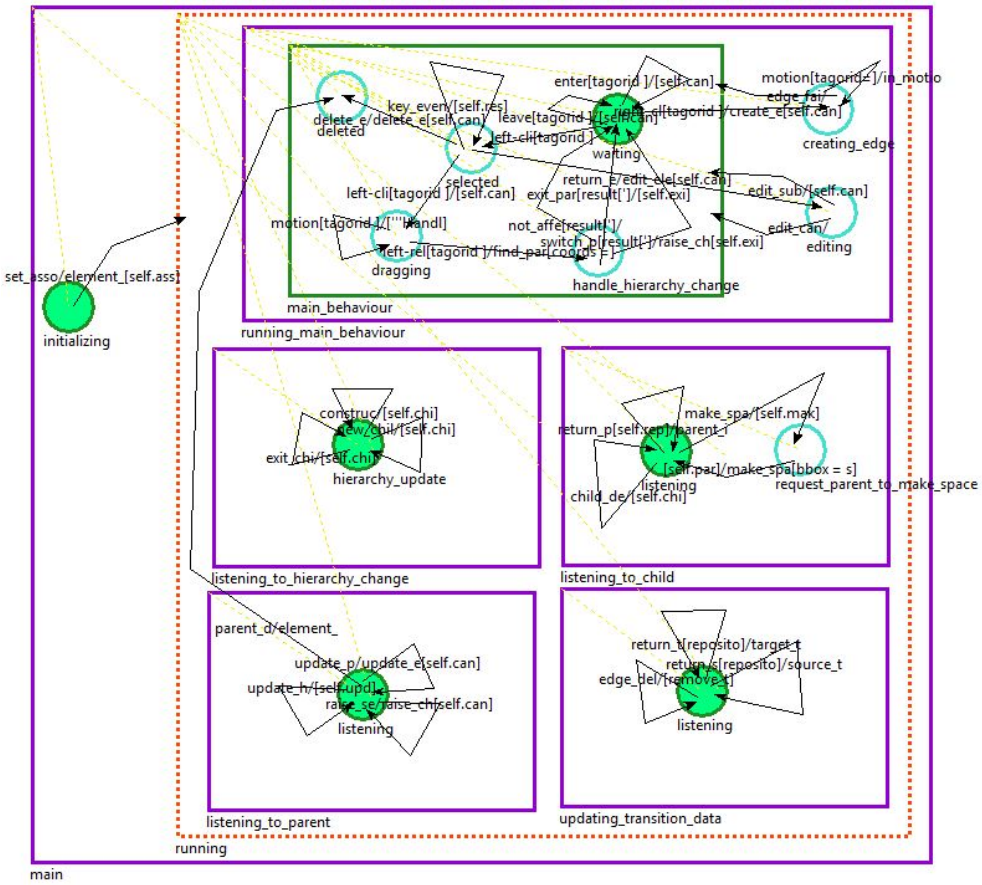


Figure 16: A simulation snapshot of an SCCD_CompositeState class with its current (initial) states highlighted in green colour.

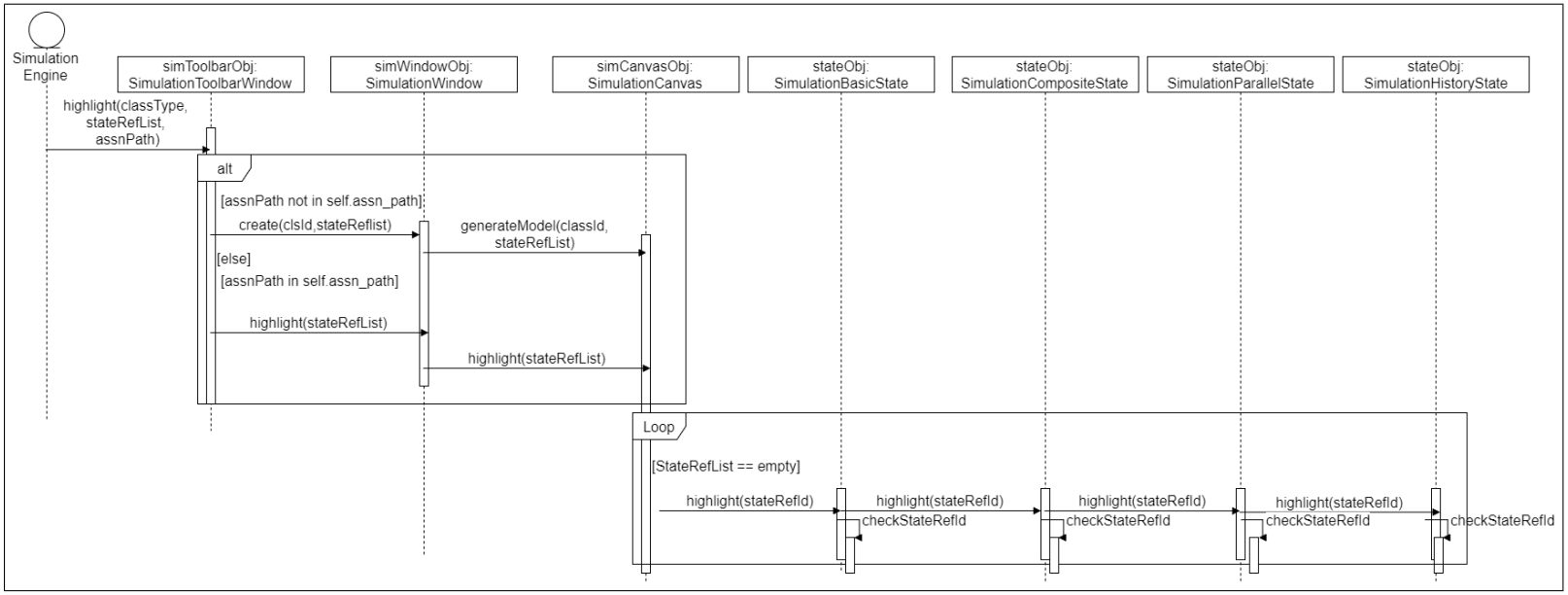


Figure 17: A sequence diagram depicting object interaction at run-time for "highlighting" a state in Statecharts model of a given class instance during simulation.

4.1.4. External classes

The application front-end uses external classes/libraries to implement its user interface requirements. These libraries include mainly widgets from Python's Tkinter[6] package, which is a GUI (Graphical User Interface) programming toolkit. Some of these classes are briefly discussed as follows:

- *tk.Toplevel*: This class serves as a window widget and is used to display extra application windows, dialogs, and other "pop-up" windows.
- *tk.Frame*: Frame widgets are used to group other widgets into complex layouts. They are also used for padding, and as a base class when implementing compound widgets, such as the toolbar used in this application.
- *tk.Label*: The Label widget is a standard Tkinter widget used to display a text or image on the screen.
- *tk.Button*: The Button widget is a standard Tkinter widget used to implement various kinds of buttons. As used in the toolbar of the application, buttons can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.
- *Widget*: This class does not reside in Python's Tkinter package and is a class constructed for the purpose of translating UI events to Statecharts events.

4.2. Model Repository API

This python module created includes a number of functions used to access the model repository and serves the purpose of an interface for CRUD operations of model information in a repository. A local (in memory) data structure is created as a model repository for the editor. However, this should in future be replaced by the Modelverse [7], which is a meta-modelling framework and model repository.

4.3. Code Generator

This module is a python class used for transforming the Visual SCCD models to SCCDXML concrete syntax. This operation is triggered when a button named "export_to_sccdxml", in the class-reference editor toolbar, is clicked. Thus, this exporting class receives a diagram id and makes use of it to retrieve the corresponding model data, via the repository API, to produce SCCDXML code. Ultimately, by leveraging the existing compiler for SCCDXML- in different platforms, executable code can be generated for running an application modelled in the SCCD visual editor.
While generating the SCCDXML code, this exporting class is also responsible for resolving state-reference-id of all Statecharts modelling instances. State-referencing-id is the location path in a string format consisting of state identifiers separated by forward slashes (/), presenting the hierarchy to be traversed

before reaching a given state. Thus, this can be useful for identifying the target class of a transition, and also for exhibiting simulation effects where state instances on the simulation canvas evaluate their state-references against a list of *current states* (also in state-reference format) received as a parameter from the simulation engine.

5. Bootstrapping

This procedure involves creating an SCCD model, that describes the behaviour of the modeling environment, in the visual editor, which in turn regenerates the environment's behaviour from within itself. Figure 18 depicts a top-level class-reference model of the modelling environment, where each class, except the external classes shown in dotted-lines, has a Statecharts model.

Bootstrapping the developed visual editor can subsequently serve as a sub-

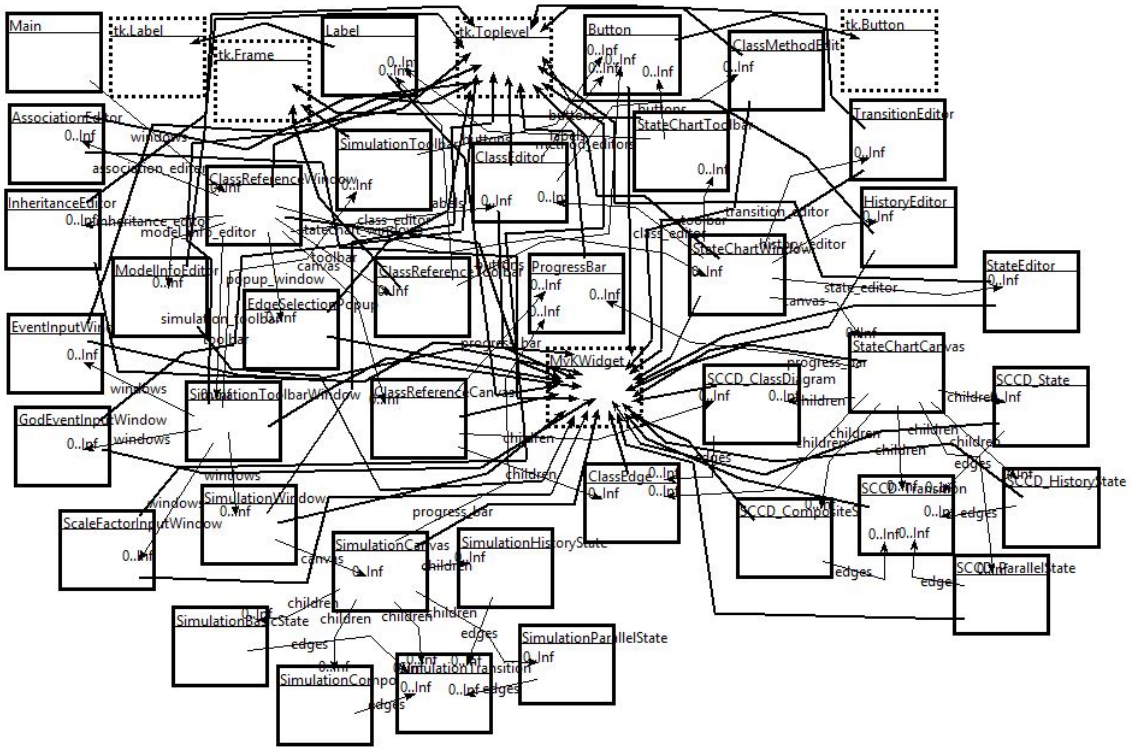


Figure 18: This figure shows the top-level class-reference model of bootstrapping the modelling environment.

stantial 'stress test' for the modelling environment and allows to claim that its

functionality is sufficient. Furthermore, besides serving as a visual documentation, it can also be used for extensibility and maintenance of the developed application.
The bootstrapping procedure resulted in a positive outcome where all models on canvas consisting of classes and their corresponding Statecharts were able to be precisely transformed to SCCDXML model structure. Thus, using the existing compiler for SCCDXML- python platform, executable code was successfully generated that modelled and created the modelling environment within itself.

6. Conclusion

In this research internship, a visual modelling and simulation environment for the SCCD formalism has been developed and discussed.The SCCD formalism combines Harel's Statecharts with UML's Class-diagram and allows users to model complex, timed, autonomous, reactive, and dynamic-structure systems. A concrete textual syntax, i.e. the SCCDXML notation, was previously developed for representing SCCD models and a compiler was also created that generates executable code from SCCDXML models for different platforms.
Thus, the output of this work brings together the visual concrete syntax of UML's Class-diagram with Harel's Statecharts and provides a user-friendly environment for modelling and simulation of SCCD models. The editor also supports for code generation, i.e. transforming SCCD models to their corresponding SCCDXML notation which can then generate executable code. The developed visual editor is also bootstrapped, by creating and modelling the editor and its UI behaviour within itself.

References

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

[2] J. Hopcroft, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[3] OMG, UML 2.0 superstructure specification, Tech. rep., Object Management Group (2005).

[4] S. Van Mierlo, Y. Van Tendeloo, B. Meyers, J. Exelmans, H. Vangheluwe, SCCD: SCXML extended with class diagrams, in: 3rd Workshop on Engineering Interactive Systems with SCXML, part of EICS 2016, 2016.

[5] S. Van Mierlo, SCCD documentation.
URL `https://msdl.uantwerpen.be/documentation/SCCD/`

[6] Tkinter - Python interface to Tcl/Tk.
URL `https://docs.python.org/2/library/tkinter.html, Visited on 2017-02-16`

[7] Y. Van Tendeloo, B. Barroca, S. Van Mierlo, H. Vangheluwe, Modelverse specification.
URL `https://msdl.uantwerpen.be/documentation/modelverse/`