University of Antwerp
Antwerp, August 2021

Model-Based Development of a Modelling and Simulation Environment for the Statecharts and Class Diagrams (SCCD) Formalism.

Sylvain Elias
Supervisor: Prof. Hans Vangheluwe

Master thesis, Software Engineering
Major: Computer Science

University of Antwerp

Abstract

Statecharts is a formalism that allows the creation of complex event and time-driven reactive systems. Multiple tools have been developed to allow the creation of such statecharts, and each of them handles the design of more complex multi-state machine systems differently.
In this paper, we will first start by exploring how two different statecharts formalisms can be used to develop such complex systems: Yakindu and SCCD. The system to be developed involves multiple statechart objects functioning independently, but also interacting with each other to have one complete, complex reactive program. At first, I will explain the system to be developed with its requirements and its different challenges. Then I will go in detail into its development using the two tools previously mentioned, and will discuss how well each of them handles the different requirements.
After comparing both tools, I will take the tool that is better at handling such systems and improve on it in order to comply with all of such systems' requirements. Therefore, SCCD will be used to create a visual editor and simulator for itself. I will explain the reasoning behind why state machines, and most importantly SCCD, are perfect for the design of a graphical user interface. Then I will go through the implementation of this system and the decisions taken during the process.
The contributions of this thesis are then twofold. The primary contribution is the design, description, and implementation of the combination of statecharts and class diagrams, as defined by SCCD, of a graphical user interface that allows to develop and edit SCCD systems. The second contribution is the explanation and implementation of a simulation environment that generates a modified instance of the system developed using SCCD, and connects it with the visual editor. This will show the changes in the states and data of its different objects on the user interface, and will allow the developer to interact with the simulated system by directly raising events to it and watching their effects.

Nederlandstalige Samenvatting

Statecharts is een formalisme dat de creatie van complexe event en time-driven reactieve systemen mogelijk maakt. Meerdere tools zijn ontwikkeld om dergelijke statecharts te maken, en elke tool behandelt het ontwerp van meerdere complexe multi-state machine systemen op een andere manier.
In deze paper zullen we eerst onderzoeken hoe twee verschillende statecharts formalismen kunnen gebruikt worden om dergelijke complexe systemen te ontwikkelen: Yakindu en SCCD. Het te ontwikkelen systeem omvat meerdere statechart objecten die onafhankelijk van elkaar functioneren, maar ook met elkaar samenwerken en die samen één compleet, complex en reactief programma maken. Eerst zal ik het te ontwikkelen systeem uitleggen met zijn vereisten en zijn verschillende uitdagingen. Daarna zal ik in detail ingaan op de ontwikkeling ervan met behulp van de twee eerder genoemde tools, en zal ik bespreken hoe goed elk van hen omgaat met de verschillende vereisten.
Na vergelijking van beide tools zal ik de tool kiezen die beter omgaat met dergelijke systemen en deze verbeteren om te voldoen aan alle vereisten van dergelijke systemen. SCCD zal hierdoor gebruikt worden om een visuele editor en simulator voor zichzelf te maken. Ik zal uitleggen waarom toestandsmachines, en vooral SCCD, perfect zijn voor het ontwerp van een grafische gebruikersinterface. Daarna zal ik de implementatie van dit systeem, en de beslissingen die genomen zijn tijdens het proces doornemen.
De bijdragen van dit proefschrift kunnen tweeledig opgesplitst worden. De eerste bijdrage is het ontwerp, de beschrijving en de implementatie van de combinatie van statecharts en klassendiagrammen, zoals gedefinieerd door SCCD, van een grafische gebruikersinterface. Deze implementatie maakt het mogelijk SCCD-systemen te ontwikkelen en te bewerken. De tweede bijdrage is de uitleg en implementatie van een simulatieomgeving die een aangepaste instantie van het met SCCD ontwikkelde systeem genereert, en deze verbindt met de visuele editor. Dit toont de veranderingen in de toestanden en gegevens van de verschillende objecten, en stelt de ontwikkelaar in staat het gesimuleerde systeem te interageren door er direct gebeurtenissen aan toe te voegen en de effecten ervan te bekijken.

Acknowledgements

I would like to thank my supervisor Hans Vangheluwe. Without his advice and encouragement I would never have realized my potential to make a contribution to domain-specific visual modeling. A special thanks to my friend, Edward Nauwelaerts, who helped me translate my thesis abstract into Dutch. Finally, a big thank you to my mom, without her constant encouragement and support I might not have studied for so long, let alone completed this thesis.

Contents

List of Figures

1 Introduction

A number of new systems that we analyze and develop these days are getting more complex with the interaction between the physical components and the real-time software, and all the data that these systems have to process. Most of the times, these complex systems exhibit some form of reactivity to the environment: the system reacts to input events coming from the environment by changing its internal state, it can also influence the environment through output events that it emits in response to data changes or events received. These reactive systems run continuously, and often with multiple components executing in parallel.
Statecharts is an advanced construct that allows the creation of such complex event and time driven systems. Multiple tools have been created around the statecharts formalisms with each having their own additional features and different ways of implementing them, some of these tools include Yakindu and SCCD which will be the two main tools used in the rest of this paper.
We will start by explaining what statecharts are and how they function, then we will dive deeper into Yakindu and SCCD by using both tools to develop a complex railcar system. After having explored and compared both tools to each other, we will focus on SCCD to enhance its features in order to better handle the creation and testing of the systems. We will create a graphical user interface for it and a simulator for the systems developed using it in order to add all the functionalities missing from this tool. We will then recreate the same railcar system using this GUI and verify that it now complies with all the requirements of the complex systems. Finally we explore related work and conclude the paper.

1.1 Background

1.1.1 State machines for system development

Statecharts, as defined by D. Harel in [10], are mainly created to help in the development of complex systems. In [22] and [10], a simple example of a digital watch is used to demonstrate how state machines can help with the development of time and event based reactive systems; and in [9], Harel takes a on a more complex example of a railcar system and demonstrates how its development can be completed using statecharts.

1.1.2 State machines for user interface development

Statecharts and other state machine formalisms have been used in Human Computer Interface research for a long time. In [19], the “Ultimate Hook” is described as a Graphical User Interface (GUI) design pattern where events bubble from inner GUI components to outer GUI components, which is directly supported by the Statecharts syntax of its hierarchical states.
State machines have also been used to directly incorporate working user interfaces and facilitating the rapid prototyping of UI behaviour during development [1], [4].

1.2 Contributions

The first contribution of this thesis is the use of the combination of class diagrams and statecharts, which is done using SCCD, to create a graphical user interface that will be used to develop other SCCD systems. Therefore, SCCD will be used to create the SCCD graphical editor.
The second contribution is the addition of a simulator to this GUI that can directly simulate the system under development.

2 Definitions

2.1 Statecharts

Statecharts are a formalism used to specify the behaviour of timed, autonomous, and reactive systems using a discrete-event abstraction. They extend Timed Finite State Automata with depth, orthogonality, broadcast communication, and history [11].
Statecharts were invented in 1987 by David Harel [10], and they are created using the following building blocks:

2.1.1 States

States model the mode a system is in. When orthogonal regions aren't added to the system, only one state can be active at any point in time of the system's execution.
A state is identified by a unique name, and it executes different actions when it is entered and when it is exited. Exactly one state in the model is set at the initial state which will initialize the system to it on start-up.
The actions that the state can execute are:

- Raise events inside of the system, either to the same component or to the others.
- Perform computations and assignments on the system's variables.

A state is visually represented by a rectangle with round corners

2.1.2 Transitions

Transitions connect two states together: the source state and the target state. When the system is running and events are being received and the time is passing, the transitions can trigger if their conditions are met, which leads to a change in the current state of the system from the source state to the target state of the transition.
The triggering conditions of the transitions contain the following optional elements:

- *Event*, which, when received by the system, triggers the transition if its other conditions are also met.
- *Time*, usually noted as 'after X time', which fires the trigger after the time specified has passed since the origin state of the transition has been entered. Only an *event* or a *time* can be added to the transition at a time, not a combination of both.
- *Condition* which is evaluated whenever the transition triggers, if the condition is met then the system changes its state to the target state of the transition, otherwise no change happens.

If a transition has no elements then it said to be spontaneous, where the system enters the origin state, executes its actions, then immediately changes its active state to the target of the transition (in Yakindu, a spontaneous transition is modelled by adding the 'always' event to it).
A transition can also perform actions when it is triggered.
Fig. 2.1 shows an example of two states and two transitions between them created using Yakindu:

Figure 2.1: States and transitions example

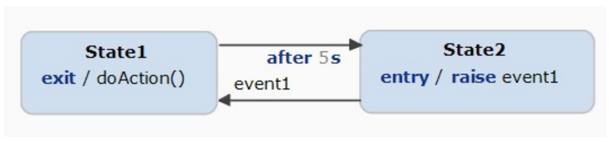
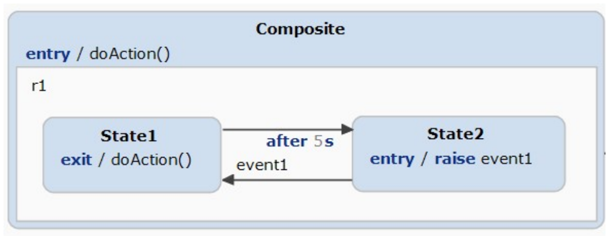


Figure 2.2: Composite states example



- State1 performs the doAction function when it is exited.
- State2 raises the event1 event when it is entered.
- The transition with State1 as origin and State2 as target is triggered after 5 seconds of entering State1.
- The transition with State2 as origin and State1 as target is triggered when the system receives the event1 event (which is raised directly when State2 is entered).

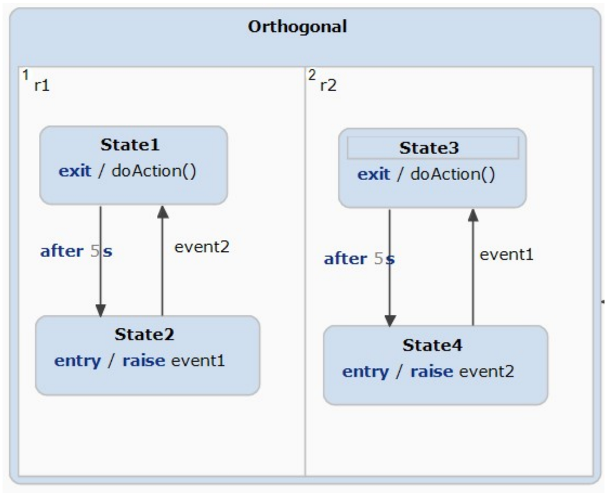
2.1.3 Composite states

A composite states is a collection of sub-states, which themselves can be either normal states or composite states as well. This allows to nest states to arbitrary depths. The main purpose of these composite states is to group the behaviours that logically belong together. Similarly to the statechart model, composite states also have one initial state that gets set to active when the composite states is entered.
Composite states are also connected through transitions and they can perform actions when entered and exited the same way normal states do.
Fig. 2.2 shows an example of State1 and State2 as children of the composite state 'Composite' using Yakindu. Composite also performs the doAction function when it is entered.

2.1.4 Orthogonal regions

Other than a hierarchical combination of the states in the system using composite states, these states can also be combined orthogonally in the orthogonal regions.
Each region in the orthogonal regions has its own initial state and, when these orthogonal states are added to the system, multiple states can be active at the same time as long as these states are all elements of the different regions in the same orthogonal regions.
Each region can also perform actions on entry or on exit similarly to the normal states.
Fig. 2.3 is an example of four states added to the orthogonal regions 'r1' and 'r2' of 'composite'. The system will have two states active at the same time where one of them is in r1 and the other is in r2.

Figure 2.3: Orthogonal regions example



2.1.5 History

A history state is a state that can be only placed inside of a composite states or inside of the regions of the orthogonal regions. It remembers the current state the composite states is in before it is exited and allows to directly enter that states when a transition that has the history as its target is triggered.
There are two types of history states:

- *Shallow history* which remembers the current state at its own level.
- *Deep history* which remembers the current states at its own level and all lower levels in the hierarchy.

Unlike the other elements of statecharts, history states do not perform any action.
Fig. 2.4 shows the same composite states example explained above but with a shallow history state added to it. In Yakindu, a history state also acts as an entry state hence why it has a transition exiting it. When the system exits the composite states, the history state memorizes the last active state inside of this hierarchy and, when the system enters this composite state again through the transition leading to the history state, the saved state will be the first one entered even if it isn't the initial state.

2.1.6 Entry and exit

Entries and exits are used mainly as a visual representation for the initial and final states of the system being modelled.
An entry with a transition to a state means that this state is the initial one in the system, and an exit with a transition to a state means that this state is the final one in the system. The entries and exits are also added inside of the composite states and orthogonal regions to specify their initial and final states.
Fig 2.5 shows the states State1 and State2 with State1 being the initial state of the system and state2 leading to the final state (or exit state) when event1 is fired.

Figure 2.4: History example

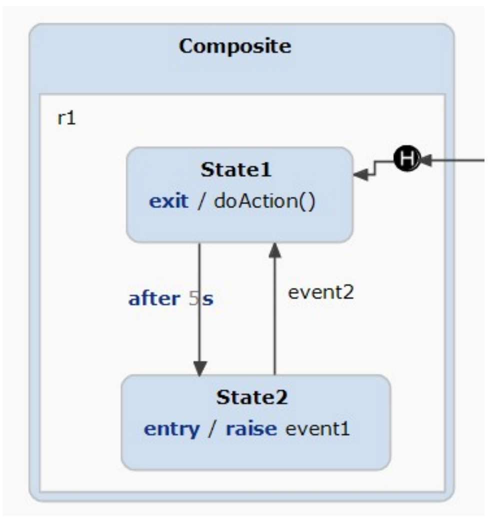
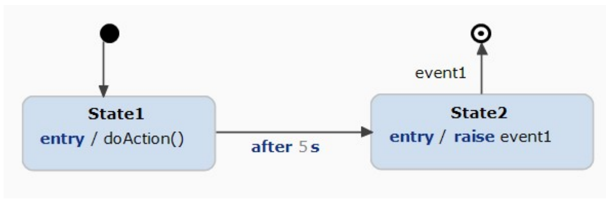


Figure 2.5: Entry and exit example



2.2 Yakindu

Yakindu is a commercial statechart tool that is used to create statechart systems using a visual graphical user interface [15]. Yakindu also allows to directly simulate the system while modeling it, it also offers code generators that translate the visual state machines into source code for a variety of target platforms.

2.3 SCCD

SCCD, as defined in [21], is a tool that combines:

- Statechart XML (SCXML): which is an XML-based markup language that is used to define state machine notations textually. It is a Statecharts variant developed by the W3C, and is described in a W3C working draft specification [2].
- Class diagram: which describes the structure of a system by defining the system's classes and the relationships between them. These class diagrams allow to extend SCXML with the dynamic creation of new object instances and the communication between them. In [17], similar features have been added to SCXML using a different approach.

The language generated from SCCD is called SCCXML. Therefore, SCCD is a non commercial textual statechart tool (containing its own language and compiler) that allows the creation of a state machine that is composed of multiple classes communicating with each other.

The following sections explain the SCCD formalisms that extend the normal statecharts behaviours as explained above.

2.3.1 Top-level element

The top-level element presents an input and output interface that allows the system to communicate with its environment similarly to the approach taken in UML-RT [20], it also allows to import libraries, and to hold a number of class definitions. Similarly to the statecharts, one of these classes is set to be the initial one with the *default* attribute and it is instantiated when the application is launched.

Listing 1: The SCCD top-level element.

```
<diagram>
  <top>
    from Tkinter import *
  </top>
  <inport name=`inport'/>
  <class name=`Class1' default=`true'>...</class>
  <class name=`Class2'>...</class>
  <class name=`Class3'>...</class>
</diagram>
```

Listing 1 shows an example of a top-level diagram. It imports the Tkinter library, it defines 'inport' to be one of its input ports which receives events from the external environment like when the user interacts with the UI, and four classes, explained in the following subsections.

2.3.2 Class elements

Classes are the main addition of the SCCD language. They model the structure of the system using class attributes and relations between the classes, and its behaviour using class methods which access and change the values of attributes of the class. Each class also has a user defined constructor and destructor which run when the an instance object of this class is created and deleted respectively.

Listing 2: The SCCD class element.

```
<class name=`Example'>
  <relationships>...</relationships>
  <constructor>...</constructor>
  <destructor>...</destructor>
  <method name=`method1'>
    <parameter name=`parameter1' />
    <body>...</body>
  </method>
  <scxml initial=`state1'>
    <state id=`state1'>...</state>
    <state id=`state2'>...</state>
  </scxml>
</class>
```

Listing 2 shows an example of a class element. It defines the relationships with other classes (discussed in the next subsection), a constructor and destructor, a method called 'method1' that takes in a parameter called 'parameter1', and a SCXML model that consists of two states.

2.3.3 Relationship elements

Relationships in SCCD are of two different kinds:

- *Associations* which are defined between a source and a target class, and they hold the following information:
  - *Name* of the association which is used by the class to raise events to other classes.
  - *Minimum* and *maximum* cardinality, representing how many instances of the target class have to be minimally an maximally associated to each instance of the source class.
- *Inheritance*, which allows the source of the relation to inherit all the attributes and methods from the target of the relation like normal inheritances of the object oriented programming languages.

Listing 3: The SCCD association element.

```
<class name=`Class1'>
  <relationships>
    <association name=`association1' class=`Class2'
        min=`1' max=`1'/>
    <association name=`Class3' class=`Button' />
    <inheritance class=`Toplevel' />
  </relationships>
   ...
</class>
```

Listing 3 shows the relationships of class1. It has an association to class2 where exactly one instance of that link has to exist between each two instances of these objects in the running application, and an association with class3 with no limits on the number of links that can be created. Class1 also inherits all the methods and attributes of the Toplevel class (which is a Tkinter class).

2.3.4 Events

Events in SCCD can be accompanied by a number of parameter values, where each parameter has a name that can be used as a local variable in the action associated with the transition that catches the event.
Since SCCD adds public input and output interfaces using ports, as well as classes and associations between them, event scoping becomes important in order to send the events to the correct targets. In traditional SCXML models, events are sensed by the statecharts model that generated them, but SCCD adds the ability to transmit events to other class

instances or to output ports, hence why the raise tag is extended with a scope attribute that can have any of the following values:

- *local*: The event is only visible to the same class instance that raised it. This is the default behaviour of SCCD.
- *broad*: The event is broadcasted to all the class instances of the system.
- *narrow*: The event is sent to a specific instance only.
- *cd*: The event is sent to and processed by the object manager (explained in the next subsection).

Listing 4: An example transition that narrow-casts an event.

```
<transition event=`event1' target=`.'>
  <raise event=`raised_event' scope=`narrow' target=``class2''>
    <param expr=`self.event_name' />
  </raise>
</transition>
```

Listing 4 shows an example of a transition which, when it fires after having received event1, raises the event 'raised_event' to another class named 'class2' in the association definition of this class.

2.3.5 Object manager

In SCCD, a central entity called the *object manager* is responsible for creating, deleting, and starting class instances. Instances can send events to the object manager using the *cd* scope as explained previously, these events can be:

- *create_instance*: Creates a new instance of the target class.
- *delete_instance*: Deletes the instances specified by the link reference.
- *start_instance*: Starts the execution of the statecharts model of the instances specified by the link reference.
- *associate_instance*: Creates an association between two existing instances

All of these events are executed if no multiplicity constraints are violated, and an event is sent back to the system in case of success or failure.

2.3.6 Runtime platforms

The semantics of SCCD are executed on a user chosen runtime platform, which provides the scheduling of the timed events used by the runtime kernel. SCCD presents the following three runtime platforms as shown in Fig. 2.6:

- *Threads*, where the platform runs one thread that manipulates a global event queue made thread-safe by locks, this allows external input to be interleaved with internally raised events.

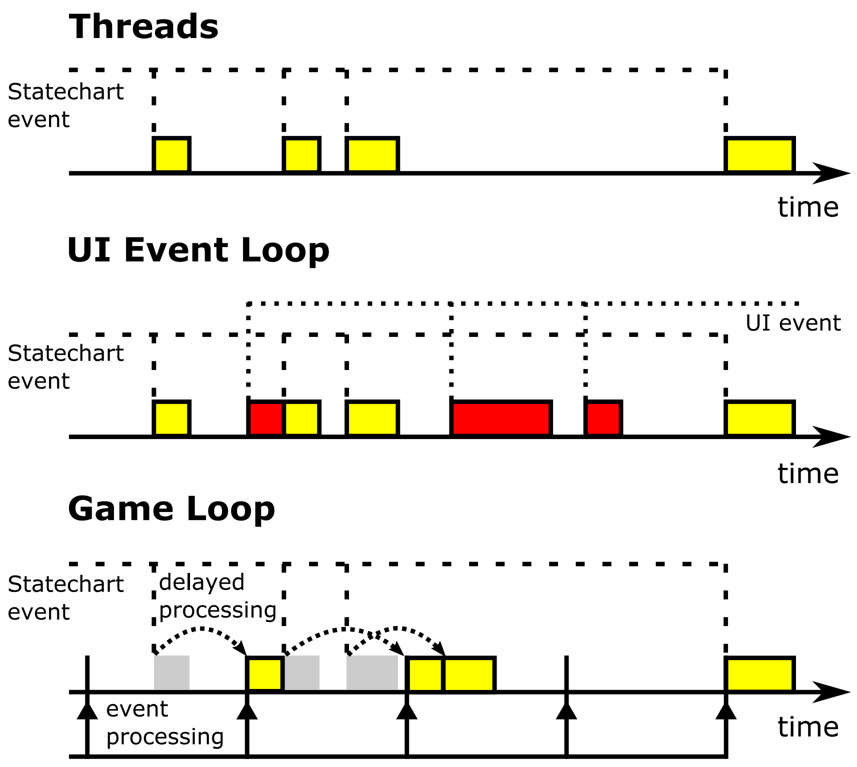


Figure 2.6: The three runtime platforms.

- *UI event loop*, where the platform reuses the event queue managed by an existing UI platform, such as Tkinter, which provides functions for managing time-outs for timed events, as well as pushing and popping items from the queue.
- *Game loop*, where the platform facilitates integration with game engines where objects are updated only at predefined points in time, decided upon by this engine.

2.4 Statecharts semantics

The basic structures of statecharts have not changed since their original definition [10]. but many of their semantics choices have been left undefined and each tool specified its own like in STATEMATE [12]. More recently, a study of big-step modelling languages like Statecharts has been performed in [6], and a set of semantic variation points, with which the different Statecharts execution semantics can be classified, was described.
In the development of the systems explained in the rest of this paper, similar semantics have been used in both Yakindu and SCCD so that the systems developed can act as close to each other as possible.

- In *Yakindu*, the *event driven* semantics are used with *SuperSteps* turned off. [13]
- In *SCCD*, the *big step maximality* semantics are used with *takeone*. [7]

2.5 Tkinter

Tkinter is the standard GUI library for Python [8], it allows the creation of user interfaces using the python programming language and Tk, which is a GUI toolkit for Tcl/Tk. This

library will be used in the rest of the paper for everything GUI development.
Some of the important widgets and features of Tkinter are:

- *Toplevel* widget, which provides a container that holds other widgets like the frame. When a tkinter GUI is initialized, the first shell that is created is of type *TopLevel* and is called the *root*.
- *Frame* widgets, which are containers for other widgets that are contained by the *TopLevel* widget. One of its most common uses is as a master for a group of other widgets.
- *Canvas* widgets, which are used to draw complex objects, using lines, ovals, polygons and rectangles, place images and any other *Tkinter* widgets.
- *Event binding* which is an important feature of *Tkinter* where the developer can bind the events received by the UI to custom functions that they have created. For example, the developer can bind the mouse click on the canvas to a custom function that sends this event to the controller of the SCCD system so that it can be used to fire statecharts transitions.

3 Railcar system

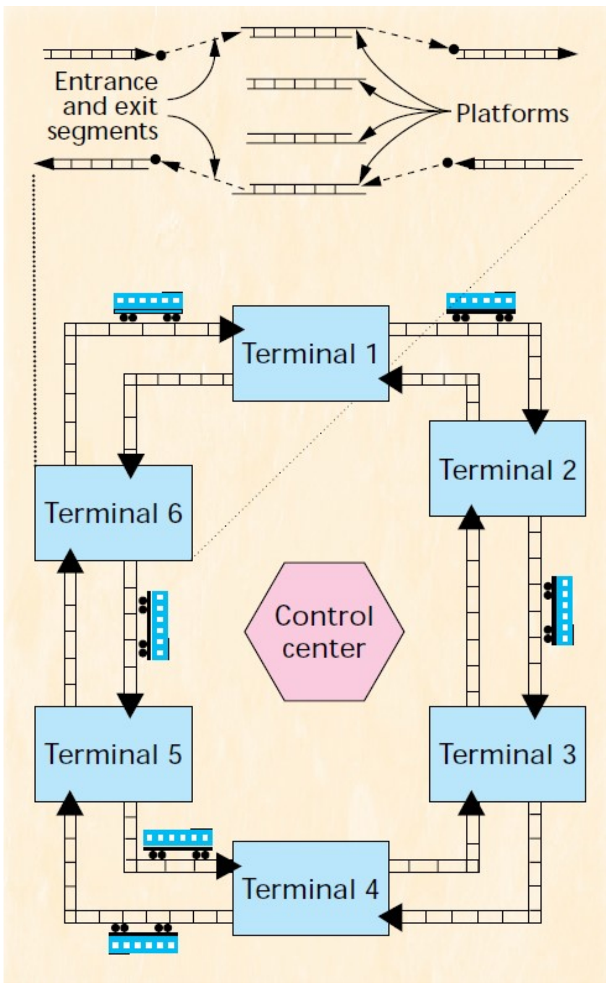
The previous explained what statecharts are, how they work, and the formalisms of the tools that will be used. This section will introduce the railcar system as defined by David Harel in [9] and go through its implementation using both Yakindu and SCCD tools. This section was a part of my research project but it contains essential information needed for the continuation of the thesis work. It is the first introduction to a complex system and its requirements and, the results of the comparison of the two systems will define the scope of the master thesis work.

3.1 Introduction

The railcar system presents the following requirements:

- The system has six terminals located on a cyclic path, these terminals are connected using rail tracks as shown in Figure 3.1 (for simplicity, only a single rail track between each two adjacent terminals is implemented, this rail can lead the cars only in one direction instead of the original two way connections).
- Several railcars are available to transport passengers between the terminals. The number of railcars doesn't have to be predefined, instead it can be given as an argument to the system which will then generate the corresponding number of cars.
- A control center acts as the center of the entire system. It receives, processes, and sends events to the various components of this system.
- Each terminal contains four parallel platforms that can be used as parking areas for the cars, therefore each terminal can park a maximum of four cars at a time.
- Cars are equipped with an engine and a cruise controller to maintain their speed.
- The cars can maintain a maximum speed unless there's another car within 80 yards. In this case, the car is to stop until there's enough space between it and the car in front of it.
- When the car is 100 yards away from a terminal, the system has to allocate it an entry segment and, in case the terminal is not the final destination, an exit segment. If the allocation is delayed, the car has to stop and wait for the allocation to be done before being able to continue its journey. This allocation is done using a car handler object that is created by the terminal when the car is 100 yards away from the it and is destroyed when the car passes it.
- When a car is to depart from a terminal, the terminal has to allocate it an outgoing track via its exit segment.
- The passenger in a terminal can choose their desired destination. When the destination is chosen, the control center assigns the closest car to this journey. The car can either be already parked in the same terminal as the passenger and take them directly to their final destination, or the car can be called from another terminal to pick the passenger up and take them to their destination.
- The system developed in this paper is only a simulation of the real world and will

Figure 3.1: Railcar System as defined by D. Harel



not be implemented in an actual railcar system. For this reason, the proximity sensor and the passengers have to also be simulated as individual objects.

The system has to be programmed in a way that is independent from the code generated from it. It should be possible to simulate the system directly using a statecharts simulator without the need for additional back-end software to connect its different components.

3.2 Multi state machine modeling

The railcar system explained above presents multiple state machines (or objects) that need to interact with each other. These different state machines can exist and function independently, but they can also communicate between each other in order to form a more complex system that can do more than just the basic tasks at hand.
The railcar system, as explained by Harel, consists of the following state machines: car, car handler, control center, cruiser, entrance, exit, exits manager, platform manager, and terminal.
A car can exist and function by itself, but it needs access to the cruiser in order to control the speed, the control center in order to get the information related to the passenger, etc. Therefore, these different objects need to communicate with each other in order to have a fully functional railcar system. It should be noted that it is possible to simulate the

system in a single state machine, but this system would be more complex to create than a multi state machine one, and it can only be used in the simulation and cannot be applied to a real world system where each component exists independently of the others.

3.3 Dynamic modeling

One of the requirements of the system is that whenever a car approaches a terminal, a $carHandler$ has to be created in order to bridge the communication between the terminal and the car. This $carHandler$ should also be deleted when the car passes the terminal to allow for the car and terminal to communicate with new objects.
The creation and deletion of these state machines is called dynamic modeling and it is one of the main parts studied in this paper.

3.4 Multi-state machine modeling in Yakindu

Yakindu has a built-in way of modeling multi-state machine systems. These systems help in the separation of concerns where each state machine has a specific task at hand, which, in turn, allows for a better software architecture. This feature is essential in our railcar system since it presents a big and complex system, and each one of its elements should be able to function separately and communicate with the other elements for a complete functional system.
Linking multiple statecharts into one system using Yakindu is as simple as importing the target state machine and defining it as a variable:

$$import : "Car.sct"$$
$$var\ car :\ Car$$

These parent state machines can easily communicate with their children by raising events, accessing their variables, listening to their out events, and checking their active states.
Sending an event to a child state machine can be done by raising that event in the child object through the parent: *raise car.detinationSelected*. The event that is being raised can also contain a variable to be sent to the child. In our system, this behaviour can be seen in the majority of its parts. For example, when the control center receives a request to transport a passenger, it sends the request to the car by raising the $car.destSelected$ event and giving it the terminal destination as a variable; the car can then read the destination value using $valueof(destSelected)$ and react to the event that it received.
In turn, the control center should be able to know when the car arrives to its final destination. This can be done in two ways:

- Since the control center is the parent in this specific case, then it can raise an event to the car object and add its own object instance as one of the event variables so the car can save it in its list of variables and directly raise events to it. But Yakindu doesn't present a way for an object to send its own instance to a child (a parent object can send an instance of a child object that is defined as one of its variables but it can't send an instance of itself). To solve this issue, we can have a single statechart object whose only purpose is to initialize all the different objects in the system and give each one of them references to each other before any of the actual events happen. This way, all the different objects of the system can have access to instances of the other objects and directly interact with them.

- Yakindu offers a direct way for the child to interact with its parent. Instead of directly raising an event in the target object, one can define the events as *out events* instead of the normal *in events* and raise them without specifying a target. These out events don't have an impact on the object that raised them, but they can be read by any other object that has a reference to it. So in our example, the car object would raise the *carArrived out event* which the control center parent can listen to by using *car.carArrived* as one of its transition events. This method is more efficient than the one previously mentioned, but it only allows direct interaction between the child and its parent. So, in our system, the car has no direct way of raising an event to a different car, but instead it has to raise it to the parent which is the control center. (a solution to this issue is discussed in 3.7)

For our system we used a combination of all the possibilities mentioned above:

- *out events* are used to raise events from an object to any other object that has a reference to it. These events can also be routed to be sent to a specific object if needed.
- The single statechart object that initiates all the objects approach is also implemented, but instead of having all the objects being instantiated by one common object, references to a child object are sent to a parent object when requested.

3.5 Dynamic modeling in Yakindu

One of the requirements of the system is to dynamically create a car handler when the car is close to the terminal and delete it when the car passes.
Yakindu doesn't support this dynamic modeling of the objects, but it offers the following two methods:

- *Enter*: when an instance of an object is created in Yakindu, the object is instantiated but not entered (all of its states are still inactive). The *enter* method is used to enter the initial state of the object.
- *Exit*: when an object arrives to its final state, the *exit* method can be used to completely exit the object; it deactivates all of its states and returns it to where it was in before the *enter* method was called.

A combination of these two methods can be used in order to simulate the dynamic properties of the system (a more in depth explanation is available in 3.8.7).
This solution is not optimal since it simulates the dynamic properties and the car handlers can only be instantiated when the terminal is created whether a car needs to have access to it or not, but it is the only solution available with the current Yakindu semantics.

3.6 Creating multiple instances of an object

Another requirements of the system is to be able to specify its number of cars before starting it. In general, this problem needs to be solved using a list type variable since the number of instances isn't predefined.
Yakindu presents multiple domains for the statecharts to be created in including:

- Default domain: contains only the simple type definitions (integer, real, Boolean, string, void). This domain can be used to generate code in any of the languages supported by Yakindu.
- C/C++ domain: can use types from any imported header on top of the types included in the default domain. This domain can only be used to generate C/C++ code.

Since the goal of this project is to be able to generate the system for any platform, then only the default domain can be used to create the statecharts, which means that lists aren't available to be used to instantiate multiple cars and that we need to have a predefined number of instances of every object before running the system.
This means that the system has to be instantiated with a specific number of cars and that new instances of these objects can't be added while running the system.
The addition of these lists can be done using the code generated by the statecharts, but this defeats the purpose of this project, which is to be able to simulate the whole system without needing additional implementations on top of the statecharts themselves.

3.7 Event routing between the different parts of the system

As mentioned in 3.4, different statechart objects can only communicate directly with their parent by raising out events, or with the children by raising events inside of them, but Yakindu doesn't present a built-in way of communicating between the different parts of the system that aren't directly connected to each other.
Event routing basically consists of raising an event from the child to an object in the system by raising the *out event* to the parent which then reroutes this event to the correct target.
The event routing problem can be solved in Yakindu by giving the destination object as an argument to the event being raised. For example, let's suppose that car1 in our system needs to raise an event called *honk* to car2 but only the control center has a reference to both of these cars. To solve this issue, we can raise the *honk* event in $car1$ as an *out event* that can be read by the control center, and give it the id of the target car as an argument which can be identified by the control center in order to route the event to the correct object.
This solution can be implemented and simulated directly in Yakindu, but it presents the following two problems:

- Events in Yakindu can only hold one argument, so if the argument was the target of the routing then no other argument can be given which can cause problems depending on the system at hand.
- Simple event routing can be simulated directly in Yakindu, but complex ones cannot. For example, if car1 was to send an event to the cruiser of car2, then the argument accompanying this event can only to be a string containing the id of the destination car and a reference to the cruiser. This argument can only be in the form of a single string in Yakindu and will have to be processed in order for the right object to receive the event, but this processing of the string can't be done without an operation being called and in Yakindu, operations can only be defined in the generated code.

3.8 Development of the system

3.8.1 Car

- The car statechart starts in its idle state where it sets both its car handler and terminal variables to null.
- When the control center raises the *destSelected* event, the car sets its destination from the argument added to the event and starts its journey.
- The cars are originally parked in the terminals so they need to access the terminal's car handler in order to leave their platform. Since the car isn't initialized with references to any other object in the system except its own parts, it raises the *getTerminal out event* that is read by its parent the control center which sends it the instance of the terminal it's currently parked in. This is an example of event routing since the car gives its id as argument to the event it's raising and the control center reads this argument in order to know which car should receive the requested terminal.
- The car gets access to the car handler by raising an event to the terminal which returns the instance of the car handler.
- After having received its car handler, the car goes into its departure state where it waits for the car handler to complete its cycle and allocate the exit for the car. At the same time, the car starts its cruiser and readies it for departure.
- During its voyage, the car is in its cruising state where it keeps updating its location every second until it receives an $alert80$ event, which means that there's another car close to it and it needs to stop its cruiser, or and $alert100$ which means that the car is close to a terminal and it needs to access its car handler to allocate the right entrance, platform, and exit in the terminal.
- After having received an instance of the terminal and the car handler, the car goes into its *waitArrival* state which checks if this terminal is its final destination and asks the car handler to allocate the right path to either park in the terminal or pass through it.
- The *waitArrival* state is composed of an orthogonal state nested inside of another orthogonal states. This was done because three different tasks need to be accomplished at the same time: updating the location of the car when it's still moving towards the terminal, checking if it receives an $alert80$ to stop the cruiser until there's enough space between it and the car in front of it, and ask the car handler to allocate the correct path to continue its journey.
- After the *waitArrival* state, the car either goes back to cruising if the terminal wasn't its final destination or goes to its initial *idle* state until it receives another *destSelected* event from the control center to drive a new passenger to their destination.

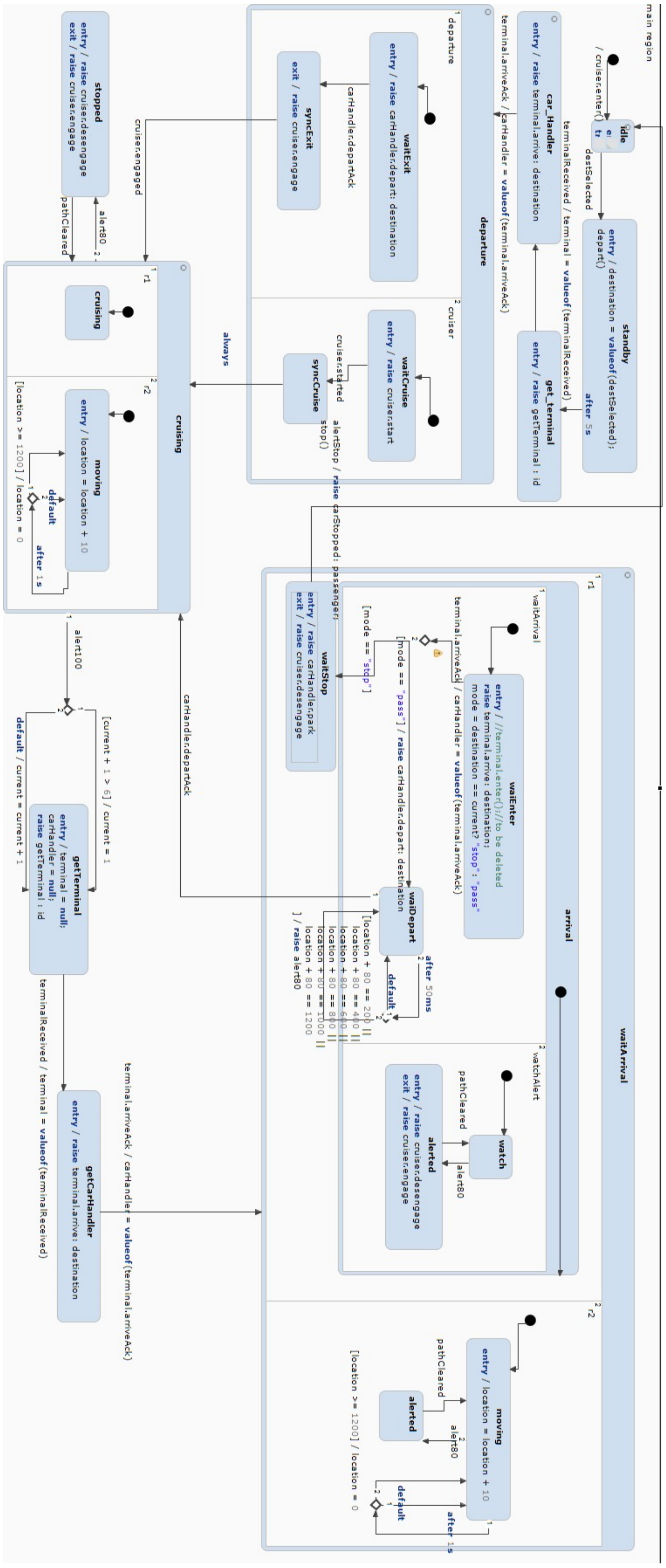
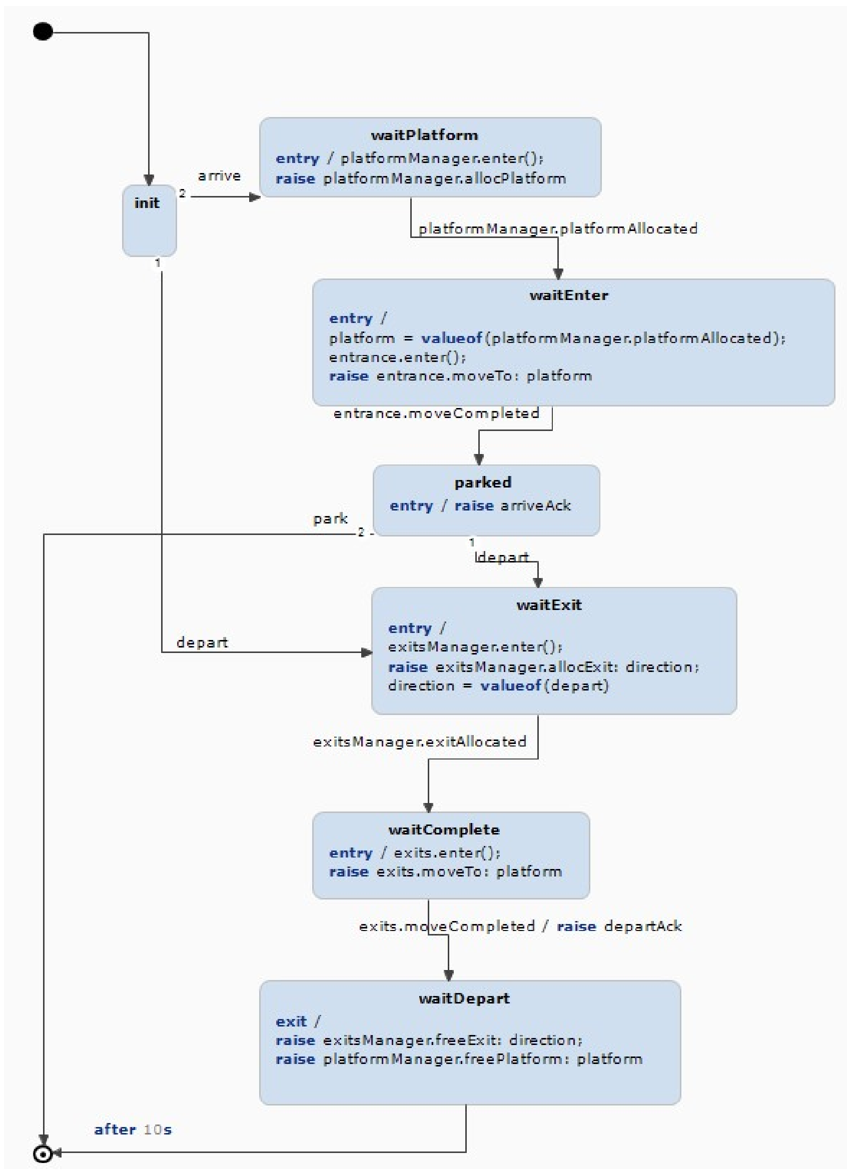

main region
idle
/ cruiser.enter()
destSelected
standby
entry / destination = valueof(destSelected); depart()
after 5s
car_Handler
entry / raise terminal.arrive: destination
terminalReceived / terminal = valueof(terminalReceived)
get_terminal
entry / raise getTerminal : id
departure
terminal.arriveAck / carHandler = valueof(terminal.arriveAck)
departure
waitExit
entry / raise carHandler.depart: destination
carHandler.departAck
syncExit
exit / raise cruiser.engage
cruiser
waitCruise
entry / raise cruiser.start
cruiser.started
syncCruise
alertStop / raise carStopped: passenger, stop()
always
cruiser.engaged
stopped
entry / raise cruiser.desengage
exit / raise cruiser.engage
alert80
pathCleared
cruising
r1
r2
moving
entry / location = location + 10
default
after 1s
[location >= 1200] / location = 0
alert100
[current + 1 > 6] / current = 1
default / current = current + 1
getTerminal
entry / terminal = null; carHandler = null; raise getTerminal : id
terminalReceived / terminal = valueof(terminalReceived)
getCarHandler
entry / raise terminal.arrive: destination
terminal.arriveAck / carHandler = valueof(terminal.arriveAck)
carHandler.departAck
waitArrival
arrival
waitArrival
waitEnter
entry / //terminal.enter();//to be deleted
raise terminal.arrive: destination;
mode = destination == current? "stop" : "pass"
terminal.arriveAck / carHandler = valueof(terminal.arriveAck)
[mode == "pass"] / raise carHandler.depart: destination
[mode == "stop"]
waitStop
entry / raise carHandler.park
exit / raise cruiser.desengage
waiDepart
after 50ms
default
[location + 80 == 200 || location + 80 == 400 || location + 80 == 600 || location + 80 == 800 || location + 80 == 1000 || location + 80 == 1200] / raise alert80
watchAlert
pathCleared
watch
alert80
alerted
entry / raise cruiser.desengage
exit / raise cruiser.engage
moving
entry / location = location + 10
alerted
pathCleared
alert80
default
after 1s
[location >= 1200] / location = 0

Figure 3.2: Statecharts of the car system

Figure 3.3: Statecharts of the car handler system



init
arrive
2
1
waitPlatform
entry / platformManager.enter();
raise platformManager.allocPlatform
platformManager.platformAllocated
waitEnter
entry /
platform = valueof(platformManager.platformAllocated);
entrance.enter();
raise entrance.moveTo: platform
entrance.moveCompleted
parked
entry / raise arriveAck
park
2
1
depart
depart
waitExit
entry /
exitsManager.enter();
raise exitsManager.allocExit: direction;
direction = valueof(depart)
exitsManager.exitAllocated
waitComplete
entry / exits.enter();
raise exits.moveTo: platform
exits.moveCompleted / raise departAck
waitDepart
exit /
raise exitsManager.freeExit: direction;
raise platformManager.freePlatform: platform
after 10s

3.8.2 Car handler

- The car handler starts in its *init* state where it waits until it receives an event from the car.
- After its *init* state, the car handler can either receive a *depart* event, which means that the car was originally parked in this terminal and that the car handler only needs to allocate an exit; or an arrive event, which means that the car is arriving to the terminal and that the car handler needs to allocate an entrance and a platform on top of the exit.
- The out events raised by the car handler like *arriveAck* and *departAck* are received by the car which uses them to transition to different states.

3.8.3 Control center

- When the control center is initiated, it initiates the locations manager, cars, and terminals of the system.
- The cars aren't created directly in the control center, but they are taken from the locations manager, so both have the same instance of each car instead of having duplicates. A different approach would be to instantiate the cars in the control center and then send their instances to the locations manager.
- After having instantiated all the objects, the control center goes to its *tmp* state where it waits until it receives events either from the cars or from the passengers.
- When a car raises a *getTerminal* event, the control center assigns that car to its *carSelected* variable using the id attribute added to the event, and sends it the instance of the terminal that the car requests. The *carSelected* variable is added in order to minimize the number of states needed. So instead of having a different state for every car so that the control center can know which car sent the event and should receive an event back, only one common state is needed which sends the event to the correct car using the event routing technique.
- The control center knows which terminal the car belongs to by accessing the car's current variable which is updated whenever the car gets to a new terminal.
- When a passenger wants to travel to a destination, the passenger statechart raises a *passengerWaiting* event inside of the control center and sends an integer value which represents the current terminal the passenger is waiting in. The control center then selects the closest car to this terminal by calling the *selectCar* function. If the selected car is currently parked at the same terminal as the passenger then the control center goes to its *carWaiting* state, otherwise it goes to its *sendingCar* state.
- Before getting to the *sendingCar* state, the control center sends the right terminal to the car so it can access its *carHandler*.
- In the *carWaiting* state, the control center raises a *carWaiting* out event which is read by the passenger and is used to notify them that the car has arrived to their terminal. The passenger then sends the number that identifies their destination terminal which is, in turn, sent to the car to start the journey.

3.8.4 Cruiser

- The cruiser statechart is composed of 4 states that show the cruiser's current status: *stopped*, *started*, *engaged*, *disengaged*.
- The cruiser's in events are only raised by the car and its out events are received by the same car.

3.8.5 Locations manager

The locations manager's only purpose is to create different location managers (or proximity sensors) for every car and give access between them. Without the locations manager we

Figure 3.4: Statecharts of the control center system

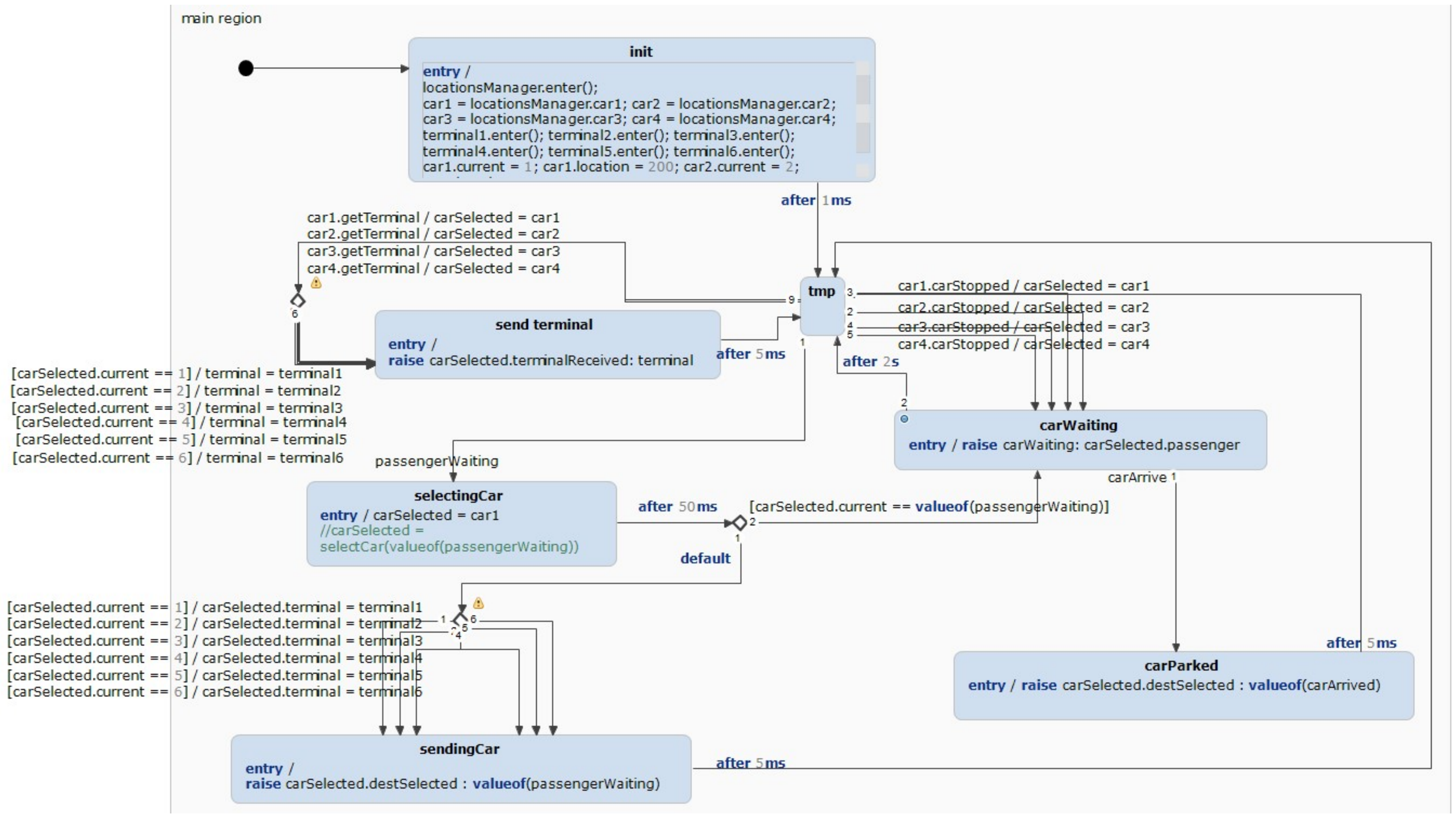

main region
init
entry /
locationsManager.enter();
car1 = locationsManager.car1; car2 = locationsManager.car2;
car3 = locationsManager.car3; car4 = locationsManager.car4;
terminal1.enter(); terminal2.enter(); terminal3.enter();
terminal4.enter(); terminal5.enter(); terminal6.enter();
car1.current = 1; car1.location = 200; car2.current = 2;
after 1ms
car1.getTerminal / carSelected = car1
car2.getTerminal / carSelected = car2
car3.getTerminal / carSelected = car3
car4.getTerminal / carSelected = car4
tmp
car1.carStopped / carSelected = car1
car2.carStopped / carSelected = car2
car3.carStopped / carSelected = car3
car4.carStopped / carSelected = car4
send terminal
entry /
raise carSelected.terminalReceived: terminal
after 5ms
after 2s
[carSelected.current == 1] / terminal = terminal1
[carSelected.current == 2] / terminal = terminal2
[carSelected.current == 3] / terminal = terminal3
[carSelected.current == 4] / terminal = terminal4
[carSelected.current == 5] / terminal = terminal5
[carSelected.current == 6] / terminal = terminal6
carWaiting
entry / raise carWaiting: carSelected.passenger
passengerWaiting
carArrive
selectingCar
entry / carSelected = car1
//carSelected =
selectCar(valueof(passengerWaiting))
after 50ms
[carSelected.current == valueof(passengerWaiting)]
default
[carSelected.current == 1] / carSelected.terminal = terminal1
[carSelected.current == 2] / carSelected.terminal = terminal2
[carSelected.current == 3] / carSelected.terminal = terminal3
[carSelected.current == 4] / carSelected.terminal = terminal4
[carSelected.current == 5] / carSelected.terminal = terminal5
[carSelected.current == 6] / carSelected.terminal = terminal6
carParked
entry / raise carSelected.destSelected : valueof(carArrived)
after 5ms
sendingCar
entry /
raise carSelected.destSelected : valueof(passengerWaiting)
after 5ms

Figure 3.5: Statecharts of the cruiser system

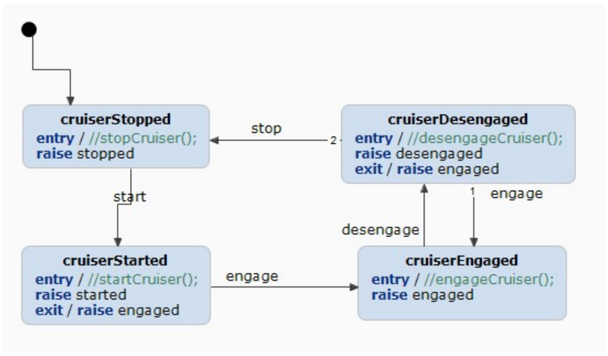
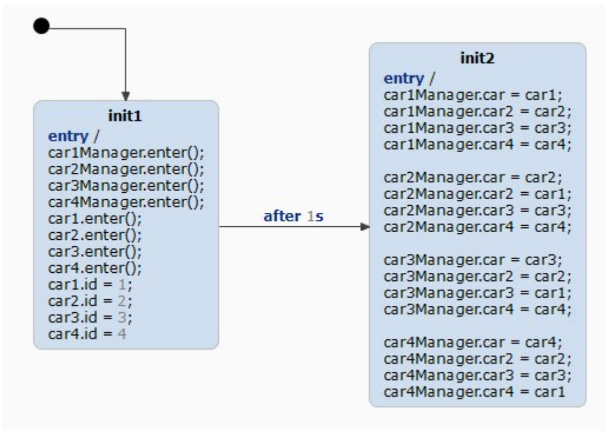


Figure 3.6: Statecharts of the locations manager system



would have a single statechart with as many orthogonal states as the number of cars we have defined in the system.

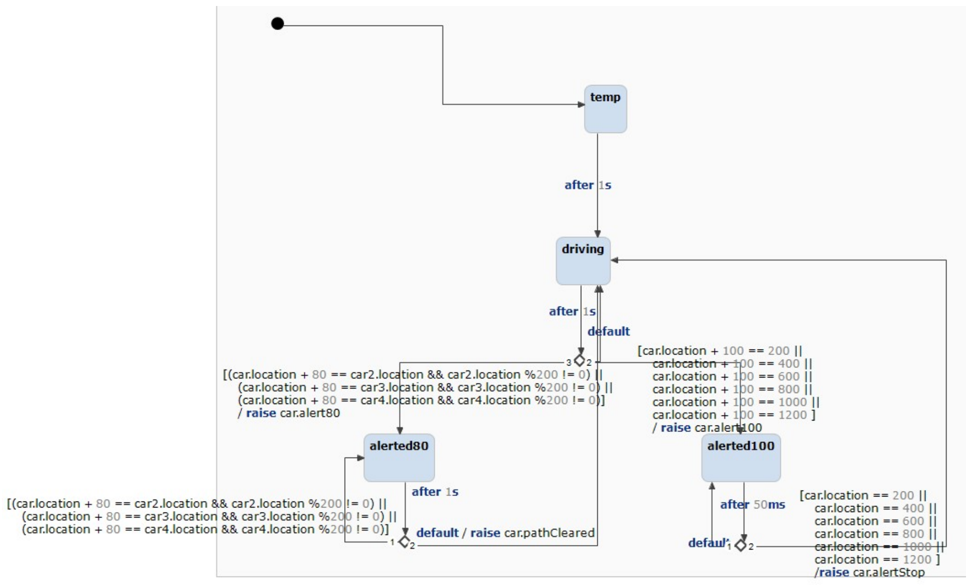
3.8.6 Location manager

- Every car has its own location manager which serves to check if it's close to a terminal or to another car. This statechart is implemented because the system is only running in a simulation where there are no proximity censors and physical distances available. In a real world application of this system, this statechart would be simplified to only raising the events to the car when needed without having to check the different cars' locations.
- Every second, the location manager checks if the car is within 80 yards from another car to send an $alert80$ event, or if it's within 100 yards from a terminal to send an $alert100$ event. Those events are raised by the location manager directly to the car.

3.8.7 Terminal

- The terminal statechart is composed of two orthogonal states: one that is used to send the instance of the car Handler when requested, and another one that takes care of exiting the car handler to simulate their "dynamic creation".
- When the terminal receives an arrive event from the car, it selects a car handler that isn't currently in use by checking its *isActive* property, enters its starting state

Figure 3.7: Statecharts of the location manager system


temp
after 1s
driving
after 1s
default
[(car.location + 80 == car2.location && car2.location %200 != 0) ||
(car.location + 80 == car3.location && car3.location %200 != 0) ||
(car.location + 80 == car4.location && car4.location %200 != 0)]
/ raise car.alert80
[car.location + 100 == 200 ||
car.location + 100 == 400 ||
car.location + 100 == 600 ||
car.location + 100 == 800 ||
car.location + 100 == 1000 ||
car.location + 100 == 1200]
/ raise car.alert100
alerted80
alerted100
after 1s
[(car.location + 80 == car2.location && car2.location %200 != 0) ||
(car.location + 80 == car3.location && car3.location %200 != 0) ||
(car.location + 80 == car4.location && car4.location %200 != 0)]
default / raise car.pathCleared
after 50ms
default
[car.location == 200 ||
car.location == 400 ||
car.location == 600 ||
car.location == 800 ||
car.location == 1000 ||
car.location == 1200]
/raise car.alertStop

to activate it, and sends it to the car that requested it.

- Every second, the terminal checks if any of its car handlers are in their final state by checking their *isFinal* property, if they are in their final state then the terminal exits them in order to be entered again once requested by another car.

3.8.8 Passenger

- The passenger statechart has three states it can be in: *main* which means that the passenger is idle, *wait* which means that the passenger has selected their destination and is waiting for the car to arrive, and *moving* which means that the passenger is in the car and heading towards their destination.
- The passenger statechart only communicates with the control center, which in turn processes the events, changes states, and raises events to the car.

3.8.9 Simulation

The simulation statechart is only used to simulate the system, it waits for the user to select an origin and destination terminal to create a new passenger with the given information, and raises a *destinationSelected* event in the created passenger statechart which starts the cycle with the rest of the system until the passenger gets to their destination.
In the real world application of this system, this statechart doesn't have to be implemented since real passengers will be interacting with the system.

3.9 Statecharts simulation

Yakindu has the option to directly simulate and debug the system without the need to generate code from the created statecharts. Using this statechart simulation we can easily see which events are firing and their effect on the transitions between the states (as shown in Figure 3.11). When simulating the system directly in Yakindu, instances of the

Figure 3.8: Statecharts of the terminal system

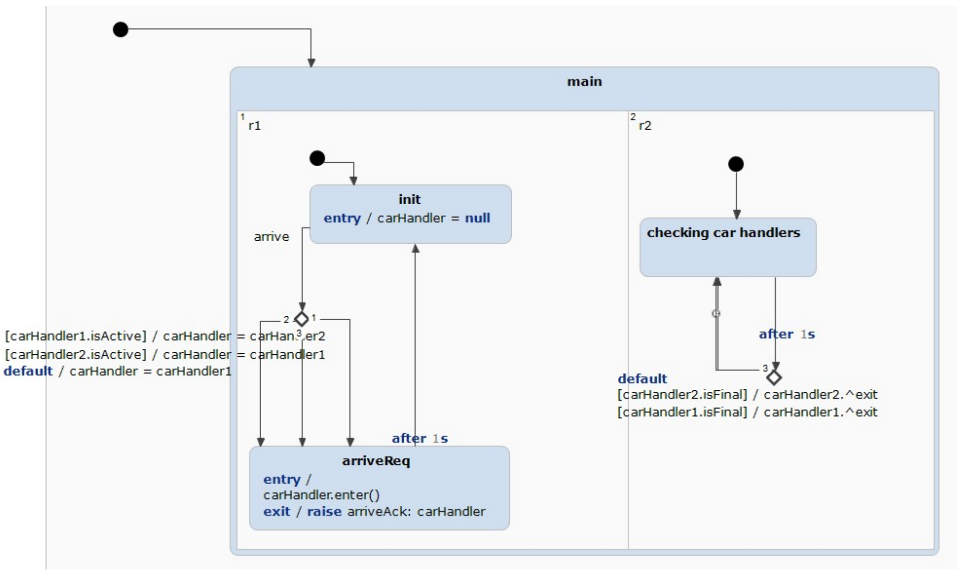

main
r1
r2
init
entry / carHandler = null
arrive
[carHandler1.isActive] / carHandler = carHandler2
[carHandler2.isActive] / carHandler = carHandler1
default / carHandler = carHandler1
after 1s
arriveReq
entry / carHandler.enter()
exit / raise arriveAck: carHandler
checking car handlers
after 1s
default
[carHandler2.isFinal] / carHandler2.^exit
[carHandler1.isFinal] / carHandler1.^exit

Figure 3.9: Statecharts of the passenger system

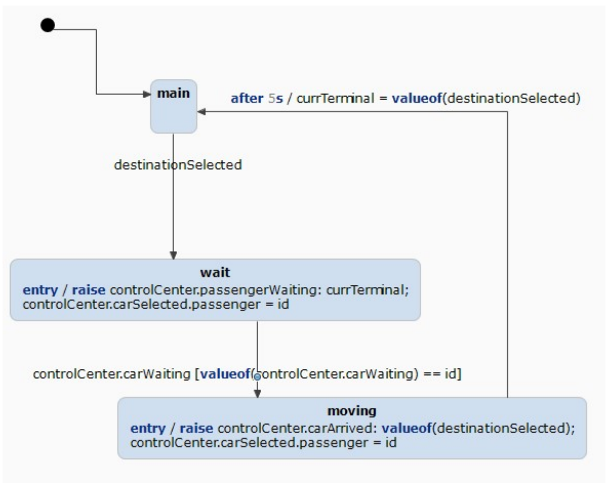

main
after 5s / currTerminal = valueof(destinationSelected)
destinationSelected
wait
entry / raise controlCenter.passengerWaiting: currTerminal; controlCenter.carSelected.passenger = id
controlCenter.carWaiting [valueof(controlCenter.carWaiting) == id]
moving
entry / raise controlCenter.carArrived: valueof(destinationSelected); controlCenter.carSelected.passenger = id

Figure 3.10: Statecharts of the simulation system

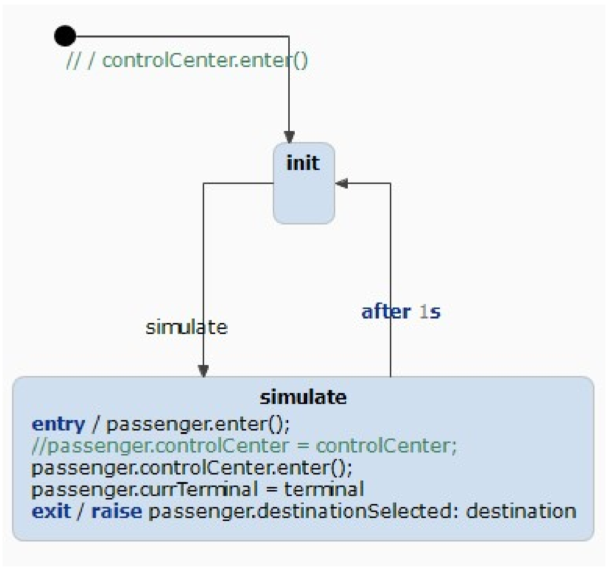
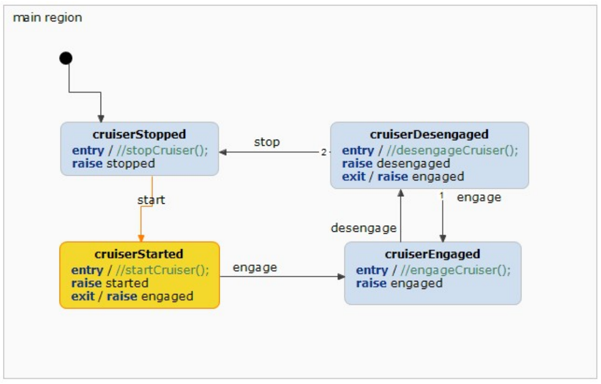

// / controlCenter.enter()
init
simulate
after 1s
simulate
entry / passenger.enter();
//passenger.controlCenter = controlCenter;
passenger.controlCenter.enter();
passenger.currTerminal = terminal
exit / raise passenger.destinationSelected: destination

Figure 3.11: Simulation interface in Yakindu

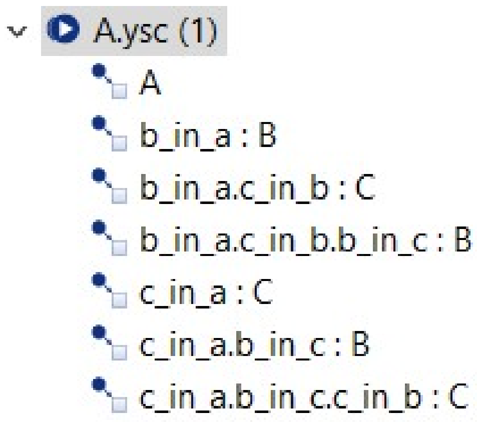


statecharts are created wherever a variable that has as value that statechart is inserted. For example, in our system, the car object has a terminal variable of type terminal. This variable is never instantiated directly in the car object, but it is used to save the instance of the terminal sent by the control center when requested. Yakindu instantiates a terminal object when it starts the simulation and saves it in the terminal variable of the car object even though that instantiated object is never accessed and is immediately replaced by the terminal sent by the control center. This causes the simulation to take a long time to start especially when we have statecharts nested in other statecharts.
To test the creation of those nested statecharts and whether they can result in an infinite loop of object creation, created a new test project is created which has three simple statecharts A, B, and C:

- Statechart A is considered as the parent, it has two variables: b_in_a of type B and c_in_a of type C.
- Statechart B has one variable c_in_b of type C.
- Statechart C has one variable b_in_c of type B.

If the purpose of this system is similar to the railcar system where statechart A, the parent, sends the objects of type B and C to statecharts C and B respectively, then we will only need a total of 3 objects created. But running the simulation of statechart A results in 7 objects being created as shown in Figure 3.12, therefore the simulation doesn't go in an infinite loop of object creations, but it does indeed create more instances of each object than needed. This disregards the problem of an infinite loop, but it proves that bigger systems will have more objects created than they are needed.
Yakindu presents a way to enter a statechart's initial state using the enter function, but it creates the objects on its own whenever the simulation starts running.
Additionally, Yakindu doesn't offer a way to define the operations before generating code, so the *selectCar* operation, which is used to select the closest car to the passenger in order to start the journey, cannot be implemented. For this reason, the operation was changed in the simulation only to directly select car1 no matter where the passenger is. This means that the results from running the simulation will not be accurate since only one car can be used, but it allows us to use the built-in simulator without any additional changes.

Figure 3.12: Instances created by the system



A.ysc (1)
A
b_in_a : B
b_in_a.c_in_b : C
b_in_a.c_in_b.b_in_c : B
c_in_a : C
c_in_a.b_in_c : B
c_in_a.b_in_c.c_in_b : C

3.10 Code generation

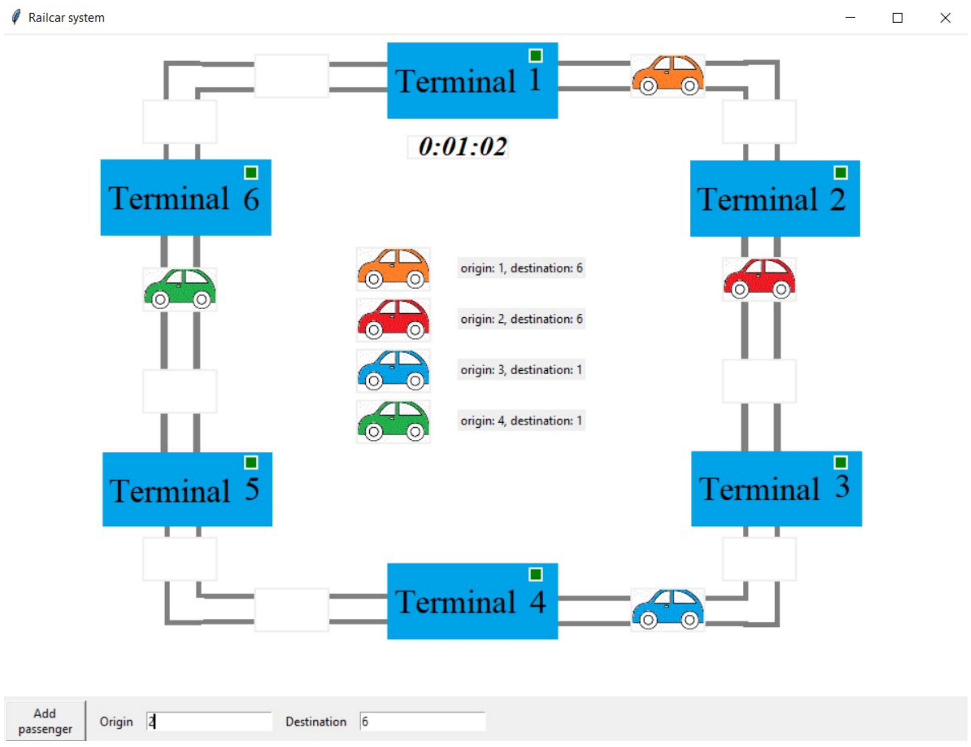
Yakindu has built in code generators that allow to generate code for multiple programming languages out of the statecharts created. The programming language used for the graphical user interface implementation of this system is Python with the TKinter library, therefore the statecharts were used to generate Python code to run in the graphical application.
Generating code out of the Yakindu statecharts doesn't have the same issue as the simulation where instances of the objects are created even when they are not needed since the different statechart objects have to be instantiated manually and added to the variables they belong to. So, for the test system mentioned in 3.9, we will only need to instantiate three objects instead of the seven objects instantiated by the Yakindu simulation.
On top of the manual instantiation of the objects, the operation added to the statecharts (mainly $selectCar$) and the functions linking the user interface to the generated code needs to be implemented.
Having to do changes to the system after generating the code means that the statecharts created in Yakindu cannot be directly used to deploy a created system on the target components, but a code change is needed whenever a new programming language is chosen as target for the code generation or whenever the system's functionality is changed.

3.11 Graphical user interface

The graphical user interface (GUI) is composed of the following:

- A canvas that is the center of the application which contains all the terminals along with the tracks between them and the cars.
- Multiple canvases on the tracks where there's two between every couple of terminals. These canvases are used to contain the car image to be able to visualize its movement. There's only two canvases between every two terminals because there can only be a maximum of two cars on a single track: the terminals are placed 200 yards away from one another and the car's proximity censor (or in the simulation, their location manager) stops it from moving if it is 80 meters away from another car.

Figure 3.13: A running example of the GUI



- A canvas on top of each terminal that signals whether there's a car currently parked in that terminal. The canvas turns red if there's a car parked and green otherwise.
- A timer in the middle of the application screen that shows the passage of time.
- A button in the bottom of the application along with two input fields that allow the user to add the current and destination terminals of a new passenger and add them to the simulation.
- A list in the middle of the screen that shows the cars that are currently traveling to a destination terminal along with their origin and target terminal to make it easier to follow along their journey.

To run the GUI in a more efficient way, a new statechart object called *Timer* is added to the system, it calls a *second_passed* function every second. The *second_passed* function is used to check the location of the different cars along their journey in order to update their location on the map accordingly. This function is also used to update the timer placed in the graphical user interface and the list shown underneath it.

3.12 Multi-state machine modeling in SCCD

SCCD is mainly built around multi-state machine modeling since it incorporates the state diagram in its language.
Relationships between the different objects in the system can be defined when an object is created, SCCD's object manager is then called to instantiate, start, or delete the object defined. Creating a relationship between a parent and a child object not only creates a reference to the child in the parent, but it also creates a reference to the parent in the child which allows a bidirectional connection between them. This means that, unlike in

Yakindu, out events are useless since the event can be directly raised to the parent.
After having defined the different relationships in the system, an object can directly raise an event in a target object by changing the scope of the event being raised to *narrow* and giving it the name of the object being targeted:

$$< raise\ scope = "narrow"\ target = "'passenger'"\ event = "destinationSelected" / >$$

Similarly to Yakindu, statechart can only communicate with the parent/child object, if an event needs to be raised between two objects that aren't directly related to each other then a workaround needs to be implemented. But unlike in Yakindu, SCCD allows the object to send a reference of itself to a target object, which means that the required relationships between the different elements of the system can be easily defined in the system using the approach of having an object raise an event with the instance of the requested object as its attribute. In order to be able to use this functionality, we have to use the underlying methods of the generated code in SCCD, which, in python, can be done using:

$$associations['association\_name'].getInstance(0)$$
and
$$associations['association\_name'].addInstance(cs)$$

3.13 Dynamic modeling in SCCD

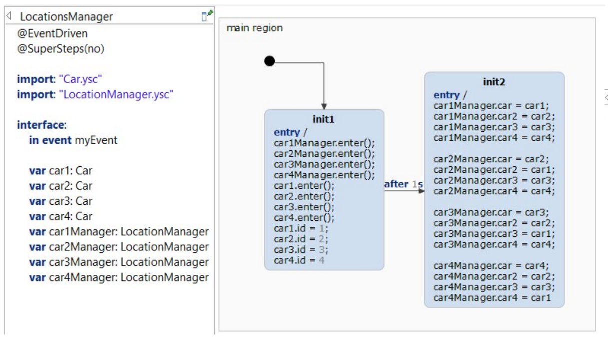
Unlike Yakindu, SCCD has a simple way of dealing with dynamic modeling using its object manager.
The object manager can be used to create new instances of an object, start the state machine behind an object and enter its initial state, and delete an already created object. Because of this, the *carHandler*, which needs to be dynamically created and deleted, can function as intended without the need to simulate the dynamic properties by entering and exiting the state machine.

3.14 Creating multiple instances of an object

Since SCCD incorporates the state diagram into its language, we can easily create multiple instances of an object by defining the relationship's minimal and maximal cardinality. This means that the number of cars in the system don't have to be predefined but they can be specified before running the system which is one of the requirements of the railcar system as defined in 3.1.

3.15 Development of the system

In order to make the development of the system easier and more conform to the one explained in 3.8, Yakindu's code generator was used to generate the SCXML code behind the system.
Yakindu can only generate SCXML code which, unlike SCCD, doesn't define classes and the relationships between them. Therefore the system that is used to generate the code cannot include multi state machine modeling and needs to be created using the SCXML domain which is the only domain that allows SCXML code generation.



```
LocationsManager
@EventDriven
@SuperSteps(no)

import: "Car.ysc"
import: "LocationManager.ysc"

interface:
  in event myEvent

  var car1: Car
  var car2: Car
  var car3: Car
  var car4: Car
  var car1Manager: LocationManager
  var car2Manager: LocationManager
  var car3Manager: LocationManager
  var car4Manager: LocationManager
```

```
init1
entry /
car1Manager.enter();
car2Manager.enter();
car3Manager.enter();
car4Manager.enter();
car1.enter();
car2.enter();
car3.enter();
car4.enter();
car1.id = 1;
car2.id = 2;
car3.id = 3;
car4.id = 4
```

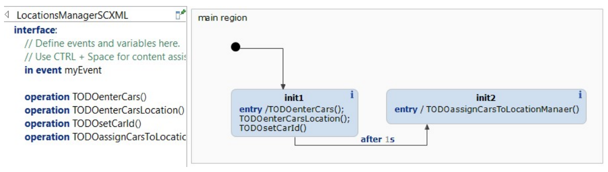
```
init2
entry /
car1Manager.car = car1;
car1Manager.car2 = car2;
car1Manager.car3 = car3;
car1Manager.car4 = car4;

car2Manager.car = car2;
car2Manager.car2 = car1;
car2Manager.car3 = car3;
car2Manager.car4 = car4;

car3Manager.car = car3;
car3Manager.car2 = car2;
car3Manager.car3 = car1;
car3Manager.car4 = car4;

car4Manager.car = car4;
car4Manager.car2 = car2;
car4Manager.car3 = car3;
car4Manager.car4 = car1
```

(a) Yakindu statechart



```
LocationsManagerSCXML
interface:
  // Define events and variables here.
  // Use CTRL + Space for content assis
  in event myEvent

  operation TODOenterCars()
  operation TODOenterCarsLocation()
  operation TODOsetCarId()
  operation TODOassignCarsToLocatic
```

```
init1
entry /TODOenterCars();
TODOenterCarsLocation();
TODOsetCarId()
```

```
init2
entry / TODOassignCarsToLocationManaer()
```

(b) SCXML statechart

Figure 3.14: Comparison between the Yakindu and SCXML statecharts of the Locations Manager object

The new SCXML system developed in Yakindu is a copy of the one explained in 3.8 but without any of the semantics used for multi state modeling, these semantics are replaced with normal events or operations to make the manual editing of the generated SCXML code simpler (as shown in Figure 3.14). After defining the new SCXML system and generating its code, some changes need to be done to the code in order to comply with the SCCD semantics and to bring multi state modeling back.
The only differences between the systems created in Yakindu and SCCD is that, in SCCD, the $LocationManager$ and $LocationsManager$ objects have been combined into one. This change is made mainly because the semantics of SCCD allow for a combined object that can still have functionalities of both of the old objects without making the system too complex and hard to read. This change doesn't affect the underlying system and its functionality though.

3.16 Statecharts simulation

SCCD is a textual language and code generator tool, therefore it doesn't have any built-in way of visually simulating a system without generating its code and connecting it to a GUI.

3.17 Code generation

Similarly to Yakindu, SCCD has its own set of code generators that are used to create code for the target language out of the system created. Since the GUI is written in *Python*, the same language is used as target for the generated code.
The advantage of SCCD over Yakindu in this part is that the code generated doesn't need to be edited in order to add the functionality of the operations defined in the state machine system and to instantiate the objects used like in Yakindu. Instead, all the definitions can be directly added to the statecharts and the only addition needed to run the system is a runner file that instantiates it.
SCCD presents three different runtime platforms that can be used in the runner file, these runtime platforms define the functions used by the runtime kernel which translates into the way it handles events during execution.
The three runtime platforms are:

- *Threads*, which manipulates a global event queue made thread safe by locks. However, running an application on this platform can interfere with other scheduling mechanisms like the UI.
- *UI Event Loop*, which fixes the problem defined in the *threads* platform by reusing the event queue managed by an existing UI platform, such as Tkinter.
- *Game Loop*, which facilitates integration with game engines where objects are updated only at predefined points in time.

Since the final version of this system is based on a GUI implementation, the *UI event loop* is used when running the system.

3.18 Graphical user interface

The GUI used for the SCCD system is the same as the one used for the Yakindu system, they are a simple copy of each other with just the connection to the statecharts that control the system being changed.

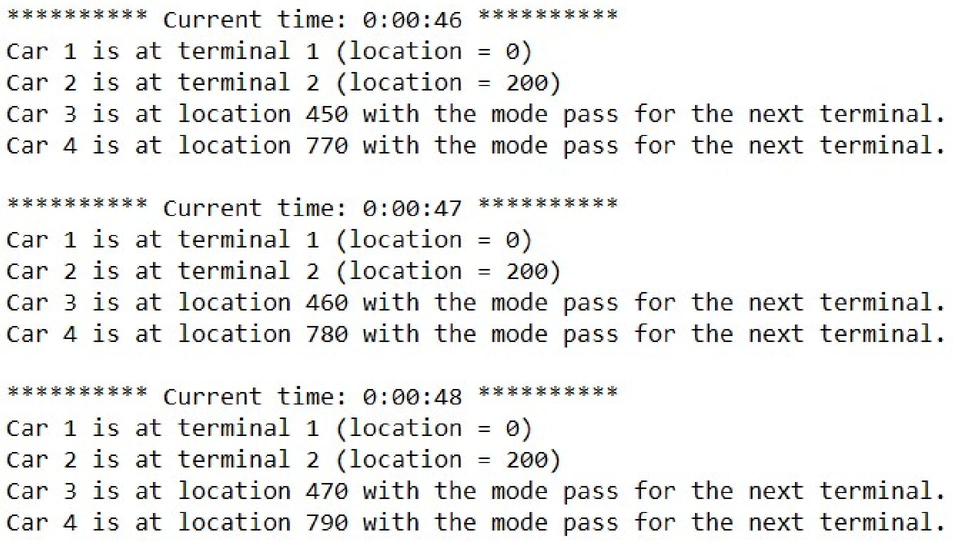
3.19 Comparison of the two tools

3.19.1 System trace

In order to compare the systems created by both tools, a trace of both of them is generated when they are run.
The trace is a simple text file that describes the passage of time and the movements of the cars between the terminals. (An example of the content of the trace file is shown in Figure 3.15)
When comparing the traces between the two system, we notice a difference in the movements of the car depending on the time even when the passengers have been added to the system at the exact same time. This difference is mainly attributed to the way both tools handle the creation of the different objects and not to how the events propagate through the system:

- In Yakindu, all the objects needed in the system are manually instantiated before starting the program using the code generated; but in SCCD, the objects are created directly by the statecharts when the object gets to the state that calls for their creation. For example, the *carHandler* objects need to be instantiated whenever the car gets close to a terminal using SCCD, but with Yakindu the *carHandler* objects are predefined and already instantiated whether a car is requesting them or not.
- Instantiation of objects in SCCD requires more states and transitions than in Yakindu which can cause a delay in the system: the object manager is first called in state A to instantiate the object, the object manager raises the event *instance_created* which transitions the object to state B, the object manager is called in state B to start the object, the object manager raises the event *instance_started* which transitions the system to state C, finally in state C the object can start its actual cycle. Meanwhile in Yakindu, the *object_name.enter* can be called in any state to enter an object and no transitions are needed afterwards.

Figure 3.15: Trace generated by the system



```
********** Current time: 0:00:46 **********
Car 1 is at terminal 1 (location = 0)
Car 2 is at terminal 2 (location = 200)
Car 3 is at location 450 with the mode pass for the next terminal.
Car 4 is at location 770 with the mode pass for the next terminal.

********** Current time: 0:00:47 **********
Car 1 is at terminal 1 (location = 0)
Car 2 is at terminal 2 (location = 200)
Car 3 is at location 460 with the mode pass for the next terminal.
Car 4 is at location 780 with the mode pass for the next terminal.

********** Current time: 0:00:48 **********
Car 1 is at terminal 1 (location = 0)
Car 2 is at terminal 2 (location = 200)
Car 3 is at location 470 with the mode pass for the next terminal.
Car 4 is at location 790 with the mode pass for the next terminal.
```

3.19.2 Railcar system requirements satisfaction

3.19.2.1 Yakindu

In Yakindu, some of the requirements couldn't be met due to how the tool works:

- Dynamic creation and deletion of the *carHandler* objects is not possible to be done, the effect of such dynamic properties have been modeled in the system by using the *enter* and *exit* methods but this does not give the system real dynamic properties.
- Not being able to create the cars on runtime depending on the number specified before starting the system and instead having to use a predefined number of instances of each object.

On top of that, a lot of workarounds needed to be found and implemented in order to have the system function as intended:

- Sending events to the target object and the problem of event propagation and the object not being able to send a reference of itself.
- Not being able to run the simulation without having to edit the system and lose some of its functionalities because of the operations that can only be defined in the generated code.

3.19.2.2 SCCD

In SCCD, the only requirement that couldn't be met is the direct simulation of the system. And the only workaround that needed to be found is how to raise events between objects in the system that aren't directly related to each other by creation, but this issue was easily resolved using the underlying code of the system and using the fact that SCCD doesn't force us to only add one attribute to an event but it allows us to add more if needed.

4 SCCD editor and simulator

After having explained the implementation of a complex system, like the Railcar system, using both tools, and compared the work needed and the end result; it becomes apparent that SCCD is built to better handle the different and more complicated requirements, but it still lacks behind with a harder approach to the development caused by the use of SCCXML and lack of a visual user interface. SCCD also doesn't have any simulation environment to visualize the system and interact with it without having to deploy it first, which means that this functionality needs to be added in order to comply with all the requirements of a complex system as given in the railcar example. Therefore, this section explains how this tool can be expanded to include a visual user interface that allows to easily edit and create new systems, and a simulator to test out the system before deployment.

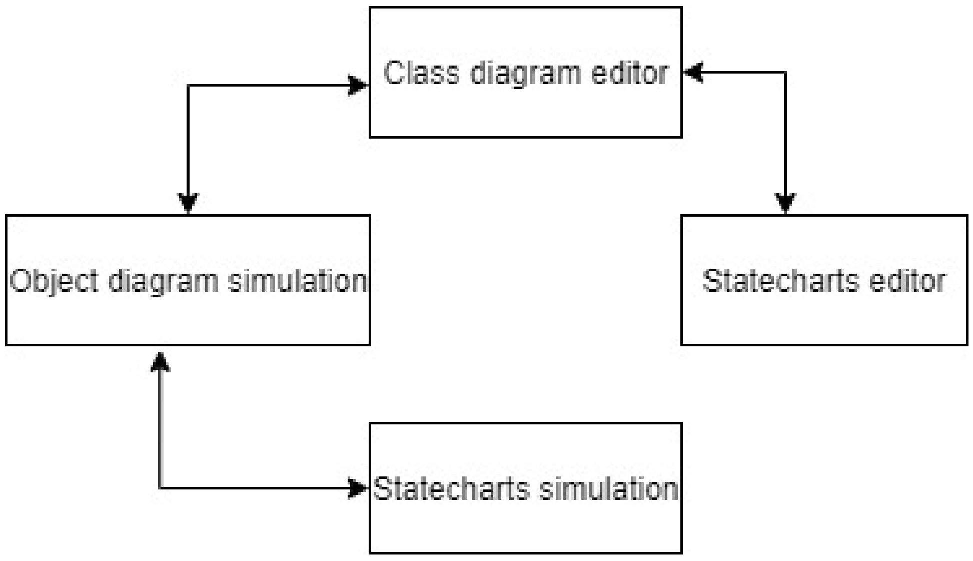
4.1 Requirements of the SCCD editor and simulator

The new features that should be added to the SCCD tool have the following requirements:

- A visual class editor that allows the creation of the different classes of the system. The class editor should support all the functionalities of SCCD as explained in 2.3.
- A visual statecharts editor that allows the creation of the statecharts for each class added to the system. The statecharts editor should also support all the functionalities of SCCD.
- The possibility to save the system being created and to load one in order to edit it.
- An exporter that generates an SCCXML file of the system created so that the developer can deploy it when the development is done.
- A simulator of the developed system that contains:
  - An object diagram simulator that shows the instances of the objects being created during the simulation.
  - A statecharts diagram simulator that shows how the system switches between its different states during the simulation. It should also display the variables of that object and how their values change when the system is running. The statecharts diagram has to also allow the developer to interact with the system by raising events and watching their effects on the active states and the data of the object instances.

The SCCD tool is used to develop the SCCD editor and simulator, therefore the system will be separated into multiple classes and statecharts related to each one of them following the SCCD formalism. The programming language used is python with the Tkinter library to design the UI

Figure 4.1: SCCD Tool system overview



4.2 Motivation for developing the system using SCCD

4.2.1 Why use statecharts

Most graphical user interfaces are event driven, meaning that the components of the visual system themselves generate events when the user interacts with them. These events typically trigger actions attached directly to the UI components. Statecharts inject themselves between the event being generated and the action being performed and they take the event created somewhere to make a decision on the actions that should happen [18]. Additionally, statecharts are perfect for developing systems where such events need to be passed around the different hierarchical elements with its ability to react to events and raise new ones accordingly, which is an important aspect in the system to be developed.

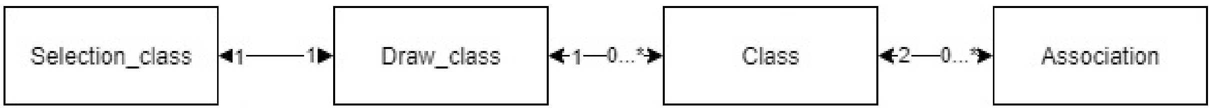
4.2.2 Why use SCCD

The system to be developed is complex and contains multiple components that react differently to the events being raised.
Multiple statecharts formalisms are available and can be used in the development, but SCCD's approach in separating the system into classes where each class has its own statecharts is perfect for the goal of the system. This allows us to have a separate class for each component, and lets them run independently, yet still interact with each other when needed.

4.3 Development of the system

This section will start by showing the UML class diagram of the system to explain its different classes and show how they are connected to each other. It also explain how the user events (like mouse clicks...) are sent to the system. Finally, it explains the statecharts implementation of these classes to also show how each one of these objects functions separately.

Figure 4.2: Class diagram editor UML class diagram



4.3.1 Class diagram

The system can be separated into four main parts connected together as shown in fig. 4.1, these four parts are the following:

- Class diagram editor
- Statecharts editor
- Object diagram solution
- Statecharts simulation

These four parts form together the complete SCCD editor and simulator tool.
In order to simplify the diagrams and their explanation, each one of these parts is explained alone, then their connection together is shown. The rest of the explanation in the following subsections will also be divided into these four parts for simplicity.

4.3.1.1 Class diagram editor

The class diagram editor part contains all the classes necessary to create a UML class diagram as shown in Fig. 4.2:

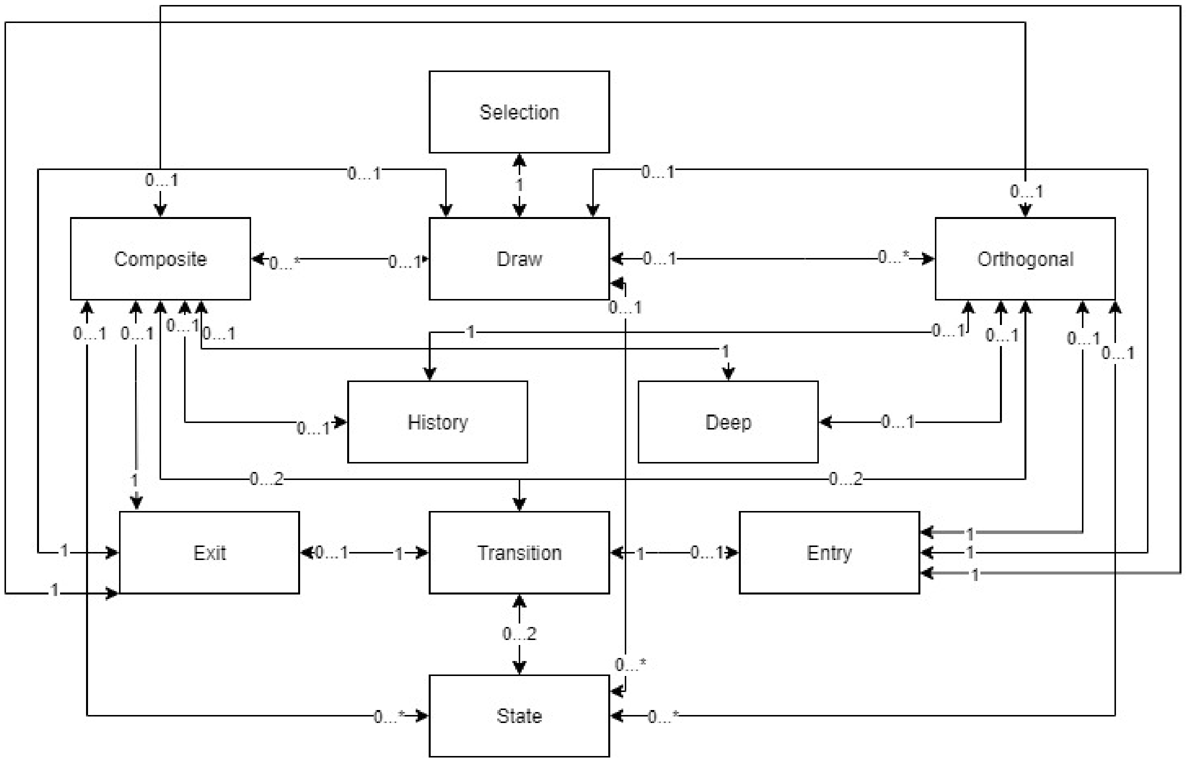
- *Draw_class* is the parent of all the other elements of the class diagram editor. It represents the UI canvas that will contain the drawing of the elements added and allows the user to interact with them.
- *Selection_class* represents the toolbar used by the user to choose new elements to add them to the diagram (here only classes and associations). It has a one to one association with *draw_class*.
- *Class* represents the different classes added by the developer to the system they're creating. A *class* has to be connected to a *draw_class* but the *draw_class* can have zero to many *classes*.
- *Association* represents the associations between the different classes added by the developer. The *association* is connected to two *classes* while the *classes* have a zero to many connection with *association*.

4.3.1.2 Statecharts editor

The statecharts editor contains all the classes necessary to create a statecharts for the created classes of the system as shown in Fig. 4.3:

- *Draw* is the parent of all the other elements of the statecharts editor. It represents the UI canvas that will contain the drawing of the elements added and allows the user to interact with them.

Figure 4.3: Statecharts editor UML class diagram



- *Selection* represents the toolbar used by the user to choose new elements to add them to the diagram (states, composite states, orthogonal states, transitions, histories, entries and exits). Every *draw* class must have one *Selection* class associated to it and every *selection* class must be associated to one *draw* class.
- *Composite* represents the composite states of the statecharts
- *Orthogonal* represents the orthogonal regions of the statecharts
- *History* and *Deep* both represent the two kinds of histories available in statecharts: shallow history and deep history respectively. Both histories only contain an association with the *orthogonal* and *composite* classes that can contain at most one history, and every history has to be associated to either one *orthogonal* or one *composite* class.
- *Entry* and *Exit* represent the initial and final state in the system being created. the *draw* class must have at most one *entry*/*exit* and the same goes for the *composite* and *orthogonal* classes. And each *exit* and *entry* has to be associated to one of these classes.
- *State* represents the states containing the main logic behind the system. *Draw*, *orthogonal* and *composite* classes can all have no states or an infinite number of them while every state has be associated to one of these classes, but *state* has to be connected to at least one of them.
- *Transition* represents the transitions between the different elements of the statecharts. The *transition* classes can be associated to a maximum of two *state*,

Figure 4.4: Object diagram simulation UML class diagram

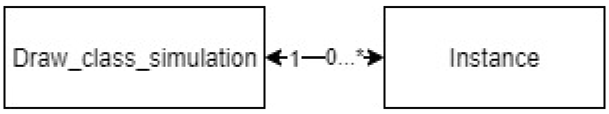
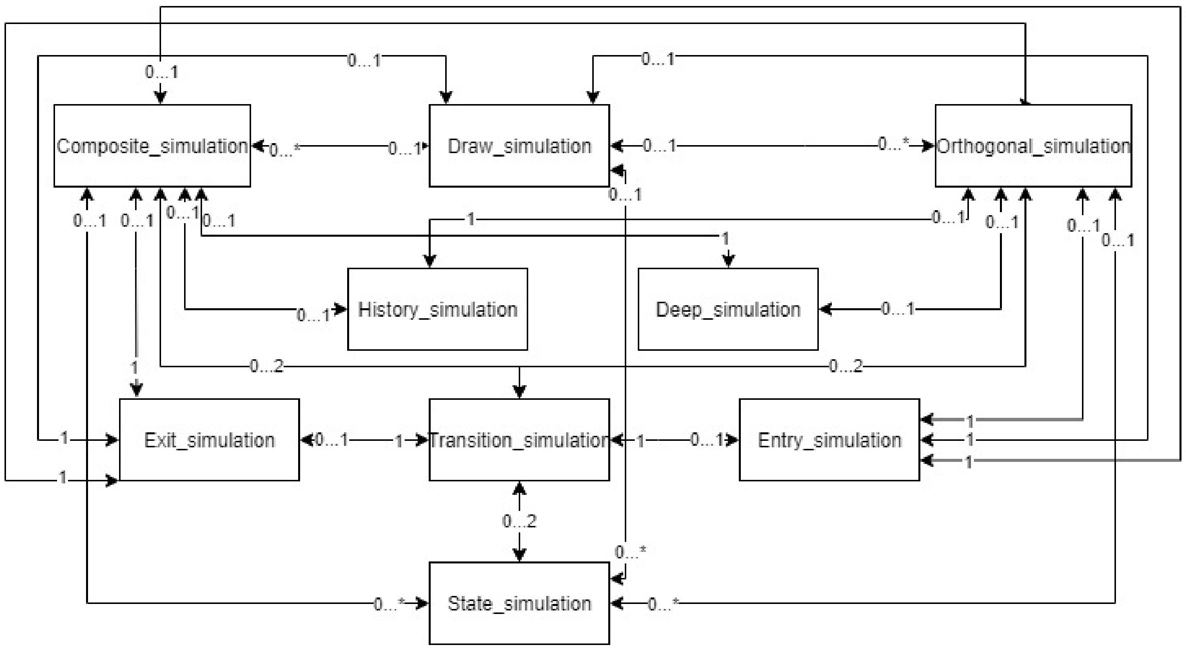


Figure 4.5: Statecharts simulation UML class diagram



orthogonal or *composite* instances and a maximum of one *entry* and *exit* instances, and the same is true for the opposite directions of the associations except for *entry* and *exit* that have to be associated to one transition.

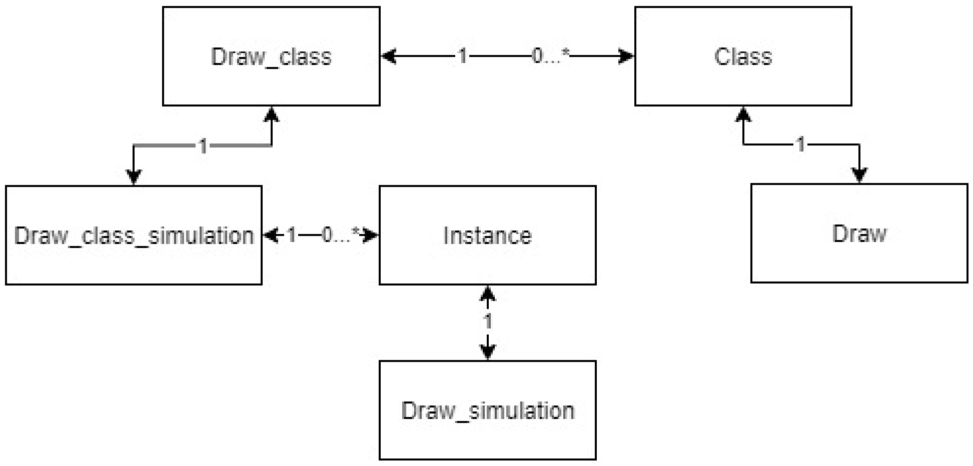
4.3.1.3 Object diagram simulation

The UML class diagram of the object diagram simulator (Fig. 4.4) is the same as the diagram of the class diagram editor in 4.3.1.1. The only difference is that the *association* and *selection* classes are omitted. The simulation of the system should not allow the developer to edit it, but to only observe and interact with it.
Therefore, the object class simulation contains the *draw_class_simulation* and *instance* classes which have associations between each other similarly to the *draw_class* and *class* classes explained in 4.3.1.1.

4.3.1.4 Statecharts simulation

Similarly to the object diagram simulation explained in 4.3.1.3, the statecharts simulation UML class diagram (Fig. 4.5) is the same as the statecharts UML class diagram explained in 4.3.1.2.
Therefore, the statecharts simulation contains the *draw_simulation*, *composite_simulation*, *orthogonal_simulation*, *deep_simulation*, *history_simulation*, *entry_simulation*, *transition_simulation*, *exit_simulation* and *state_simulation* classes which have associations between each other similarly to the

Figure 4.6: Associations between all the parts of the system



draw, *composite*, *orthogonal*, *deep*, *history*, *entry*, *transition*, *exit* and *state* classes explained in 4.3.1.2 respectively with the *selection* class omitted.

4.3.1.5 Associations between all the parts of the system

Fig. 4.6 shows how these 4 parts of the system are connected together/

- *Class* is the class representing the classes added by the editor to the class diagram editor and *Draw* is the class representing the UI canvas of the editor in the statecharts editor. Both of these classes have a single association between them to imitate the behavior of SCCD.
- *Draw_class* is another class of the class diagram editor which represents its UI canvas, and *draw_class_simulation* is the class representing the UI canvas of the object diagram simulator. Both of these classes have a single association between them as explained in the point above.
- Similarly to the *class* and *draw* classes, *instance* of the object diagram simulator and *draw_simulation* of the statecharts simulator have a single association between them.

4.3.2 TopLevel

The toplevel of the SCCD system being created contains a list of all the libraries that are imported and, most importantly, it contains an inport named "input" which is used to send the user UI events as explained in the next subsection.

4.3.3 Event bindings

Since the created system is UI based and needs to allow interactions between the developer and the elements created on the canvas, the Tkinter event bindings are used to route such events into the controller of the system.
The event binding can be done like the following:

$$canvas.bind("<Button>", self.on\_click)$$

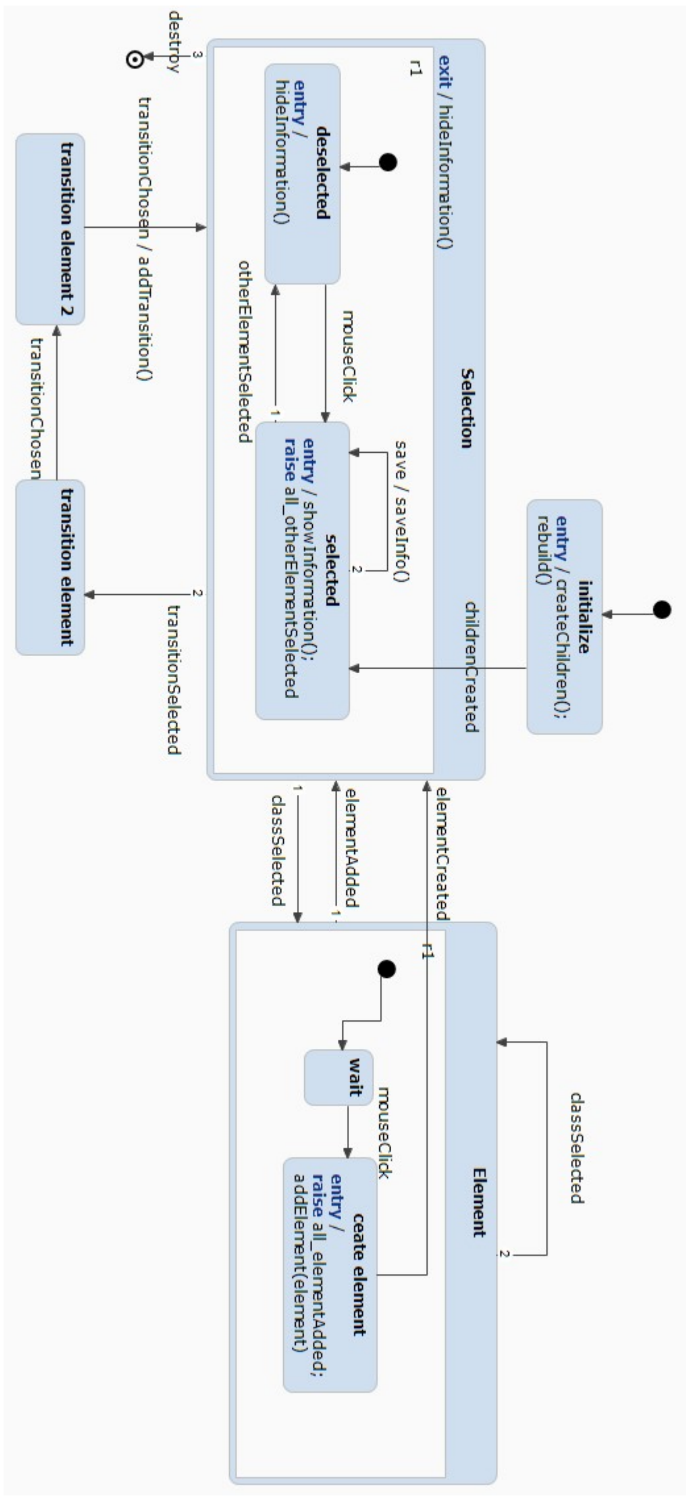


Figure 4.7: Statecharts of the class draw_class

This line of code binds the mouse click of the canvas to a function the *on_click* function which sends this event to the system using the *input* inport that was added to the toplevel of the system. The event can be sent back to the system using

$$controller.addInput(Event("mouseClick", "input", [\,]))$$

Where the first attribute in *Event*() represents the name of the event being raised, the second one represents the name of the inport being used, and the third one is an array that holds the parameters that the developer wants to accompany the event.
The events that have a binding in this system are:

- *Button* which fires when a mouse button is clicked, this event can be later filtered using its 'num' attribute in order to know which mouse button specifically was clicked.
- *ButtonRelease* which fires when the mouse button is released.
- *Motion* which fires whenever the mouse moves.
- *Key* which fires on the press of a keyboard key.
- $Double - Button - 1$ which fires when a double click is performed on the left mouse button.
- All the buttons added to the UI also have their on click event bound and routed to a SCCD event.

4.3.4 Statecharts diagrams

The statecharts of the different classes of the system have been developed using the approach explained in 3.15:

- The statecharts have been modeled using Yakindu's visual editor as an SCXML system because creating diagrams is easier to be done using a visual interface than a textual one.
- SCXML code is generated out of the SCXML statecharts created, then this code is edited and expanded on in order to become a completely functional SCCD system.

Therefore, the statecharts figures added in the following sections are all taken from Yakindu.
Since Yakindu doesn't offer the same class functionalities of SCCD, the events that are fired in the system have been named in a way to make it easier to translate into SCCD compatible events. The event names have the following meanings:

- *all_eventName* means that this event is broadcasted to the entire system using SCCD's *broadcast* event.
- *parent_eventName* means that this event was only raised to the parent of the element using SCCD's *narrow* event.
- *children_eventName* means that this event was only raised to the children of the element using SCCD's *narrow* event.
- *eventName* without any prefix means that this event was raised locally.

Some events, like mouse button clicks, are broadcasted to all the object instances in the system and, since the mouse click should only have an effect on the element that was under the mouse when the click happened, all broadcasted events are accompanied by two attributes which are added as conditions to all the transitions related to them:

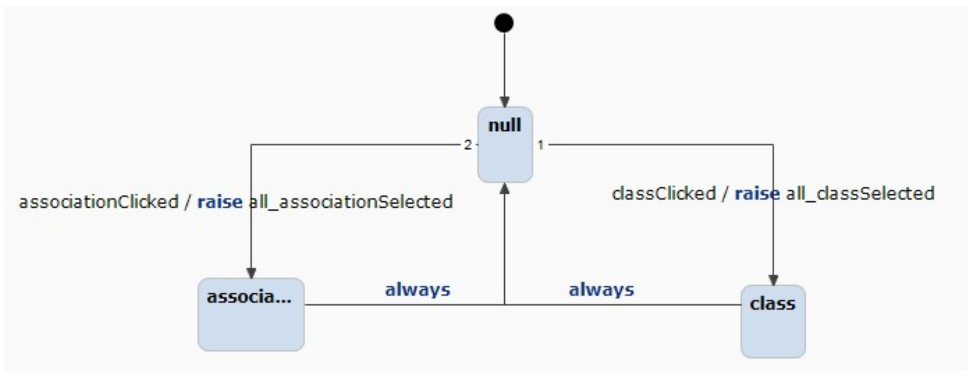
- Class id, which is the id given by the canvas to the visual representation of the class on the diagram, this helps in separating the events between the different classes and their statecharts.
- Element id, which is the id given by the canvas to the visual representation of the element added to it, this helps in separating the events between the different elements contained in the same statecharts of a class.

4.3.4.1 Class diagram editor

Draw_class statecharts Fig. 4.7 shows the statecharts of the draw_class, which represents the canvas of the class diagram editor of the system. The draw_class is the initial class of the system, it is the first one that gets instantiated when the program is run and it takes care of instantiating the other objects when needed.
The information held by the *draw_class* object is the same as the information that is saved in SCCD's top level element as explained in 2.3.1.

- When the program is first started, *initialize* is the first state that will be active. In this state, the system instantiates the *selection_class* object after which it goes to its *selection* composite states where *selected* is its initial one.
- The *selected* state is only active when the user doesn't have any class or transition selected in the class diagram. Therefore, when this object is in its *deselected* state and the user clicks on the background of the canvas, a *mouseClick* event is raised which leads it to enter its selected state where it shows the information this object holds to allow the developer to edit them: it also broadcasts an *otherElementSelected* event to all the objects of the system so they can go to their deselected state since only one element can be selected at a time.
- In the *deselected* state, the system hides the information this object holds from the developer so they become un-editable.
- When the developer wants to add a transition to the diagram, the draw_class exits its *selection* composite states and enters the *transition element* state where no action happens. When the developer chooses the origin of the transition, a *transitionChosen* event is raised which activates the *transition element* 2 state, the same happens when the developer selects the target of the transition but this time the *deselected* state gets activated and the *addTransition* function is called, which creates the visual representation of the transition on the canvas.
- When the developer wants to add a class to the diagram they're modeling, a *classSelected* event is raised which takes *draw_class* to its *element* orthogonal state where *wait* is its initial one. And, when the user clicks on the canvas, a *mouseClick* event is generated which takes the system to the *create element* state.
- In its *create element* state, the system creates and starts a new instance of the *class* object and adds its visual representation at the location of the mouse when

Figure 4.8: Statecharts of the selection_class class of the class diagram editor


null
associationClicked / raise all_associationSelected
classClicked / raise all_classSelected
2
1
associa...
always
always
class

it was clicked, it then broadcasts an *elementAdded* event which takes *draw_class* back to its *selection* composite states.

Selection_class statecharts The *selection_class* statecharts is directly related to the sidebar on the left in 4.15 which allows the developer to choose the elements to add to the class diagram. The statecharts of *selection_class* can be seen in fig. 4.8

- When this object is instantiated by *draw_class*, it activates its *null* state until it receives a click event from the user.
- When the user clicks on the association button, an *associationClicked* event is raised which will take the statecharts to its *association* state and broadcasts an *associationSelected* event to the rest of the system.
- Similarly, when the user clicks on the class button, a *classClicked* event is raised which will take the statecharts to its *class* state and broadcast the *classSelected* event to the rest of the system.
- The statecharts immediately goes back to its null state after raising the *associationSelected* or *classSelected* events to allow the developer to change the selected element to add it to the diagram without having to place the first one before.

Class statecharts The *class* statecharts represents the classes added by the developer to the system they're modeling.
The class object holds the following information:

- The *name* of the class.
- The *attributes* the class defines.
- The *inheritances* the class has.
- The *constructor* and *destructor* methods of the class.
- an *initial* checkbox which sets the class as the initial one in the system.
- Any other user defined *methods*

The statecharts of this class can be seen in fig. 4.9.

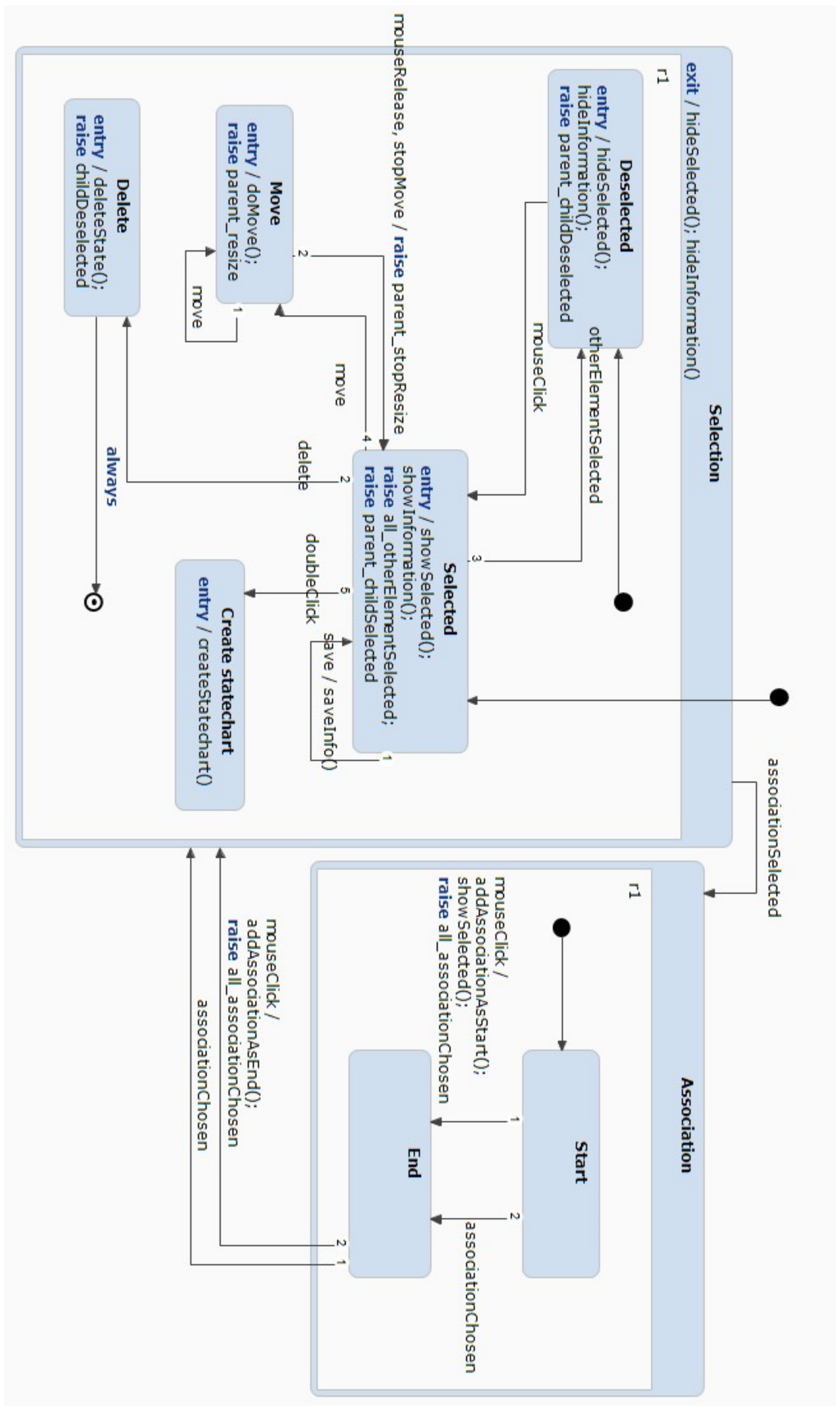
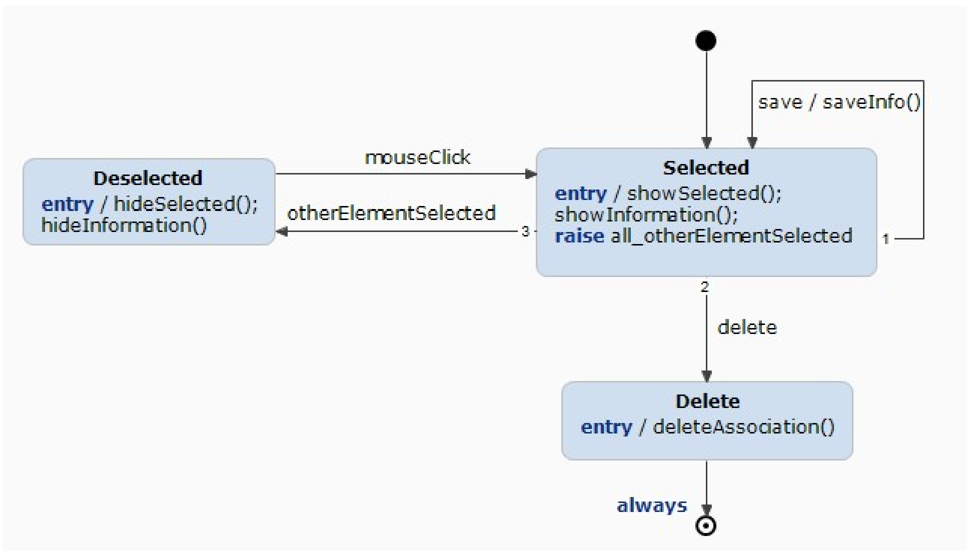

Selection
exit / hideSelected(); hideInformation()
r1
Deselected
entry / hideSelected(); hideInformation(); raise parent_childDeselected
otherElementSelected
mouseClick
Selected
entry / showSelected(); showInformation(); raise all_otherElementSelected; raise parent_childSelected
save / saveInfo()
doubleClick
Create statechart
entry / createStatechart()
delete
move
mouseRelease, stopMove / raise parent_stopResize
Move
entry / doMove(); raise parent_resize
move
Delete
entry / deleteState(); raise childDeselected
always
associationSelected
Association
r1
Start
End
mouseClick / addAssociationAsStart(); showSelected(); raise all_associationChosen
associationChosen
mouseClick / addAssociationAsEnd(); raise all_associationChosen
associationChosen

Figure 4.9: Statecharts of the class element of the class diagram editor

- When the *class* object is first initialized, it goes into its *selected* state in its *selection* composite states. In this state, the system changes the color of the class' visual representation on the diagram to make it visible for the developer which element is selected, it displays the object's information on the side of the canvas, and it raises two events: *childSelected* event that is raised to the parent of the class, and *otherelementSelected* that is broadcasted to all the object instances in the running system.
- When the developer clicks on another element in the canvas (or on its background), an *otherElementSelected* event is generated which takes this statechart to its *deselected* state. In this state, the system removes the color change of the class element, hides the information that was displayed on the side of the canvas, and raises a *childDeselected* event to its parent. The statecharts goes back to its selected state if the user clicks on the visual representation of this element on the canvas.
- The combination of having the mouse left button held down and the mouse moving fires the *move* event which leads the statecharts into its *move* state. In this state, the system changes the location of the class being moved according to the mouse location, which continues for as long as the mouse button is held down because of the transition that fires on a *move* event and leads the statecharts back into its *move* state. The statecharts only exits this state and goes back into its *selected* state if the user releases the mouse button.
- When the statecharts is in its *selected* state:
  - If the developer clicks on the save button, a save event is fired which calls the $saveInfo$ function (explained in 4.6).
  - If the developer presses the delete button on the keyboard, a delete event is generated which takes the statecharts to its delete state, deletes the element, then goes to its final state.
  - If the developer double clicks on the class, a *doubleClick* event is generated which creates and starts an instance of the *draw* class which will open a new window containing the statecharts editor of the selected class.
- When the developer clicks on the association button in the selection object, an *associationSelected* event is generated which takes the statecharts to its *association* composite states where start is its initial one.
- If the statecharts is in its *start association* state and the developer clicks on the class, a new association is created as a child of this object, the color of the class representation changes to show that it was indeed selected, and an *associationChosen* event is broadcasted in the system. If the statecharts is in its *end association* state and the developer clicks on the class, the same thing happens but the transition already created gets linked to the class using the *associate_instance* event of the object manager. The second *associationChosen* event leads the statecharts out of its *association* composite states and back into its *selection* composite state where *deselected* is its initial state in case this class wasn't chosen as the target of the added association.

Figure 4.10: Statecharts of the association class of the class diagram editor


save / saveInfo()
mouseClick
Deselected
entry / hideSelected(); hideInformation()
otherElementSelected
3
Selected
entry / showSelected(); showInformation();
raise all_otherElementSelected
1
2
delete
Delete
entry / deleteAssociation()
always

Association statecharts The *association* statecharts represents the associations added by the developer to the system they're modeling.
The information that the association holds are:

- *Name* of the relationship linking the parent to the child object along with its *min* and *max* cardinality.
- *Name* of the relationship linking the child object to its parent along with its *min* and *max* cardinality.

The statecharts of this class can be seen in fig. 4.10

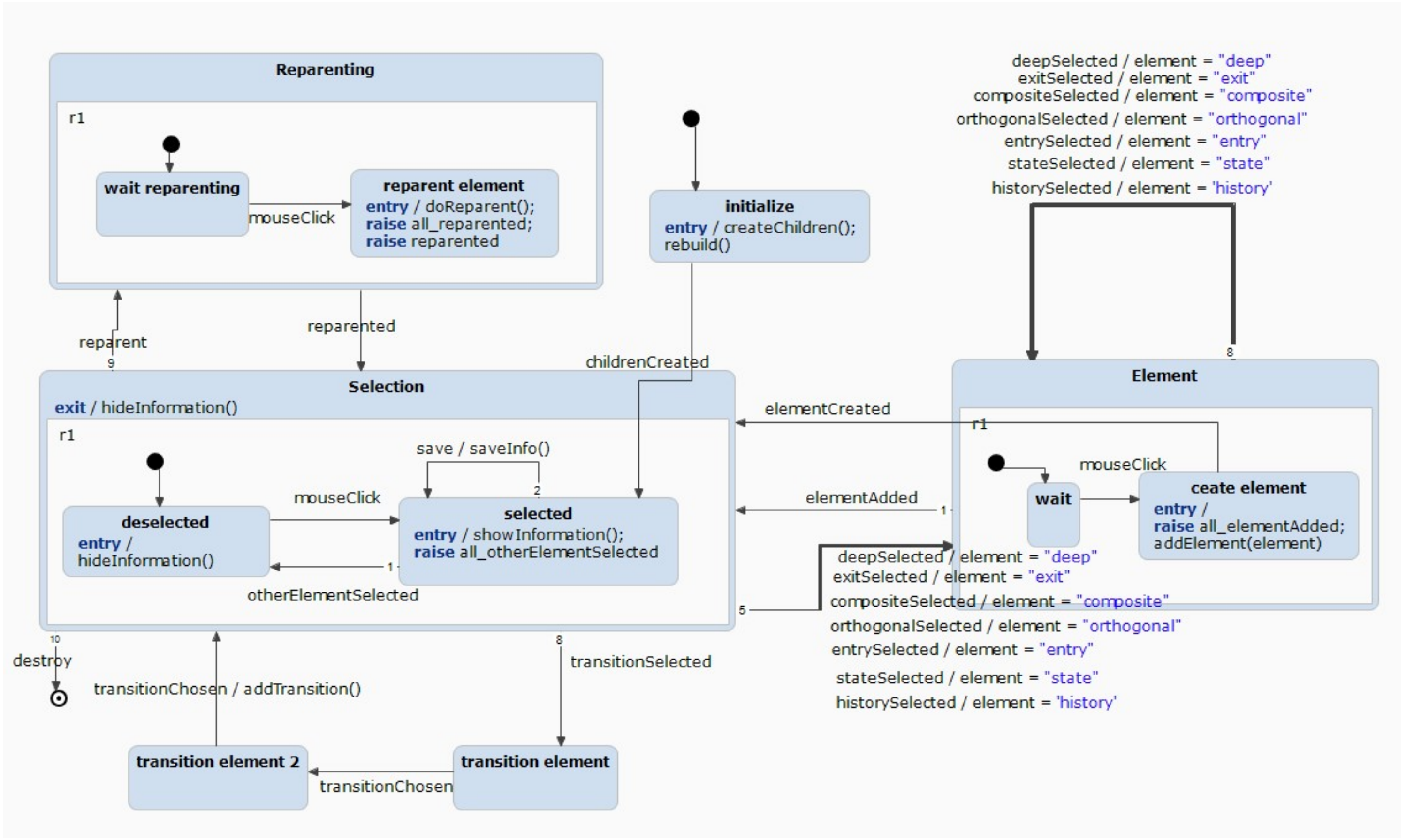
- When the association object is first initialized, it goes into its *selected* state where the color of its visual representation is changed, its information are displayed on the side of the canvas, and an *otherElementSelected* event is broadcasted to the system.
- If the *selected* state is active and the developer presses on the delete button, the transition that leads the statecharts from its *selected* state to its *delete* state fires, which deletes the object with its visual representation.
- When the developer clicks on any other element on the canvas, the statecharts goes into its *deselected* state where it changes the visual representation of the association back to its original color and hides its information from the frame. The statecharts will go back to its active state once the developer clicks on the visual representation of this association.

4.3.4.2 Statecharts editor

Draw statecharts The *draw* statecharts of the statecharts editor is the same as the *draw_class* statecharts of the class diagram editor as shown in fig. 4.11. The only differences are:

- In its *initialize* state, the system not only creates and starts the *selection* object instance, but it also rebuilds its created statecharts elements. This function is

Figure 4.11: Statecharts of the draw class of the statecharts editor



Reparenting
r1
wait reparenting
mouseClick
reparent element
entry / doReparent();
raise all_reparented;
raise reparented
initialize
entry / createChildren();
rebuild()
deepSelected / element = "deep"
exitSelected / element = "exit"
compositeSelected / element = "composite"
orthogonalSelected / element = "orthogonal"
entrySelected / element = "entry"
stateSelected / element = "state"
historySelected / element = 'history'
reparent
reparented
childrenCreated
Selection
exit / hideInformation()
r1
save / saveInfo()
mouseClick
deselected
entry /
hideInformation()
selected
entry / showInformation();
raise all_otherElementSelected
otherElementSelected
Element
r1
mouseClick
wait
ceate element
entry /
raise all_elementAdded;
addElement(element)
elementCreated
elementAdded
deepSelected / element = "deep"
exitSelected / element = "exit"
compositeSelected / element = "composite"
orthogonalSelected / element = "orthogonal"
entrySelected / element = "entry"
stateSelected / element = "state"
historySelected / element = 'history'
destroy
transitionChosen / addTransition()
transitionSelected
transition element 2
transitionChosen
transition element

explained in 4.7.2. For example, if the developer created a class named 'class1', opened its statecharts, edited it, then closed the statecharts window, when this window is opened again, all the elements that have been previously created will need to be rebuilt so the developer can edit them.

- The *reparenting* composite state becomes active when a *reparent* event is received, this event is created by the system as a response to a mouse right click on the element; and it becomes inactive again when a *reparented* event is received, which is raised by the *draw* class itself. When *reparenting* is active, it waits in its initial state *wait reparenting* until a *mouseclick* event is received where the new visual representation of the element is created in its correct place. This feature allows the developer to change the parent of any element in the system being developed without having to delete the element completely and recreate it inside of its correct parent.

State statechart The *state* statechart of the statecharts editor is the same as the *class* statecharts of the class diagram editor. The only difference lies in the following states:

- *ChildSelected* and *childDeselected* events are raised to the parent of the object depending on its active statecharts state.
- When the statecharts is in its *move* state, a *resize* event is raised to the parent (which can be *draw*, in this case nothing happens, or *composite* or *orthogonal* where the parent's visual representation is resized to keep the child's visual representation inside of it.
- When the visual representation of the parent of a state instance is moving, the statecharts goes into its *parentMoving* state where the state itself also moves as well. This makes sure that the location of the state element stays the same relatively to its parent even when the latter is moved.
- If the state is selected and the developer clicks the right mouse button on it, a *rightClick* event is generated which activates the *reparent* state, in this state, the visual representation of the element is deleted along with its connection with its parent. The state is placed again on the canvas when the developer presses the left mouse button (handled in the *draw* statecharts) and an association with its new parent is created.

The information held by the state object are:

- The unique *name* identifier of the state.
- The *on entry* and *on exit* actions which include both the methods and events raised.

Selection statecharts The *selection* statecharts of the statecharts editor is similar to the *selection_class* of the class diagram editor as shown in fig. 4.13. It has the same transitions between its different states and with the same effects on the system. The only difference is that, instead of containing the *class* and *association* elements which are specific to the class diagram editor, the *selection* contains the statecharts elements which are the *state*, *history*, *deep*, *orthogonal*, *composite*, *entry*, *exit* and *transition*.

Figure 4.12: Statecharts of the state class of the statecharts editor

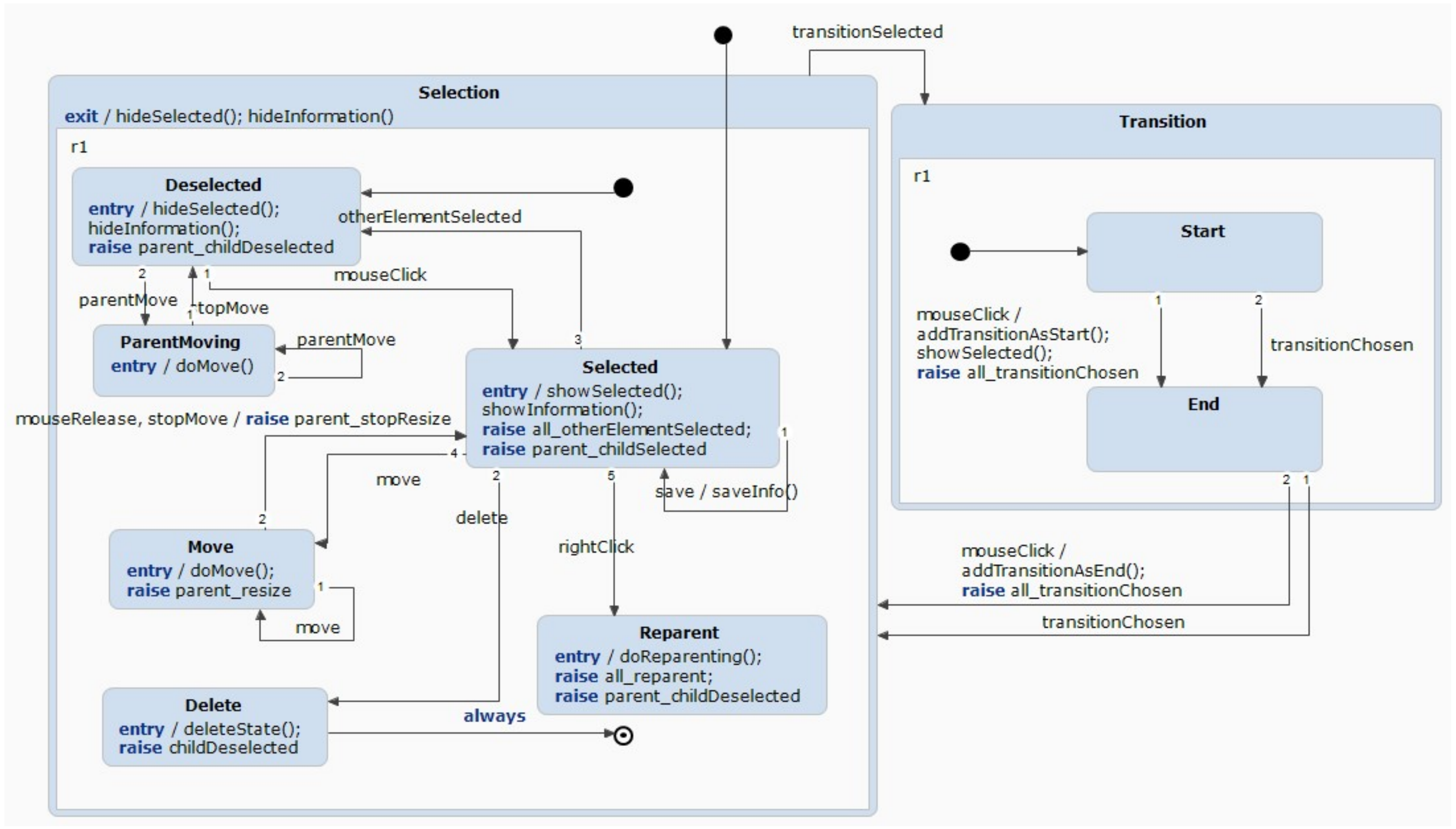
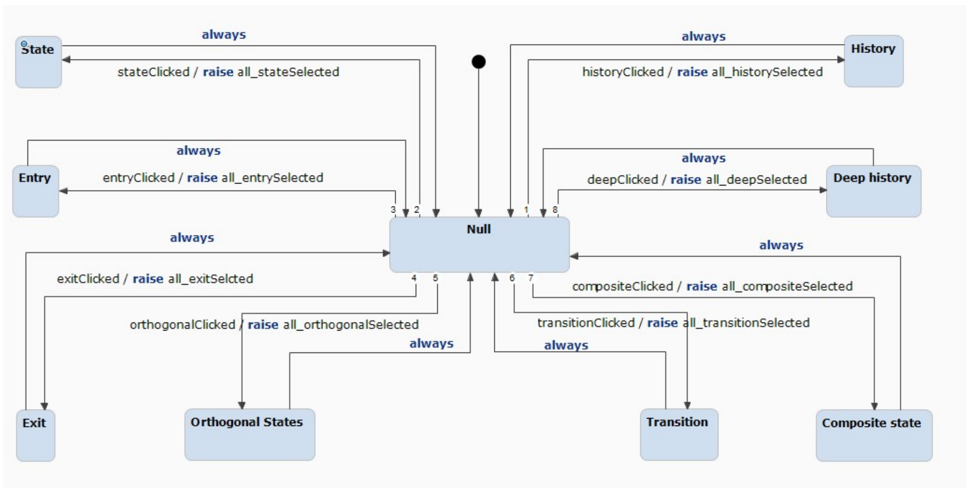


Figure 4.13: Statecharts of the selection class of statecharts editor


State
always
stateClicked / raise all_stateSelected
always
History
historyClicked / raise all_historySelected
always
Entry
entryClicked / raise all_entrySelected
always
deepClicked / raise all_deepSelected
Deep history
always
Null
always
exitClicked / raise all_exitSelcted
compositeClicked / raise all_compositeSelected
orthogonalClicked / raise all_orthogonalSelected
always
transitionClicked / raise all_transitionSelected
always
Exit
Orthogonal States
Transition
Composite state

Exit, entry, deep and history statecharts The statecharts of these four elements are similar to the *state* statechart as explained in 4.3.4.2, the differences are the following:

- The *entry* statechart only contains the *selection* composite states of the *state* statecharts and, instead of the *transition* composite state, *entry* only has the *start* state of this composite element since an entry can only be the origin of the transitions.
- The *exit*, *history* and *deep* statecharts have the same *selection* and *transition* composite states of the *state* statecharts, but the difference is that the *transition* composite states only allows the developer to add the element as the target of the transition since none of them can be the origin of the transitions in SCCD.

Composite and orthogonal statecharts Both the *composite* and the *orthogonal* statecharts are a combination of the *selection* and *transition* composite states of the *state* statecharts and the *element* and *reparenting* composite states of the *draw* statecharts as explained in 4.3.4.2. Additionally, these two statecharts also have the *child* composite state as seen in fig. 4.14, this composite states becomes active if any child of this element is selected, and it takes care of resizing the visual representation of the element if the child element is moved around; it makes sure that the child element never leaves the borders of its visual representation.
Additionally, one of the main advantages of developing a GUI using statecharts is present in these two elements, and it is the bubbling of the events through the hierarchy of the visual elements:

- When any child element is selected, a *childSelected* event is raised, which leads the statecharts of the composite and orthogonal elements to their *child* state. On entry of this state, the same *childSelected* event is raised to the parent of this element, which in turn, if applicable, raises the event to its parent as well. Therefore, if the developer was to move a state inside of a composite states element that is itself inside of another element, both elements will resize according to the state being moved on the canvas.
- The same thing happens in the opposite direction, when a parent hierarchy that

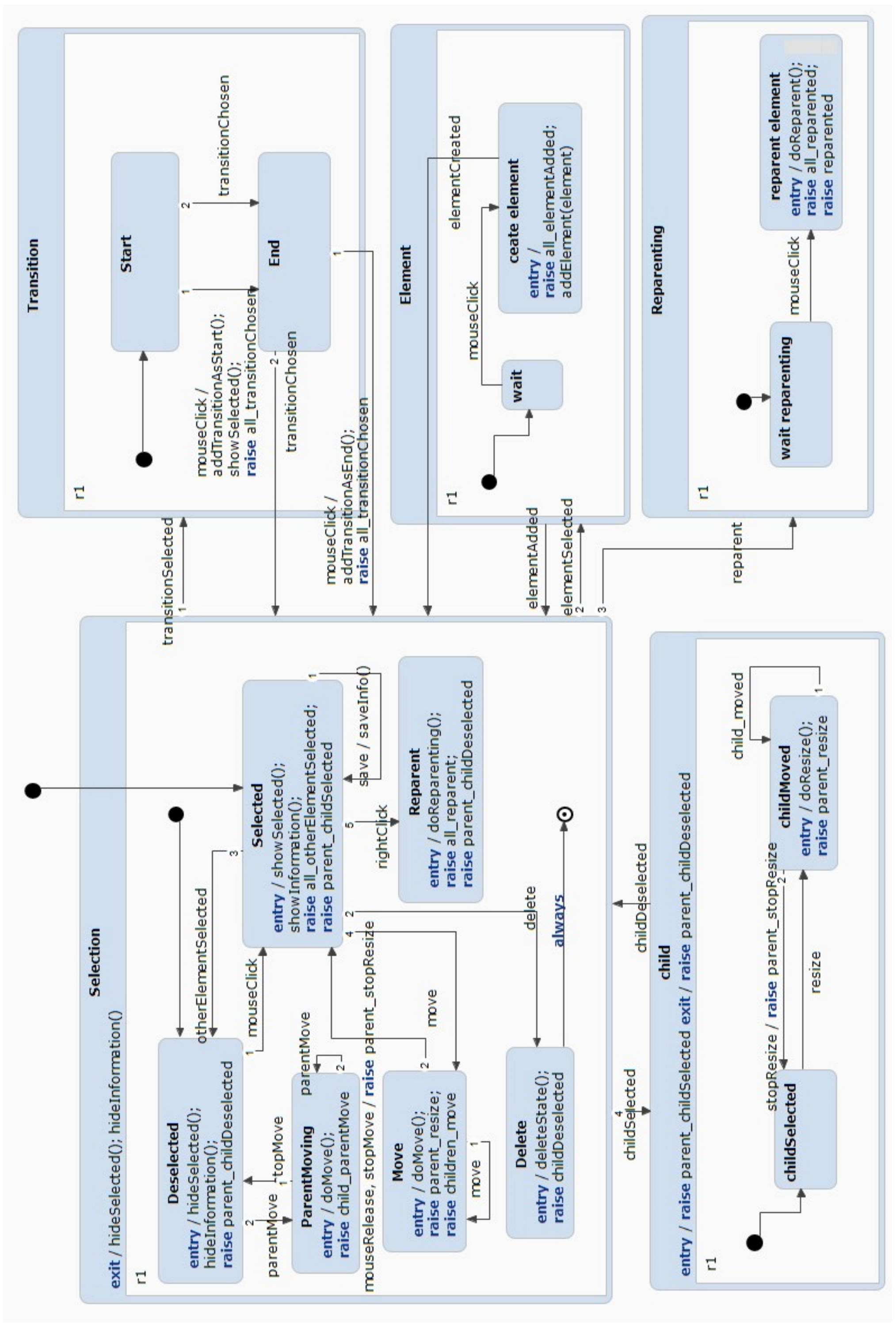

Transition
r1
Start
End
transitionChosen
mouseClick / addTransitionAsStart(); showSelected(); raise all_transitionChosen
mouseClick / addTransitionAsEnd(); raise all_transitionChosen
transitionSelected
Element
r1
wait
mouseClick
ceate element
entry / raise all_elementAdded; addElement(element)
elementCreated
elementAdded
elementSelected
Reparenting
r1
wait reparenting
mouseClick
reparent element
entry / doReparent(); raise all_reparented; raise reparented
reparent
Selection
exit / hideSelected(); hideInformation()
r1
Selected
entry / showSelected(); showInformation(); raise all_otherElementSelected; raise parent_childSelected
save / saveInfo()
rightClick
Reparent
entry / doReparenting(); raise all_reparent; raise parent_childDeselected
Deselected
entry / hideSelected(); hideInformation(); raise parent_childDeselected
otherElementSelected
mouseClick
ParentMoving
entry / doMove(); raise child_parentMove
parentMove
topMove
Move
entry / doMove(); raise parent_resize; raise children_move
move
mouseRelease, stopMove / raise parent_stopResize
Delete
entry / deleteState(); raise childDeselected
delete
always
childDeselected
childSelected
child
entry / raise parent_childSelected exit / raise parent_childDeselected
r1
childSelected
childMoved
entry / doResize(); raise parent_resize
child_moved
stopResize / raise parent_stopResize
resize

Figure 4.14: Statecharts of the composite class of the statecharts editor

contains another hierarchy containing a state is moved, the *parentMove* event is bubbled downwards in the hierarchy which results in all 3 elements moving together and staying at the same location relatively to the mouse.

Transition statecharts The statecharts of the *transition* class in the statecharts editor is the same as the statecharts of the *association* class in the class diagram editor as explained in 4.3.4.1.
The information held by the *transition* object are:

- The name of the *event* that fires the trigger, or the time *after* which the transition is triggered.
- The *parameters* of the transition. These parameters are used by SCCD to give a name to the parameters that accompany the event.
- The *condition* which only allows the transition to fire if it holds.
- The *script* containing the code run by the system when this transition is triggered.
- The *raise actions* which contains the list of actions to be raised by the system once this transition is triggered.

4.3.4.3 Object diagram and statecharts simulation

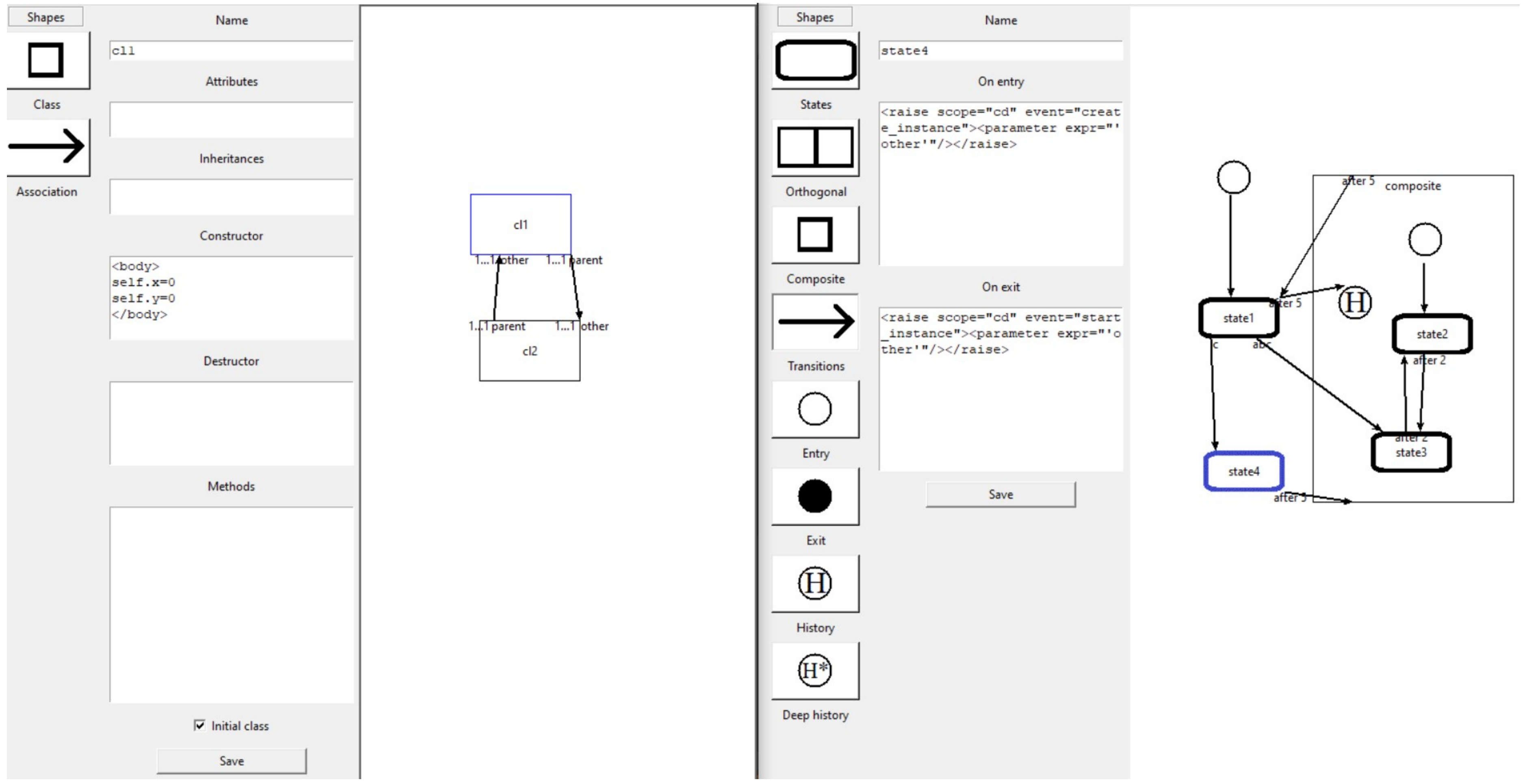
The statecharts of all the classes of the object diagram and the statecharts simulation are a copy of the statecharts of all the classes of the class diagram editor and the statecharts editor explained above. The difference is that none of these two diagrams should allow the user to edit them, therefore the statecharts of their classes only contain the *selected* and *deselected* states of the *selection* composite states. The object diagram contains additionally the *move* state to allow the developer to move the instantiated classes around the canvas to have a better view of them.

4.4 Design of the graphical user interface

The developed graphical user interface has the following design decisions:

- The root, which is the first window that opens up when running the system, is the class diagram editor. From this window the developer can save, load and export the system. From this window, the statecharts editor can also be opened and the simulation can be run and its windows can be opened.
- The developer can double click on a class in the class diagram editor to open its statecharts diagram. The new windows that opens is a Tkinter *topLevel* window. (the same applies for any other window that opens in the system like the object diagram and statecharts simulation). The name of the statecharts editor window is the same as the name of the class that holds this statecharts, this makes it easier for the developer to identify which class they're working on when multiple windows are opened.
- All the windows contain the selection to the left, where the developer can choose new elements to add to the diagram, the canvas to the right where all the visual

Figure 4.15: UI design of the running system



Name: cl1

Constructor:

```
<body>
self.x=0
self.y=0
</body>
```

[✓ Initial class]

Name: state4

On entry:

```
<raise scope="cd" event="creat
e_instance"><parameter expr="'
other'"/></raise>
```

On exit:

```
<raise scope="cd" event="start
_instance"><parameter expr="'o
ther'"/></raise>
```

representations of the elements will be drawn, and the information of the selected element in between.

- The information of the element selected contains Tkinter *labels* and *entry* widgets, which show the name of the field and allow to edit its content.
- The classes added on the canvas are rectangles drawn by Tkinter, the transitions and associations are lines with an added arrow, and all the other elements are images that are added to the canvas since Tkinter only allows to create simple shapes programatically. All these elements are colored black when deselected, and blue when they're selected.
- All the elements added to the diagrams have their unique identifier name displayed inside of their visual representation. The transitions display the event or the time after which they trigger in the middle of their line. And the associations display the min and max cardinality on both ends of the line with the name of the relationship.

Fig 4.15 shows a simple made-up system designed using the newly developed visual editor for SCCD. It shows two windows opened side by side: The class diagram editor to the left, and the statecharts editor of the class cl1 to the right.

4.5 SCCXML language in the visual editor

SCCD has its own xml based language called SCCXML as explained previously, but with the new visual editor, this language becomes obsolete during the development of the systems (but is still relevant for their deployment) because the textual aspect of the development is transformed into a visual editor instead.
Even though the design of the systems is now done graphically instead of visually, some parts of the system still need to contain written code, like adding methods to the classes, or raising events in the statecharts. For this reason, a textual programming language is still needed in some places.
Instead of having to develop a new language for these elements from scratch, a simplified version of SCCXML is used when needed, this simplified version omits some of the xml tags since they are replaced by their respective entry widgets. A comparison of some of the SCCD SCCXML language and the SCCD visual editor language is shown in table 4.1

4.6 Saving the system

The *draw_class* of the class diagram editor contains a button named 'save project', which, as its name suggests, opens a window to allow the developer to name the file containing the project, and choose the directory in which to save it.
The system being developed is saved as a text file containing all of its information. This save function follows the pseudo-code in listing 5 to generate the text file.

Name	Old SCCXML	New SCCXML

Constructor (similar to the destructor)	`<constructor>`
	  `<body>`
	    `content`
	  `</body>`
	`</constructor>`
		`<body>`
		  `content`
		`</body>`

Method	`<method name=''>`
	  `<parameter name=''/>`
	  `<body>`
	    `content`
	  `</body>`
	`<method name=''>`
		`<name=''>`
		  `<parameter name=''/>`
		  `<body>`
		    `content`
		  `</body>`
		`<method name=''>`

On entry (similar to on exit)	`<onentry>`
	  `<raise scope='' event=''`
	    `>`
	    `<parameter expr=''/>`
	  `</raise>`
	  `<script>`
	    `content`
	  `</script>`
	`</onentry>`
		`<raise scope='' event=''>`
		  `<parameter expr=''/>`
		`</raise>`
		`<script>`
		  `content`
		`</script>`

Table 4.1: Example of the differences between the old and new SCCXML

Listing 5: Python pseudo-code of the save function.

```
class Empty:
    name = ""

output = ""
output += jsonpickle.encode(topLevelInformation)

#saveClasses():
for class in self.classes:
    newClass = Empty()
    copyInfo(class, newClass)
    copyCoordinates(class, newClass)
    saveElements(class, newClass)
    output += jsonpickle.encode(newClass)

def saveElements(parent, newParent):
    for element in parent.elements:
        newElement = Empty()
        copyInfo(element, newElement)
        copyCoordinates(element, newElement)
        if element.hasChildren:
            saveElements(element, newElement)
        newParent.elements.append(newElement)
```

The system starts by adding the top level information as a JSON encoded empty class using jsonpickle (jsonpickle is a python library that allows the serialization and deserialization of complex Python objects to and from JSON). Then it loops through all the classes that have been created, creates an empty class as a copy of each, and copies the relevant information and its coordinates on the canvas. The relevant information of the classes and any other element copied are the same as the SCCD related information shown to the developer on the side of the canvas as explained in 4.3.4. A copy of the class is created with only the relevant information because SCCD saves a lot of data that isn't relevant to the saved file since everything will be recreated once the project is loaded.
Since some elements of the statecharts, like the composite states and orthogonal regions, can contain children elements of their own, and this relationship needs to be kept when the system is saved and loaded, the save elements function is called again for every occurrence of these elements in the system so that each one of them can keep an array of all of its children. Therefore, the final file would be a JSON string that similar to the following example of a class containing one composite states that has 2 states inside of it:
$'\{class : information, [\{state1 : information\}, \{composite : information, [state2 : information]\}]'$

4.7 Loading the system

The *draw_class* of the class diagram editor contains a button named 'load project' as well as the 'save project' button. This button allows the developer to load a project already created using the SCCD visual editor to either edit it or run its simulation.
The loading of a project is separated into two parts: loading of the class diagram and loading of the statecharts.

4.7.1 Loading the class diagram

The system uses the saved file to load the class diagram, it starts by decoding the JSON string back into *empty* objects, and loops through these objects to recreate each one of them.
Since the save system only saves the relevant information of the elements of the project, the actual instances of each element have to be re-instantiated when loading, and the information has to be copied into these new instances. The reason the entire instances aren't saved and loaded directly without removing any information is because that would cause errors with both Tkinter and the SCCD controller mainly due to having different instances of each.
Instead of having to model the loading function in SCCD statecharts in order to create and start the instances of the loaded elements using the SCCD events, we can use the python code that is generated from SCCXML and raise the events directly in our programmed loop. The python code to create and start an instance is the following respectively:

$$controller.object\_manager.handleCreateEvent([parent, association\_name,$$
$$target\_association\_class\_name, additional\_parameters])$$
$$controller.object\_manager.handleStartInstanceEvent([parent, association\_name])$$

Therefore, the pseudo-code of the load function is shown in listing 6

Listing 6: Python pseudo-code of the load function.

```
elements = jsonpickle.decode(loadedTextFile)
topLevelInformation = elements[0]
copyTopLevelInformation(topLevelInformation)
for element in elements - [topLevelInformation]:
    class = createAndStartClassInstance()
    copyClassInformation(element, class)

def copyClassInformation(old, new):
    new.releventInformation = old.releventInformation
    new.statecharts = old.statecharts
```

The function starts by decoding the loaded text file and looping through it, it copies the information from the first element to its top level data. Then, the function goes through the rest of the elements, creating and starting a new class instance for each, adding their visual representations to the canvas, and copying their information. The information that is copied for the classes is both the relevant information and the list of the statechart elements as they are, as in a list of empty class objects.

4.7.2 Loading the statecharts diagrams

The loading system, at first, only initializes the class diagrams, and leaves the statecharts as an untreated list of elements contained in their respective classes. This is done because this method reuses the same system used to load the statecharts diagram of each class whenever the developer double clicks on a class to edit its elements. Therefore, the pseudo-code of this loading function (or better called rebuild function) is shown in listing 7.

Listing 7: Python pseudo-code of the rebuild function.

```
def rebuild(elements, parent)
    for element in elements:
        new = createAndStartInstance(element, parent)
        copyElementInformation(element, new)
        if new.isComposite or new.isOrthogonal:
            rebuild(new.elements, new)
```

The function loops through the elements of the class and copies their information to newly created and started instances of them. And, if the element is an orthogonal regions or composite states element, then the same function is repeated for their elements, but with the newly created instances as parents.

4.8 Exporter

The system also contains an export button that allows to generate the SCCXML code of the system designed so it can be compiled and deployed. This export button calls a function that transforms all the classes and their statecharts into a fully functional SCCD project. The export function is shown in listing 8

Listing 8: Python pseudo-code of the export function.

```
output = genericCode
output += topLevelInformation(topLevelInformation)
for class in classes:
    output += classInformation(class)
    routes = List()
    for element in class.elements:
        routes.append(generateRoutes(element))
    output += statechartsInformation(class.statecharts, routes)

output += genericCode

def generateRoutes(element):
    routes = List()
    while element.hasParent:
        element = element.parent
        routes.add(element)
    return routes
```

Other than going through all the classes and their statecharts and transforming their information into SCCXML compatible code; the function has to also generate the routes, which are a list of the element and its parents, of each statechart element. The routes are essential to know how the transitions are related to the different elements of the system. In SCCD, the names of the target states are added to the event with a state reference string that identifies the states' hierarchical locations. Therefore, generating routes is essential in order to have a correct SCCXML file:

- '.' references the state itself.
- Empty string, references the root (the <scxml> element).

- '..' goes up one level to the parent state.
- 'a' if a is the child with id = 'a'.
- 'a/b' where a and b are the ids of the parent and its child target state.

4.9 Simulation

The classes and statecharts of the simulation environment inside of the system have been already explained above, in this section we will dive deeper into how the simulation of the system is created, how it functions, and how the developer can interact with it. We will start by explaining how the running simulation code interacts with its visual representation, then we will explain how the opposite is done which allows the developer to raise events in the simulation.
Fig. 4.16 shows a made-up example of a running simulation in the system with its object diagram on the top-left showing the instances of the newly created objects, and the statecharts diagram of both instances on the bottom-left and on the right with their active state and transitions that can fire as explained in the following subsections.

4.9.1 Creation of the running simulation code

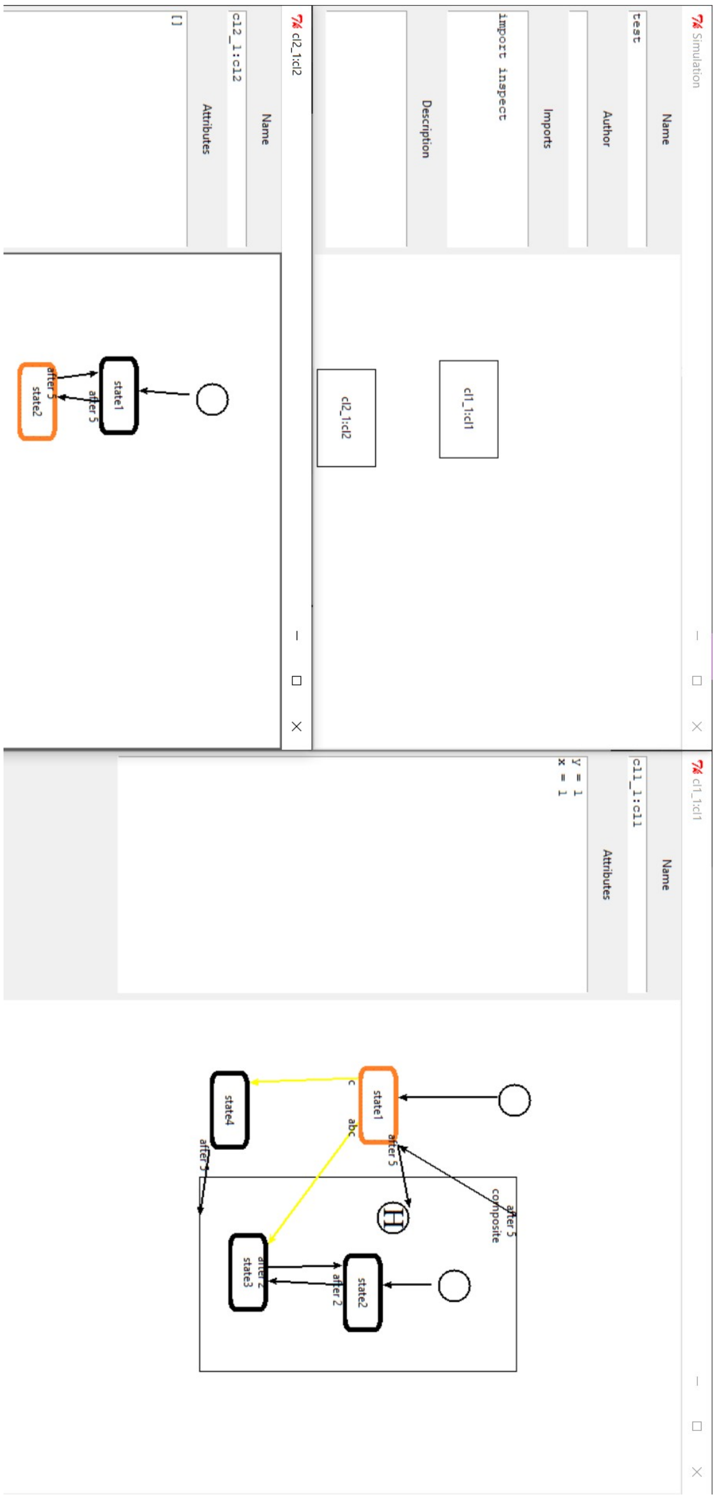
The simulation is a visual representation of the developed system's generated code running in the background. It shows all the instances of the objects that have been created in the object diagram. It also shows their running statecharts in the statecharts simulation, that is, the statecharts simulation shows the changes of the data contained in the object instance and the changes between the active states; it also allows the developer to interact with the system by raising events and watching their effects on the program.
A 'simulate' button is available in the class diagram editor and, when clicked, an edited version of the system is created, routes are created, the listener ,which connects the running program to the simulation diagrams, is instantiated, and the generated program is run.

4.9.1.1 Generation of the edited system for the simulation

When the 'simulate' button is pressed, the system starts first by exporting the SXCXML code of the developed system in order to run it. The exported code is similar to the system developed, but it also adds some code to the states, the classes and the transitions in order for them to connect correctly with the visual simulation environment.
The code that is added to the constructor of the classes is shown in listing 9 and the ones added to the states of the statecharts are shown in listings 10 and 11.

Listing 9: Python code added to the constructor of the classes.

```
if className in controller.elementInstances:
        controller.elementInstances[className] =
            controller.elementInstances[className] + 1
        instance = controller.elementInstances[className]
else:
        controller.elementInstances[className] = 1
        instance = 1
msg = className + ' create'
```



Simulation — Name: test; Author:; Imports: import inspect; Description:

cl2_1:cl2 — Name: cl2_1:cl2; Attributes: []

cl1_1:cl1 — Name: cl1_1:cl1; Attributes:

```
y = 1
x = 1
```

Figure 4.16: Example of a running simulation

```
controller.writerQueue.put(msg)
```

The code added to the constructor of the classes mainly serves to create a correct name for the created object instance. This is done by checking the number of instances of the object that are already created and adding one to them, the final name of the instance is similar to $className\_instanceNumber : className$. Then a 'create' message is added to the queue so that the newly created instance can also be shown in the object diagram. (The job of the queue is explained in 4.9.2). This code runs only once when the class object is instantiated since it is only added to the constructor of this class.
Additionally, the class' destructor is modified to add a 'delete' message to queue. This message is a simple concatination of the routing name of the object and the 'delete' keyword and it is only run when the class object is deleted.

Listing 10: Python extra code added to the on entry script of the states.

```
routingName = getRoutingName()
attributes = []
for attr in self.__dict__:
    if not attr in listOfSCCDAddedAttributes:
        attributes.append({attr: self.__dict__[attr]})
d = routingName + " entered " + str(attributes)
self.controller.writerQueue.put(d)
```

The code added to the on entry script of the states mainly serves to send a message to the listener. It goes through the list of its attributes and only saves the ones that have been added by the developer. Then an 'entered' message is added to the queue accompanied by the list of attributes and their values.

Listing 11: Python extra code added to the on exit script of the states.

```
route = getRoutingName()
msg = route + " exited"
self.controller.writerQueue.put(msg)
```

The code added to the on exit script of the states only serves to send an 'exited' message to the listener.
As for the transitions, the extra code is added to their condition and it stops the transition from firing in case the fired event belongs to a different instance of the class object.

4.9.1.2 Creation of the routing names for the system elements

Both the visual representation of the simulation and its running program have the same routing names that identify each element of the system (class, state, history...). These routing names are used to identify exactly which element of the different class instances is created, entered or exited. On the visual representation of the simulation side, routing names are created by the function called by the 'simulate' button. Each element of the system is given a routing name which is a combination of its class object's instance name and the name of the state itself. This is done because all the elements in the same class have unique names in SCCD, so two states in the same class cannot have the same name even if they're in a different parent hierarchy.
On the running program of the simulation side, routing names are created whenever a

new message needs to be added to the queue. The *getRoute* function shown in listing 10 generates the routing name by concatenating the class object's instance name and the string before the last one in the name of the function split by '_'. This is done because SCCD transforms the statecharts elements into functions, where each function is specific to only one element, and it names this function according to the hierarchy of the element. An example of such name is *_composite1_state1_enter*, where *state1* represents the actual name the developer gave to the state. The python function that generates the routing names is shown in listing 12

Listing 12: Python extra code added to the on entry script of the states.

```
def getRoutingName():
    functionName = inspect.currentframe().f_code.co_name
        .split('_')
    return self.routingName + '/' +
        functionName[len(functionName) - 2]
```

Therefore, an example of the final routing name of a state 'start' in the statecharts of the first instance of the 'car' class is *car_1 : car/start*

4.9.1.3 Running the edited system

After having generated the SCCXML of the developed system with the additional functions as explained above, the system creates a new controller for the running simulation program that uses the same Tkinter event loop as the SCCD editor and simulator tool. Running both systems in the SCCD UI event loop runtime platform and giving both of them the same event loop is necessary so that the two systems can run in parallel without causing any runtime errors. The system then generates the python code of the system created using SCCD's python code generator and starts running it in the background.
A queue, which is python's thread safe way of communication, is created and accessed by the visual representation of the simulation, its running program, and the listener. This queue is used to write all the messages of the two systems, but its values are only read by the listener. Therefore, the running simulation program and its visual representation both run independently and the actual communication happens in the listener class explained in the next paragraph.

4.9.2 Forwarding the messages in the simulation

The listener is the object that takes care of forwarding the messages between the visual representation of the simulation and its running program. It is run on its own thread, separated from the thread running the editor and the simulation; this is done because the listener does nothing but wait for new messages to be added to the queue, then routes these messages to their correct destination as events added to the controllers. This makes sure that both system are independent of each other otherwise errors will happen with them not complying with the event loop because they're waiting for a message to be received from the queue.
The listener class' only function is shown in listing 13.

Listing 13: Listener function written in Python.

```
while True:
    msg = runner.writerQueue.get()
    entry = str(msg).split(' ');
    if 'create' in entry[1]:
        visualController.addInput(
            Event("simulate_create", "input", [entry[0]]))
    elif 'delete' in entry[1]:
        visualController.addInput(
            Event("simulate_delete", "input", [entry[0]]))
    elif 'entered' in entry[1]:
        tmp = str(msg).split(' entered ')
        visualController.addInput(
            Event("simulate_entry", "input", [entry[0], tmp[1]]))
    elif 'exited' in entry[1]:
        visualController.addInput(
            Event("simulate_exit", "input", [entry[0]]))
    elif 'fire' in entry[1]:
        runningController.addInput(
            Event(entry[0], "system_input", [entry[2]]))
```

The function runs an infinite loop that processes every new message added to the queue. If the message is a 'create', 'delete', 'entered' or 'exited' message, then the listener sends it to the visual representation of the simulation as an event added to its controller. The attributes added to the event are the element routing name and, in case of an 'entered' event, the list of attributes of the object instance. If the message is a 'fire' message, then the listener sends it to the running program of the simulation as an event added to its controller. The name of the event is the same as the name contained in the message, which directly fires the correct transition in the statecharts. This event also contains the name of the instance that fired this event so that only the correct target reacts to it and not every transition of every instance that have this event.

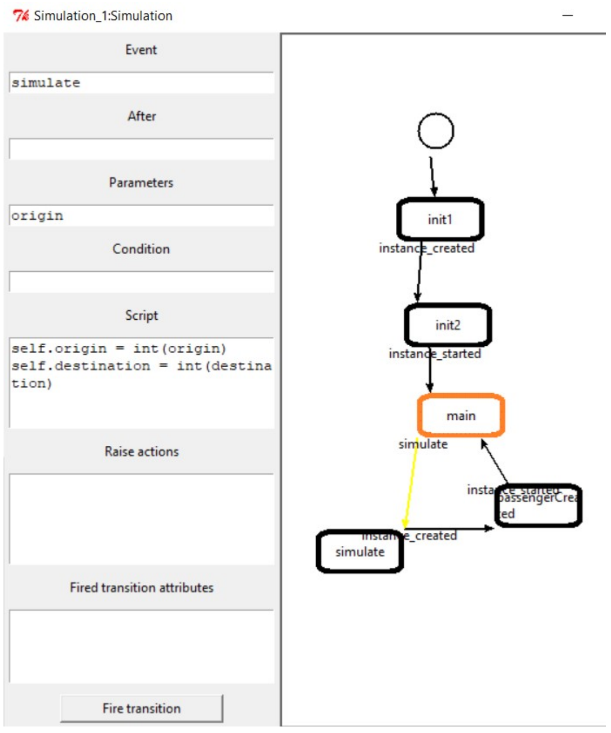
4.9.3 Applying the messages in the visual representation of the simulation

In order for the visual representation to interact with the messages forwarded by the listener, new transitions need to be added to the statecharts of the simulated system so that they could react to the events added to the controller:

- *Draw_class_simulation* has an additional transition with the *selection* composite states as both its origin and target. The event that triggers this transition is 'simulate_create', and this transition runs a *rebuild_class* function. This function is similar to the load function explained in 4.7.1 but it only creates and starts the class object that has the same routing name as the message sent to the queue of the listener.

- *Draw_class_simulation* has another additional transition with the *selection* composite states as both its origin and target. The event that triggers this transition is 'simulate_delete', which removes the visual representation of a class instance from the object diagram if this instance was deleted in the running program of the simulation. Only a deletion of the visual representation is needed since the simulator system is minimal and its different object instance elements don't interact with each

Figure 4.17: Transition that can be fired in the simulation of a system



Simulation_1:Simulation

Event

```
simulate
```

After

Parameters

```
origin
```

Condition

Script

```
self.origin = int(origin)
self.destination = int(destina
tion)
```

Raise actions

Fired transition attributes

other. The deletion of the actual object is done when the simulator is exited.

- *State_simulation* has two additional transitions with the *selection* composite states as both their origin and target. The transitions are triggered by the 'simulate_entry' and 'simulate_exit' events, and they both change the color of the visual representation of this element accordingly. The *state_simulation* event also updates the attributes of the object as received in the event.

Sometimes, some messages cannot have a direct effect on the UI if the statecharts of the object has not been opened by the user, therefore the following possibilities arise:

- The messages can be completely thrown away and no effect is visible in the UI. With this approach, when any statecharts window is opened, the developer will not see any active states which is incorrect because in statecharts, the system always has to be inside of at least one state if there were no orthogonal regions.

- The messages can be buffered inside of an array and applied one after the other once the statecharts windows is opened. This means that the developer will see a series of events firing and states activating and deactivating but not in real time. This approach is also not correct according to the statecharts formalisms.

- The optimal solution, and the one that is implemented, is to only save the last relevant message to the system, that is the last *entered* message, and only apply that message to the statecharts. This results in the system having an active state even if the window was opened after the last message has been sent which complies with the rules of statecharts.

The active states in the statecharts are colored orange, while the inactive states are black.

4.9.4 Firing events from the visual representation of the simulation

When a state of a class object becomes active, all of the transitions that leave this state have the possibility of triggering, with some that trigger automatically if the transition is set with a timer. And others that do not fire on their own and need, either an interaction with the developer to happen, or an event to be raised by the system, for them to have an effect.
When the system makes a state active in the statecharts simulation, it checks all of the transitions with this active state as their origin, and, if the transition is fired by an event and not a timer, its color is set to yellow to show the developer that they can interact with it. If the developer clicks the mouse left button on a yellow transition, the information held by this transition is shown on the left section of the simulator with an added $'firing\ transition\ attributes'$ entry box and $'fire\ transition'$ button as shown in 4.17. When this button is clicked, a message is added to the queue so that it can be forwarded by the listener to the running simulation program. The message added to the queue contains the name of the event fired, which will immediately fire the correct event in the statecharts of the running program, the instance name of the object it belongs to so that only the correct instance of the object reacts to this event, and the attributes added to the $'firing\ tranisiton\ attributes'$ entry box which allows the developer to specify additional attributes to the event if needed.

5 Railcar system using the SCCD editor and simulator

In the previous chapters, we used the railcar system as defined by Harel [9] to compare Yakindu to SCCD. And, using this comparison, we developed the SCCD visual editor and simulator in order to have a program that can fully support all the requirements of such complex systems.
In this chapter, we will revisit the railcar system and recreate it using the newly developed SCCD editor and simulator, and see how well it holds against its unmet requirements.

5.1 Development of the system

Instead of having to use Yakindu's GUI to easily create the SCXML version of the system, and then manually modify it to add the SCCXML functionalities, we can simplify the development by directly using the SCCD GUI editor. With this editor, we can directly model the classes, their statecharts and the associations connecting them. And, with its exporter, we can directly generate the SCCXML code of the system and deploy it without any additional modifications needed.
The system was developed similarly to how it was explained in 3 resulting in the same functionalities. Fig. 5.1 and 5.2 show the class diagram of the railcar system and the statecharts of one its classes, the *control center*, respectively.

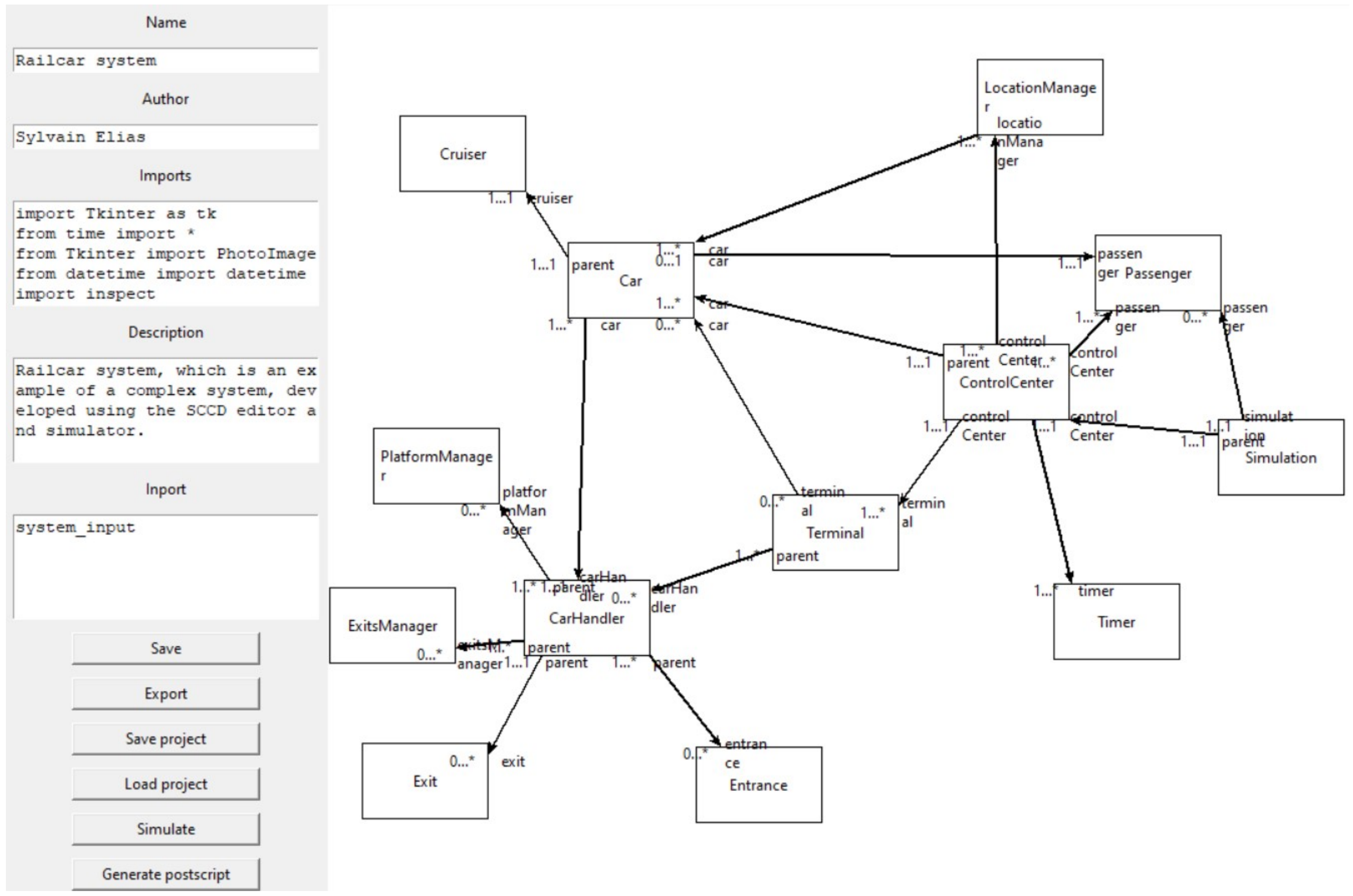
5.2 Simulation of the system

Using the simulator added to the SCCD GUI, we can directly run a real-time simulation of the system which was not a feature of SCCD previously. This simulation shows the instances of the class objects being created and deleted, and the active states of their statecharts. It also allows the developer to directly interact with the system by firing events using the GUI and watching its effects on the active states and the attributes saved by the object. Fig. 5.3 and 5.4 show the multiple instances of the objects created by the simulation, and the active state of one of the *terminal* instances with events that can be fired by the developer respectively.

5.3 Requirements satisfaction

Since the SCCD editor and simulator does not change in the formalism of SCCD but only adds functionalities on top of it, then the requirements that were previously met and explained in 3.19.2 are still valid.
On top of that, the addition of the visual editor makes the development of the system easier and more straightforward than before.
And with the addition of the simulator, SCCD is able to meet the last requirement of the complex systems, which is to be able to simulate the system while developing. On top of that, the simulator allows to have a visual representation of the multi state machine modeling by allowing the developer to see the statecharts of each object functioning independently, and how their events affect each other. It also adds a visual representation for the dynamic modeling of the system by creating and deleting the visual representations

Figure 5.1: Class diagram of the railcar system developed using the SCCD editor and simulator



Name

```
Railcar system
```

Author

```
Sylvain Elias
```

Imports

```
import Tkinter as tk
from time import *
from Tkinter import PhotoImage
from datetime import datetime
import inspect
```

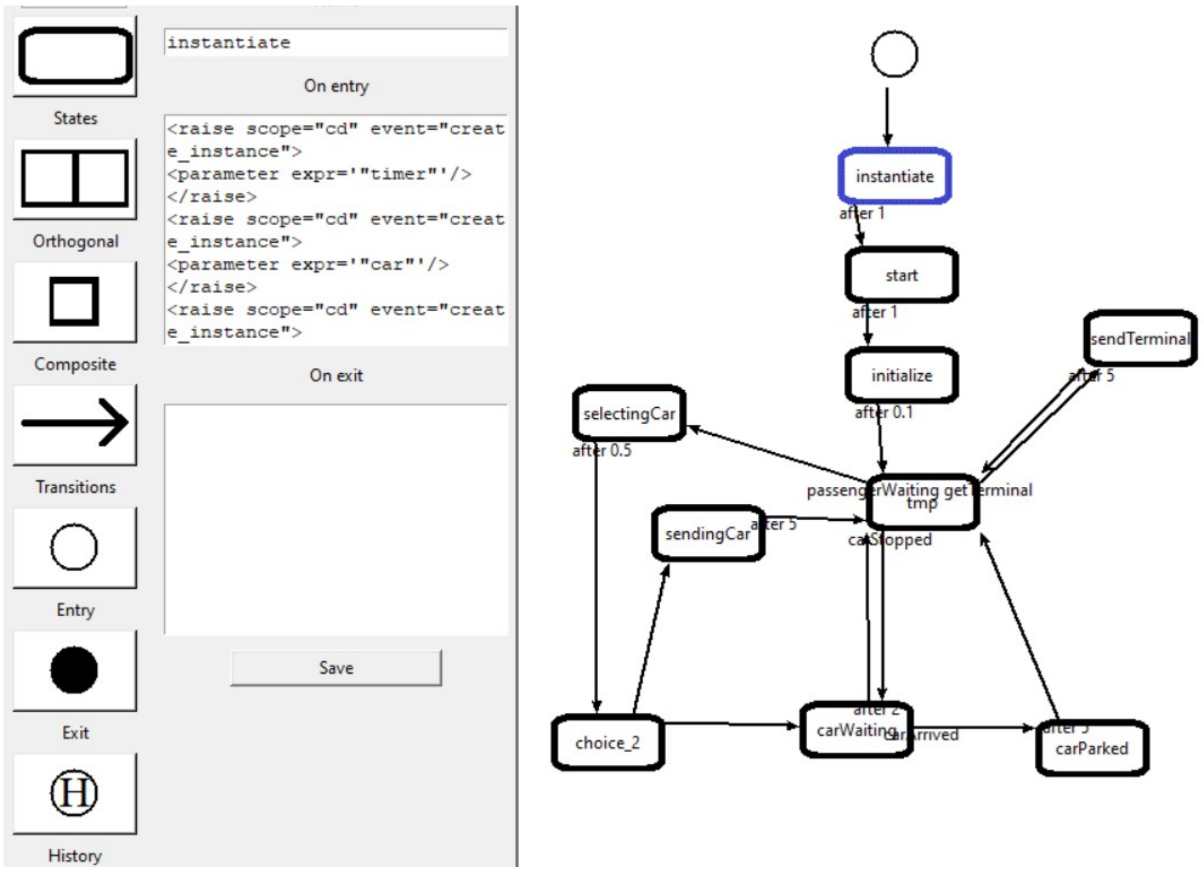
Description

```
Railcar system, which is an ex
ample of a complex system, dev
eloped using the SCCD editor a
nd simulator.
```

Inport

```
system_input
```

Figure 5.2: Statecharts of the control center class developed using the SCCD editor and simulator



```
instantiate
```

```
<raise scope="cd" event="creat
e_instance">
<parameter expr='"timer"'/>
</raise>
<raise scope="cd" event="creat
e_instance">
<parameter expr='"car"'/>
</raise>
<raise scope="cd" event="creat
e_instance">
```

Figure 5.3: Object instances of the simulation using the SCCD editor and simulator

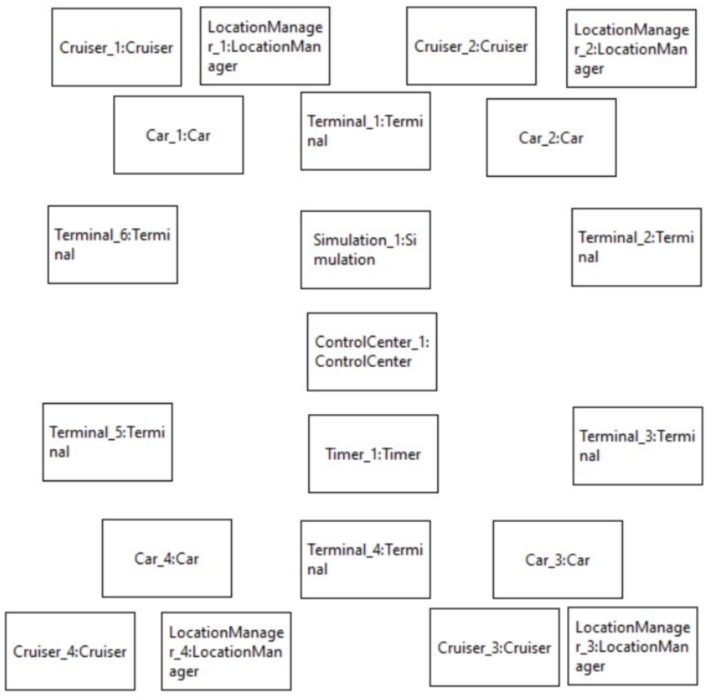
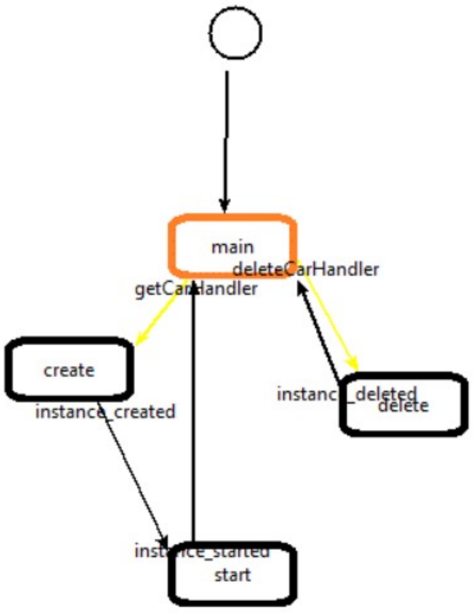


Figure 5.4: Statecharts of one of the running terminal instances using the SCCD editor and simulator



of the instances of the objects in the object diagram as it is happening in the running program in the background.

6 Testing the SCCD editor and simulator

The railcar system is used to test the newly developed editor and simulator. This test doesn't check all the aspects of the system extensively, but it is good enough to check that the system works as it should until more and better tests are added.
Since the railcar system developed by the SCCD visual editor and simulator is the same as the one developed using the textual GUI, it is easy to connect its generated code to the railcar UI and check if it works. And, since the railcar system is developed, saved, loaded and exported using the SCCD editor, then verifying the generated code can help in testing all of these different aspects of the system.
Therefore, the python code is generated using the SCCD editor's exporter functionality and SCCD's code generator tool, and it is connected to the railcar system UI previously developed, and the same trace file is generated as explained in 3.19.1. Comparing the traces of the system developed using both SCCD versions shows us that the two systems are indeed similar which means that the SCCD GUI editor generates the correct code for the developed systems compared to its textual editor.
Testing the simulator is harder than testing the editor since there's no other simulated version available that we can compare to, and the railcar system created using Yakindu isn't fully the same as the one developed using SCCD because of the multiple workarounds that needed to be added.

7 Conclusion

After working on the implementation of the railcar system using both tools and comparing the process between them, we notice that, even though Yakindu is a commercial document with an easy visual editor, it doesn't present a straight-up solution to solve the requirements of complex multi-state machines.
SCCD, on the other hand, can model and generate code for the complex system without any major workaround needed, but it does not offer the possibility to directly simulate the system neither is it as simple to design systems using it as with having a visual user interface.
After having created a visual editor and simulator for SCCD and developing the same railcar example using this new system, it becomes apparent that, with its newly added functionalities, SCCD is better at handling the development of complex system than Yakindu. It now allows to create a multi-class system using an easy visual approach instead of textual, and it also directly simulates the system without having to deploy it first.
The creation of the visual editor shows how more intuitive it is to develop GUI using statecharts by benefiting mainly from their event based approach, which integrates perfectly with the nature of UI and the user's interactions with them. These systems with their hierarchical separation and event bubbling are perfect to create graphical user interfaces. And with the use of SCCD and its combination of classes and statecharts, it becomes even easier to simplify the creation of more complex user interfaces by combining both statecharts and object oriented approach.
Moreover, SCCD has a big advantage in creating a simulation environment and linking it to its running program by taking advantage of its statecharts implementation, which allows to easily append the state and data changes of the running program to its visual UI by adding this information as events to the controller of the UI system, and forwarding the user generated events from the UI to its running program using the same approach. Additionally, all of this communication is done with both systems running on separate threads which opens up many possibilities to expand on the system. Moreover, SCCD's approach to separating the systems into multiple classes helped in the creation of one common program that can do both, development of the system and its simulation.

7.1 Future work

This thesis discussed the feasibility of creating a statecharts and diagram editor and simulator using SCCD and explained how such program can be designed and developed. But some more features should be added to expand on its capabilities and to make its user interface better.

7.1.1 Class diagram and statecharts editor

The statecharts of the classes and the statecharts' elements can have some additional design upgrades, like allowing the user to select multiple elements at the same time or even adding the possibility to move an element between the statecharts of the different classes. But with SCCD's formalisms, all of these features can be easily added by simply adding transitions and states to the SCCXML design of the system.

7.1.2 Object diagram and statecharts simulator

The object diagram and statecharts visual simulation, in its current state, runs in the same system program as its running simulation. This works the way it is, but the system would be more stable if we separate the two into separate running programs. Therefore, instead of initializing the controller of the simulation under the same system program as the user interface, the running simulation should be opened as its separate program running in the background. This means that the implementation of the queues would be obsolete, but they can easily be replaced by the use of HTTP sockets with the same listener as the middleman, or even better, completely removing the listener and adding states to the system's statecharts which, when active, take the messages from the HTTP sockets and apply them to the system.

7.1.3 Test framework

Yakindu has a testing framework built on their SCTUnit language [14] which allows to test the statecharts created using it. A part of the future work is to add a similar testing framework to the SCCD editor and simulator. Additionally, since the SCCD editor and simulator was created using SCCD itself, then this system should be bootstrapped using itself, which means recreating the SCCD editor and simulator system using the SCCD editor and simulator system. And, with the addition of the testing framework, this would allow to extensively test all the aspects of this system to be sure that no bugs or unexpected behaviour occurs.

7.2 Related work

In [5], Dubé studied graph layouts and explained the pseudo-code behind the different graphs layering and positioning algorithms. His work inspired the design used in the canvases and their elements in the visual editors.
In [16], an SCXML editor and runtime was developed for prototyping and production of user interfaces mainly for automotive in-vehicle interfaces. And, in [3], Beard explained the development of SCION using statecharts, and specifically SCCXML, which is an implementation of statecharts that is designed to be used in the Web browser for user interface development.

References

[1] Caroline Appert and Michel Beaudouin-Lafon. Swingstates: Adding state machines to java and the swing toolkit. *Software: Practice and Experience*, 38:1149 – 1182, 09 2008.

[2] Jim Barnett, Michael Bodell, Dan Burnett, Jerry Carter, and Rafah Hosn. State chart XML (SCXML): State machine notation for control abstraction. W3C Working Draft, February 2007.

[3] Jacob Beard. Developing rich, web-based user interfaces with the statecharts interpretation and optimization engine. Master's thesis, School of Computer Science, McGill University, Montreal, Quebec, Canada, 2013.

[4] Jacob Beard and Hans Vangheluwe. Modelling the reactive behaviour of SVG-based scoped user interfaces with hierarchically-linked statecharts. *SVG Open 2009 - 7th International Conference on Scalable Vector Graphics*, 01 2009.

[5] Denis Dubé. Graph layout for domain-specic modeling. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, 2006.

[6] Shahram Esmaeilsabzali, Nancy Day, Jo Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requir. Eng.*, 15:235–265, 06 2010.

[7] Joeri Exelmans. Configurable semantics in the SCCD statechart compiler. Technical report, University of Antwerp, 2014.

[8] John E. Grayson. *Python and Tkinter Programming.* Manning Publications Co., USA, 2000.

[9] D. Harel and E. Gery. Executable object modeling with statecharts. *Computer*, 30(7):31–42, 1997.

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

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

[12] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5:293–333, 10 1996.

[13] AG Itemis. Modeling and execution semantics. https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/modeling_execution. Accessed: 2021-09-28.

[14] AG Itemis. The SCTUnit language. https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/sctunit_the_sctunit_language. Accessed: 2021-09-28.

[15] AG Itemis. What are YAKINDU statechart tools? https://www.itemis.com/en/yakindu/state-machine/documentation/user-guide/overview_what_are_yakindu_statechart_tools. Accessed: 2021-09-28.

[16] Gavin Kistner and Chris Nuernberger. Developing user interfaces using SCXMl statecharts. 2014.

[17] Mathias Kühn Peter Forbrig, Anke Dittmar. Extending SCXML by a feature for creating dynamic state instances. *2nd Workshop on Engineering Interactive Systems with SCXML*, 2015.

[18] Statecharts.dev proj. Statecharts in user interfaces. https://statecharts.dev/use-case-statecharts-in-user-interfaces.html. Accessed: 2021-09-28.

[19] Miro Samek. *Practical Statecharts in C/C++: Quantum Programming for Embedded Systems*. CMP Publications, Inc., USA, 2002.

[20] Bran Selic. Using UMl for modeling complex real-time systems. volume 1474, pages 250–260, 01 1998.

[21] Simon Van Mierlo, Yentl Van Tendeloo, Bart Meyers, Joeri Exelmans, and Hans Vangheluwe. SCCD: SCXML extended with class diagrams. pages 1 – 6, June 2016.

[22] Simon Van Mierlo and Hans Vangheluwe. *Statecharts: A Formalism to Model, Simulate and Synthesize Reactive and Autonomous Timed Systems*, chapter 6, pages 155–176. Springer International Publishing, 2020.