technische universität
dortmund

Bachelor Thesis

A Graph Language for Sequentially Constructive Statecharts

Kristopher Tom Kettler
October 2023

Supervisors:
Prof. Dr. Bernhard Steffen
Dr. Steven Smyth

TU Dortmund University
Faculty of Computer Science
Chair of Programming Systems (LS V)
http://ls5-www.cs.tu-dortmund.de

Contents

Abstract

In the field of model-driven software development, domain-specific languages can make an important contribution to the development of software systems. However, the development of modeling tools for this languages has often proven to be complex and tedious. The Cinco Meta Tooling Suite is a tool to create such modeling tools, emphasizing simplicity. Its main feature is the full generation of graph-based modeling tools. Using SCCharts, a visual modeling language for the design of safety-critical reactive systems, as an example, the development process is demonstrated to show how the Cinco Meta Tooling Suite can ease development of domain-specific tools.

Chapter 1

Introduction

The increasing complexity of software systems poses new challenges for software developers. As a result, the use of modeling languages to develop such systems has grown in popularity, helping developers to gain a better understanding of the system. In addition, code generators are often part of such tools, which generate executable code from the designed model that can be integrated into applications. In general, they help to increase the efficiency of the development process. Starting with textual languages, such as VHDL for describing hardware or HTML markup language for developing web pages, to graphical modeling languages such as the well known UML, the model-based approach runs through every area of software development. Basically, a distinction is made between general-purpose modeling languages (GPMLs) and domain-specific languages (DSLs). The latter are mostly textual and require Language Workbenches, frameworks that support the development of according editors and tools for DSLs. Thereby, it provides similar features as modern Integrated Development Environments (IDEs). In addition, metamodeling frameworks, such as the Eclipse Modeling Framework (EMF) or JetBrains Meta Programming System support the development of editors for DSLs. [6, 8]

However, the development of domain-specific tools has often proved to be complex and laboriously. Also, most metamodeling frameworks focus on textual modeling languages, including those mentioned above, so support for graphical languages is rather sparse. For these reasons, the Cinco Meta Tooling Suite (Cinco) was developed. It offers a solution for the creation of graph-based modeling tools. Furthermore, it follows a simplicity-oriented approach, which means that universality is limited in favour of simplicity. The core feature of Cinco is the full generation of these domain-specific graphical modeling tools from meta-level specifications and models. This is a advantage over other approaches that only support semi-automatic generation of graphical editors from high-level specifications, and where the generated code still needs to be adapted before the editor can be executed. [6] [8]

1.1 Motivation

Although the potential of DSLs to enable domain experts to create software systems on their own without the help of programmers is not always achieved, it is possible for domain experts to understand and and verify code written by developers. Nevertheless, DSLs seem to be less popular than GPMLs, of which there are many more variations. Naujokat et al. [6] suggest that this is because people consider the complex development of DSL tools to be too complicated and elaborate, and therefore not worthwhile. This observation is the motivation for this bachelor thesis. With Cinco a solution has now been created that facilitates the creation of editors for graphical domain-specific languages by trading generality for simplicity. If it could be shown how easy it is to create a DSL tool using Cinco, this might convince more people to create their own DSL tools.

1.2 Problem Statement

The central problem addressed by this bachelor thesis is to demonstrate how Cinco, a tool that prioritizes simplicity in creating graphical DSL editors, can facilitate the development of such DSL tools. The SCCharts language [4], a visual modeling language designed for specifying safety-critical reactive systems, was chosen as an example of a graphical DSL for which the editor development process is shown. The editor developed for this purpose with Cinco must provide some functionalities to support the development of SCCharts in order to be considered a useful tool. On the one hand, it should be possible to create and extend SCChart models. Potential users must be able to read the created models correctly. The visual syntax of the model created in the editor should therefore be as similar as possible to that of SCCharts. In addition, it should not be possible to create incorrect models or it should be indicated to the user if there are errors in the model. The user interface should be as user-friendly as possible so that users who are not programmers can also use the tool. Finally, the editor should be able to generate code from the created model that can be integrated into software systems.

1.3 Structure

Chapter 2 presents the foundations of model-driven software development (Section 2.1), which are essential for understanding the editor's development process, and introduces the SCCharts language (Section 2.2), necessary for implementing the editor based on it. In Chapter 3, related work is addressed. Subsequently, Chapter 4 deals with the tools and frameworks that were ultimately used to create the editor for SCCharts. The core of this bachelor thesis, the conception and implementation of the graphical DSL tool for SCCharts, is presented in Chapter 5. First, requirements for the SCCharts tool are defined

(Section 5.1). This is followed by the design of the data structure (Section 5.2), which serves as the basis for the graph language. This is then implemented with the style that defines the visual syntax of SCCharts components in the editor (Section 5.2). Next, the user interface and the model validation are implemented (Section 5.4). Finally, the code generator is realised (Section. 5.4). After the editor has been designed and implemented, it is evaluated and a conclusion is drawn in Chapter 6. In the final chapter 7 a summary is presented and an outlook on possible extensions of the editor is given.

Chapter 2

Foundations

This chapter provides conceptual foundations that are essential for understanding the development process of the editor. First, the basics of model-driven software development (MDSD) important in this context are presented, followed by an exploration of the SCCharts language.

2.1 Model-Driven Software Development

Before clarifying what exactly MDSD is, the term model must be defined more precisely.

2.1.1 Model

Generally a model can be said to serve as a simplified or partial representation of real entities, designed to emphasise essential information or features while omitting less important details. Models are used in many fields, from science to business, and serve as valuable tools to understand, analyse and communicate complicated phenomena or processes. In the realm of software engineering, models assume diverse roles, spanning from describing systems or specifying requirements to the generation of configuration files or executable code based on the information encapsulated within these models. [2]

2.1.2 Goals of Model-Driven Software Development

According to Brambilla et al. [2], MDSD "is a development paradigm that uses models as the primary artifact of the development process. Usually, in [MDSD] the implementation is (semi)automatically generated from the models." The aim is to use models as development artefacts and to use abstraction to hide the unimportant details and highlight the important ones. In addition, it tries to automate as much of the development process as possible. This can start with the definition of requirements and reach up to the final developed software system.

2.1.3 Modeling Language

Modeling languages are an important part in the MDSD. They serve as a mechanism through which designers can articulate both the structure and behavior of their systems, utilizing either graphical or textual representations. Designers are required to adhere to the specific syntax defined by the modeling language they are using. There are two primary classes of modeling languages: GPMLs and DSLs. DSLs are modeling languages designed for a specific domain or area of application. This specialization ease the process of creating models within that particular domain. Conversely, GPMLs can be applied to a wide range of applications but lack the tailored focus found in DSLs. [2]

2.1.4 Metamodeling

The term metamodeling describes a concept for creating models from models. The model that is used to create another model is called a metamodel. Thus, a metamodel is essentially a model that describes the structure and semantic rules for a particular class of models. It is used to define the syntax and semantics of a modeling language and to ensure that models created in that language conform to the defined rules.

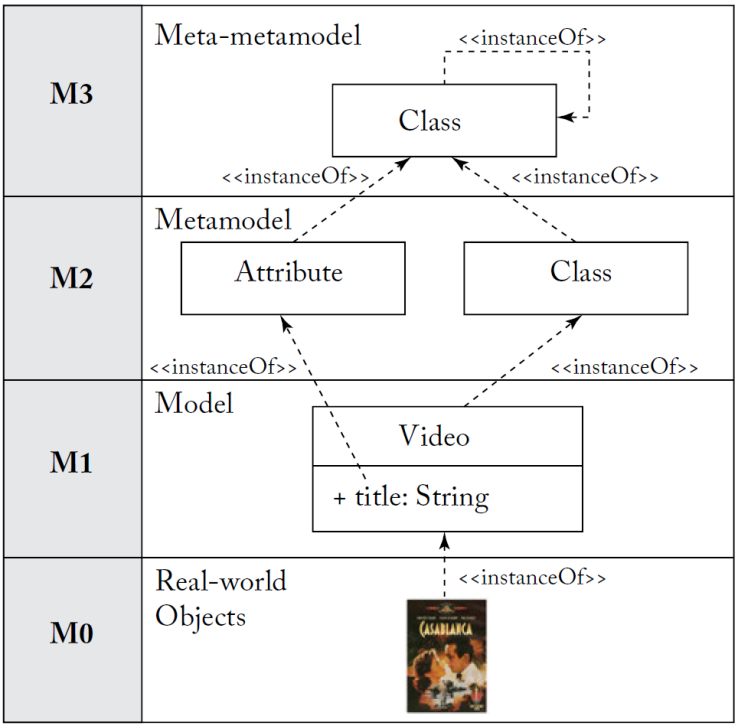

Figure 2.1: Models, metamodels, and meta-metamodels (from [2])

A suitable example for this can be seen in Figure 2.1. When describing real objects through models, the model level M1 defines the syntax and semantics of the level below it, M0. The same applies to the model level in relation to the metamodel level M2. For

the metamodel level M2, once again, the syntax and semantics can be defined in the level above it, the meta-metamodel level. This process could continue indefinitely, so that starting from the meta-metamodel level, each subsequent meta-metamodel instance refers to another meta-metamodel instance. [2]

This example can also be mapped to the creation of a DSL tool for SCCharts. In M3 is the Cinco meta tool with which a DSL tool for SCCharts is to be created (layer M2). This can then be used to create SCCharts models (M1) that represent a SCChart of the SCCharts language (M0).

2.2 Sequentially Constructive Statecharts

The visual modeling language SCCharts [4, 7], which is based on Harel's Statecharts [5], was designed for safety-critical applications and aim for easy adaptation. It utilize the visual syntax from Charles André's SyncCharts [1] and provides deterministic concurrency based on a synchronous Model of Computation.

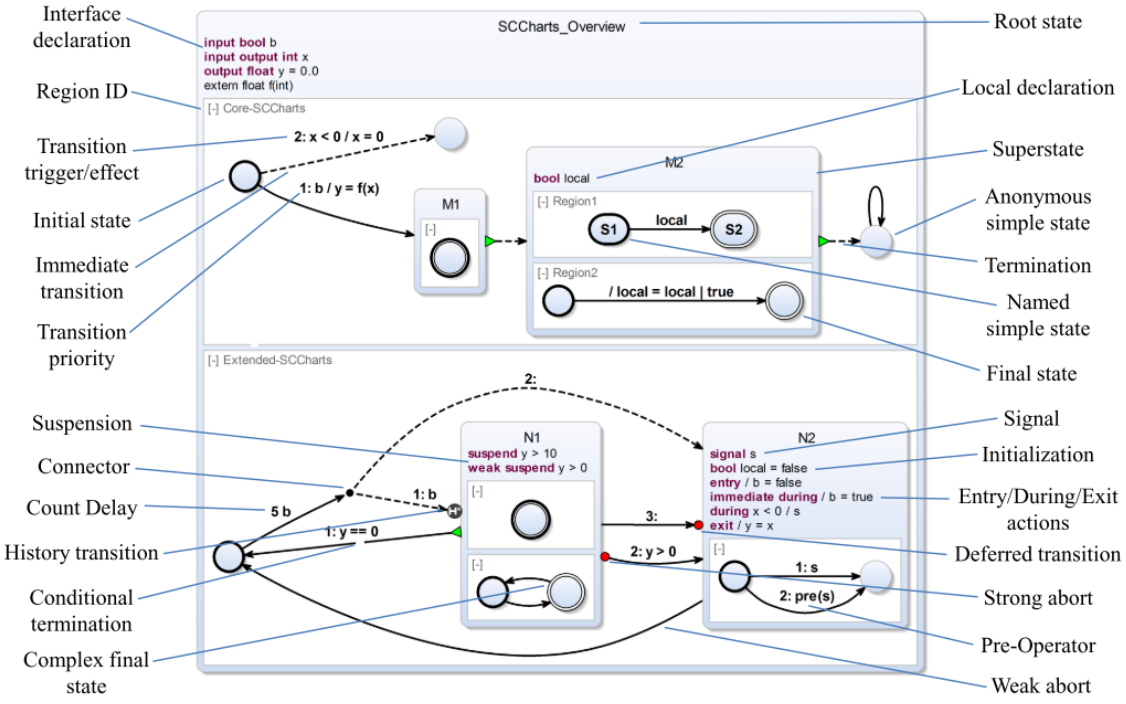

Figure 2.2: SCCharts-Overview (from [4])

To gain a better understanding of the different elements of the SCCharts syntax, Figure 2.2 is helpful. The upper part shows the Core-SCCharts, which include concurrency and hierarchy as essential features of Statecharts. While the lower part contains elements of the Extended SCCharts. It should be noted that each advanced feature can be expressed as one or more core features, but the use of advanced features reduces the complexity of

the diagram and therefore improves clarity. The components are explained in more detail below.

2.2.1 States and Superstates

States are a fundamental part of SCCharts. A distinction is made between simple states and states with internal behaviour, so-called superstates. Both types of states can function as initial states (thick border), final states (double border) or initial final states (thick border and double border). Each SCChart also has exactly one root state that behaves like a superstate and in which all behaviour of the SCChart is defined. Additionally, there are connectors (large black dot), which are essentially simple states with the purpose of connecting multiple states through outgoing and incoming edges.

2.2.2 Transitions

Another basic component is transitions. Each transition has a source state and a target state. Transitions can have a label with the syntax *[p:] [t] [/ a]*, where p stands for priority, t for trigger and a for action. The trigger is a side-effect-free boolean expression and the action is an assignment of any data type. A state is said to be active when the SCChart is in that particular state. If the source state of a transition is active and the trigger becomes true, the target state is active. The transition can be immediate (dashed transition) or delayed. Immediate means that the state becomes active on the same tick (stimulus), while delayed means that the state becomes active on the next tick. Transitions are delayed by default to avoid causality problems. The priority is used to check the trigger in ascending order, so that if there are, for example, two transitions with true triggers, the one with the higher priority is used.

2.2.3 Declarations

Declarations of variables can be made at the top of any superstate, including root states. These variables can be inputs, which are read from the environment, or outputs, which are written to the environment. Additionally, they can also be input output variables or local variables, which are used only internally. The data types for these variables can include boolean, integer, float, and string. Local variables can also be uninitialized or pre-assigned, and their values persist even when, for example, the superstate where they are defined is exited. Furthermore, there are signals used for communication with the environment and among the components of the SCChart. Pure signals are interpreted as boolean values and are true when present and false when absent. In contrast, valued signals carry a typed value in addition to their presence status. Lastly, declarations can be marked as constant, which means that the value cannot be changed after initialization.

2.2.4 Hierarchy and Concurrency

Superstates, including the root state, have inner behavior, which includes one or several concurrent regions (represented by white boxes). Each region conceptually corresponds to a thread, and each region must contain an initial state. If a final state is entered within a region, that region terminates. Termination transitions (green triangle at the source state) can originate from superstates and are taken when all regions within the superstate are in final states. These transitions are typically unconditional, and superstates usually have only one of them. If there are multiple termination transitions, the one with the highest priority is taken.

2.2.5 Actions and Suspensions

Actions and suspensions are specified beneath the declarations of superstates. There are four distinct types of actions that influence the behavior of the associated superstate. These actions have an effect and may include an optional condition. Entry and exit actions are executed when entering or exiting the superstate, either when they have no condition or when their optional condition evaluates to true. Additionally, there are during and immediate during actions, which are executed if the superstate they are associated with is active and they either have no condition or a condition that evaluates to true. Depending on their type, they may be executed with a delay or immediately.

Suspensions suppress the inner behavior of a superstate, if their trigger is evaluated to true. Their also four different types. Immediate and delayed suspensions are self-explanatory. Weak suspensions and immediate weak suspensions allows immediate inner behavior of a suspensions state but the weakly suspended state remains in the exact same internal states.

2.2.6 Complex Transitions

Beyond the simple transitions described initially, there are additional transitions that have effects upon entering or exiting superstates. One of these is history transition (H at the target state), which allows the execution to continue in the state that was active when the superstate was left. Shallow history applies only to the top level of the superstate, while deep history (H with *) reaches into the deeper levels as well. Furthermore, there are deferred transitions (red dot at the target state), which preempt all immediate behavior in the target state. Finally, there are strong abort transitions (red dot at the source state). Unlike weak aborts (default transitions), which allow the inner behavior of superstates to continue during the tick, strong aborts terminate them directly. Additionally, transitions can have a count delay. This means that the condition must be true for a specific number of ticks before the transition is triggered.

2.2.7 References

Another feature of SCCharts is their ability to reference each other. To achieve this, the corresponding inputs and outputs of the referenced SCChart must be assigned variables of the appropriate type. These references then become superstates within the inner behavior of the referenced SCChart. Figure 2.3 illustrates an example[1] of such a reference. Here, Beep is referenced as a superstate within the region of Root. In this case, the input boolean second from Root is assigned to the input second of Beep, and the output boolean speaker from Root is assigned to the output boolean beep of Beep.

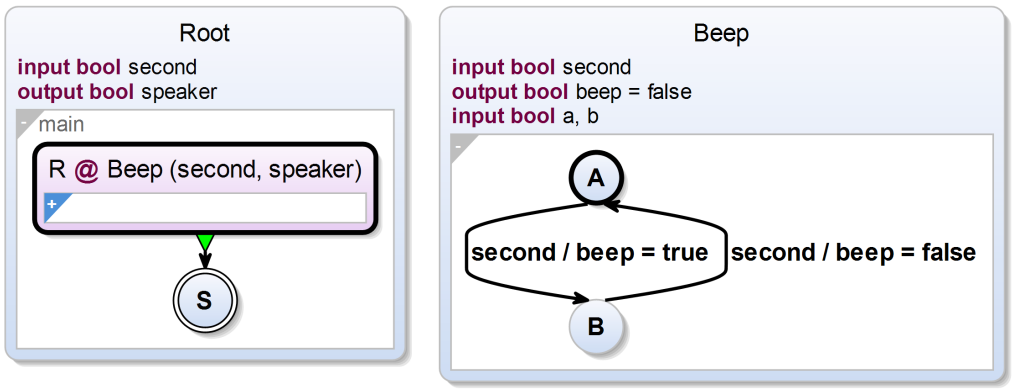

Figure 2.3: Beep Example for references in SCCharts

[1] https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Syntax

Chapter 3

Related Work

Firstly, reference will be made to similar projects where instances were created using Cinco. This is followed by scholarly works that provide more information on SCCharts.

3.1 Modeling with Cinco

For a better understanding of the background of Cinco and the reasons for its development, Naujokat et al. [8] provides a comprehensive overview. Here the new design decisions are explained and functions are shown by the creation of a Petri Net editor. In addition, it also suggests application areas and compares the tool with other meta-modeling tools.

Lybecait et al. [6] presented a Cinco Project that can be used to create web stories, an adventure game. The primary goal was to showcase the two recent additions, Pyro and GCS (Graphical Cinco Specification): Pyro extends the Cinco ecosystem by enabling web-based modeling for Cinco-based graphical modeling languages, while GCS introduces graphical editors to the meta-level in a bootstrapping fashion. This is aimed at reducing the complexity of creating DSLs via Cinco and improving the overall process. The Project can be found on the Cinco project website[1] under examples.

The second project is a graphical language for statecharts, similar to the one designed in this bachelor thesis. However, instead of using a code generator, an additional simulator for the input was implemented. A screenshot of it can be seen in Figure 3.1. While there is no written thesis available, this project is well-suited for exploring the features of Cinco. It can also be found at under examples on the Cinco project website.

[1] https://cinco.scce.info/

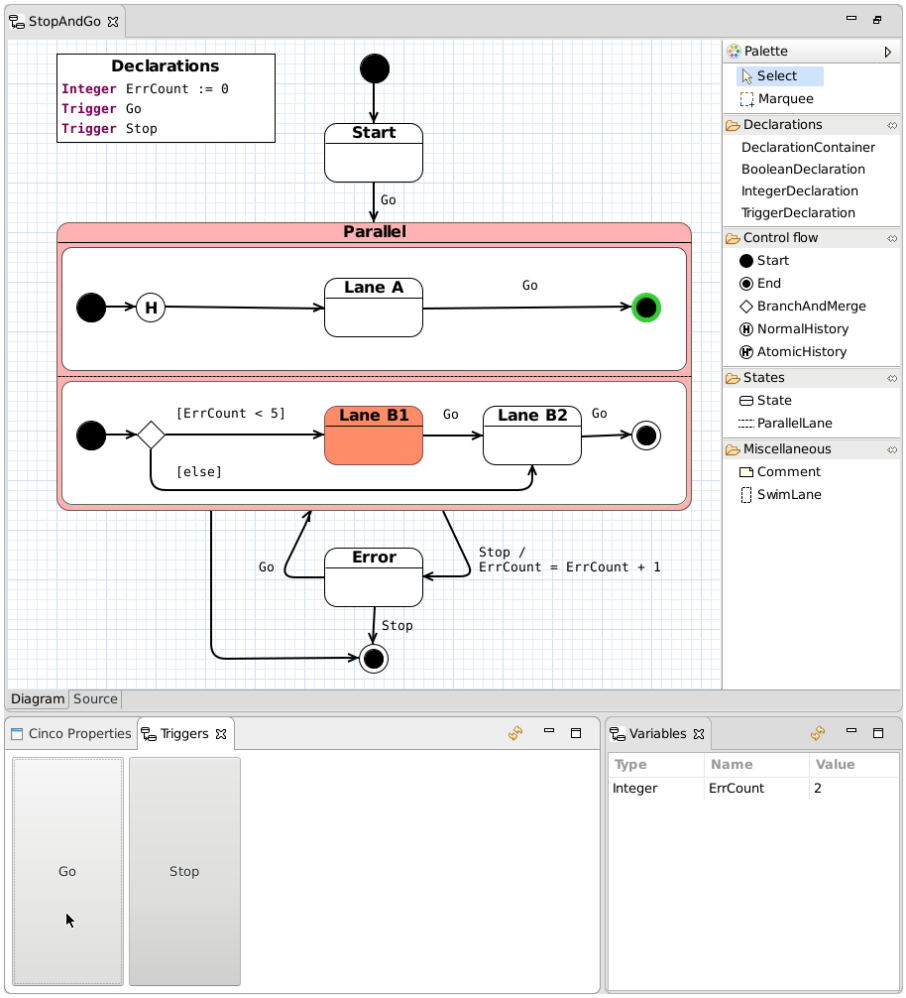

Figure 3.1: StateChart model and simulation screenshot of the CINCO Statechart project

3.2 Sequentially Constructive Statecharts

A detailed explanation of SCCharts as well as its background is provided by Motika's dissertation [7]. Synchronous languages are discussed in more detail. These are subject to the synchronous hypothesis, according to which time is divided into concrete intervals, called ticks. And the reactive system does not need time for computation, outputs and inputs are instantly available after each tick. Synccharts, on which the SCCharts language is based, are such a synchronous language with a graphical statechart dialect. Moreover, any valid SyncChart is also a valid SCChart.

Chapter 4

Frameworks and Tools

In this chapter, the tools and frameworks used within the context of this bachelor thesis will be introduced. First, Eclipse and Eclipse frameworks related to Cinco are listed, followed by a closer look at Cinco. Finally, a tool for creating SCCharts is introduced.

4.1 Eclipse and Frameworks

Eclipse[1] is an open-source software project released by IBM since 2001, primarily known as an IDE. However, Eclipse also offers a variety of frameworks and plug-ins that can be easily integrated, thanks to Equinox, an implementation of the OSGi R4 specification. Through the easy integration of these already existing or self-created plug-ins and frameworks, developers can create applications without having to implement complex functionalities themselves.

One of this frameworks is the Eclipse Modeling Framework (EMF). It is the central technology within Eclipse for MDSD, as it allows for the design of metamodels using the metamodeling language Ecore. The components of these metamodels can be generated as Java-based code and subsequently customized. Additionally, models can be designed and adapted in a tree-like editor. Furthermore, EMF provides a API. As a result, many other projects in the context of MDSD build upon EMF and offer additional functionalities, such as Cinco. [2]

Another framework is Xtext[2] which can be used for the development of programming languages and DSLs, by letting programmers define their own languages via a grammar language. It also offers a well-developed infrastructure which includes linker, parser, type-checker, etc., as well as eclipse integration.

Xtend[3] is an expressive and flexible java dialect. It compiles in java and thus enables any integration of java libraries. It also offers the function of extending existing types

[1] https://www.eclipse.org/
[2] https://eclipse.dev/Xtext
[3] https://eclipse.dev/Xtext/xtend/

without modifying them, hence the name. It also has the same runtime as equivalent Java code and offers other features such as lambda expressions or template expressions.

4.2 Cinco SCCE Meta Tooling Framework

Cinco [8] is a DSL tool to create domain specific graphical modeling tools. It leverages the extensive metamodeling and infrastructure features provided by the EMF, and the Graphiti diagram editor framework, a modeling infrastructure centered around the EMF, in case for representation and editing.

Every graphical tool created with Cinco, also called Cinco Product, has at its core the Meta Graph Language (MGL) model, from which the Ecore metamodel is ultimately generated. The MGL also defines which components exists in the model and can be created in the editor at the end. The components are all of the type *node*, *container* or *edge*. Containers are special nodes, which in turn can contain nodes or containers. Both MGL model and the three component types can have attributes, that are structurally similar to declarations in conventional programming languages, with names and data types. In addition, each component of the type node, container and edge has a style. This is defined in the Meta Style Language (MSL). There are *nodestyle* for nodes and containers, and *edgestyle* for edges. A node style is composed of *ContainerShape* and *Shape*. Container shapes can be *roundedRectangle*, *ellipse*, and other geometric shapes. These can contain further container shapes or shapes, such as *text*, *line*, etc. In the edge style, decorations can be defined that are displayed on the connecting line. These can be predefined arrows or circles, but also self-defined decorators. In addition, the *appearance* of each style object, i.e. container shape, shape, edge style or decorator, can be adjusted. This includes, for example, the colour or the line type (solid, dashed, etc.). With the *appearance provider*, the appearance of node styles and edge styles can be changed at runtime. However, with the exception of the appearance, the style can no longer be changed at runtime. This design decision is due to the simplicity approach.

Besides the MGL and MSL, there are also plug-ins, so-called *meta plug-ins*. These extend the functionality of the editor. They are activated by annotation in the MGL but also in the MSL. An example of such a meta plug-in is the Event API, which enables the editor to respond to various events that occur within it. These events can encompass actions such as node creation, deletion, or resizing. Additionally, events can be triggered when the model is saved or when an attribute is modified. The Event API essentially allows the editor to be more interactive and responsive to user actions and changes in the model. The developer has to imply what should happen after an event. There is also the Prime References meta plug-in, which allows referencing to external components of other models, or the *code generator*, which can be used to generate code from the models. These are just a few of the meta plug-ins from Cinco.

The user interface is shown in Figure 3.1. The model created can be seen in the middle. On the right side are the model components nodes and containers, which can be inserted into the model by dragging and dropping. By hovering over nodes or containers, edges can be dragged and dropped into the target node. In addition, the size of nodes and containers can be changed after they have been created. The Cinco properties, in which the attributes of components can be changed, is located in the lower area.

4.3 KIELER SCCharts Tool Suite and Compiler

The KIELER SCCharts tool suite [7] was developed as part of the Kiel Integrated Environment for Layout Eclipse Rich Client project (KIELER). Using interactive incremental compilation, it provides the user with a better understanding of the compilation process and language features. The user gains more control over compilation in terms of compilation strategy and intermediate results. The modeling language is the SCCharts language.

Figure 4.1 shows a screenshot of the KIELER SCCharts tool suite. The left window displays the textual editor, where the user can utilize SCCharts Textual Language (SCT) to define corresponding SCCharts. The tool responds to changes in SCT and showcases the generated SCChart in the middle window. The lower window presents the compilation process, offering the user the option to scrutinize individual compilation steps more closely. By clicking on one of the transformations, the interim result is displayed in the middle window. In this case, it represents a transformation from the original (on the left) to Core SCCharts (on the right). However, it's also possible to view the ultimately generated code (e.g., C, Java, etc.) here, depending on the selected target language. Additionally, in the right window, users can customize the layout, enabling configuration of the graphical views of the SCChart.

The KIELER Compiler Command-Line Interface (KIELER Compiler CLI) provides the same functionality as the KIELER SCCharts tool suite but without a graphical user interface. The compiler is accessible from the command line interface and it takes files in SCT format (*.sctx) as input. Additional parameters can be used to specify the target language, output folder and other settings. Source files and additional information can be found on the project website[4].

[4]https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Home

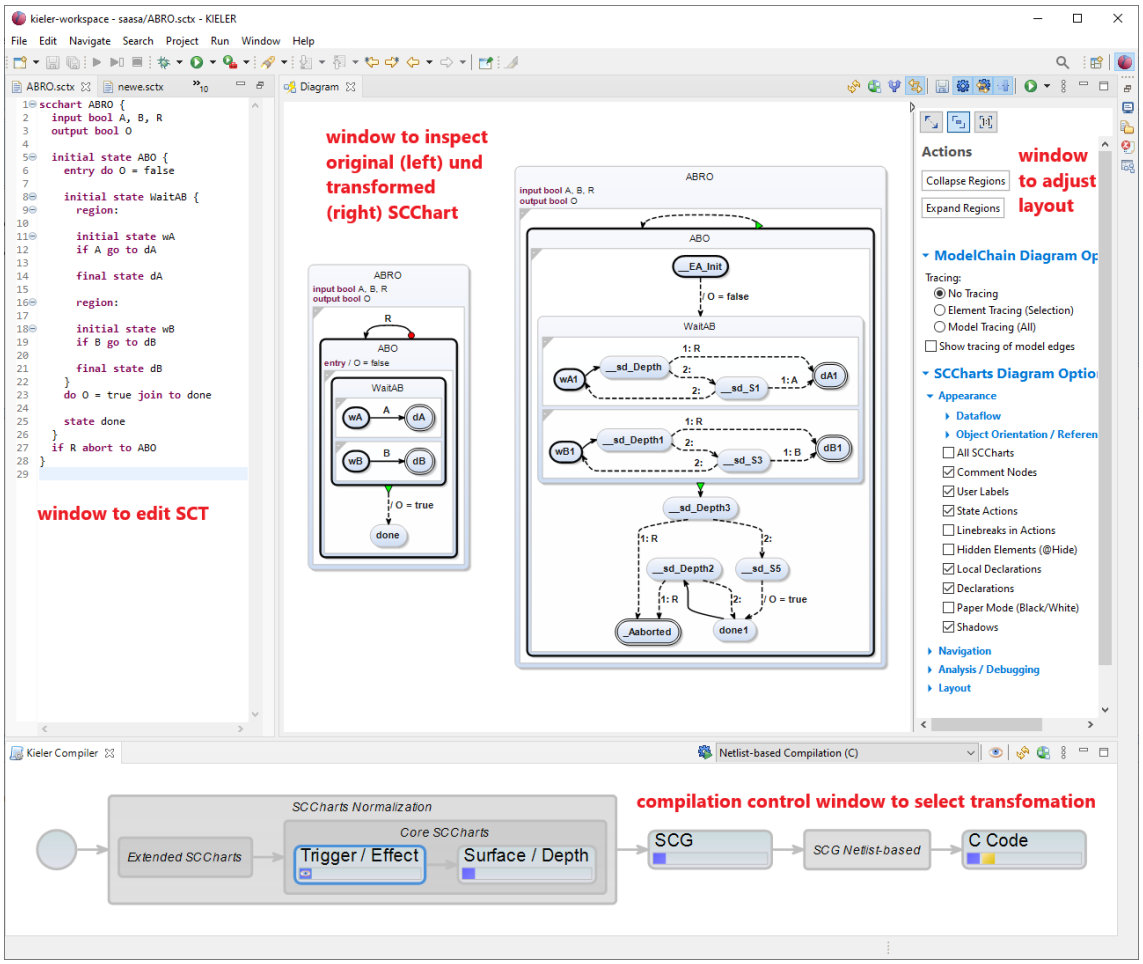

Figure 4.1: Screenshot of KIELER SCCharts tool (adapted from [7])

Chapter 5

Development of the Graph Language

In this chapter the concept and implementation of the graphical DSL tool is presented. First, the requirements for the editor are defined, and a suitable data structure with associations and attributes is designed, based on the foundations in Section 2.2. Subsequently, the MGL is implemented, which uses the designed data structure as a basis. In addition, a style is created in the MSL for each component in the MGL, which defines the appearance of the element in the editor. After that, plug-ins for a better user interface are implemented. Finally, the realization of the code generator is presented, which can generate Java or C code from the created SCCharts models.

5.1 Requirements

In the following, requirements of the editor are defined, which the editor should fulfill in order to be used as a suitable tool.

1. The underlying data structure correctly represents important relationships and attributes of the components of the SCCharts language. This means that all components can be created with their corresponding attributes. In addition, they are in correct relation to each other, so for example no simple states can be created in simple states or no transitions can start from the root state. In this way it is prevented that wrong models which do not correspond to the syntax of SCCharts can be created.

2. The visual syntax allow clear identification of the individual components of the SCCharts language. Users can thus assign each component created in the editor to the elements of the SCChart language. This prevents misinterpretation of components.

3. The user interface is intuitive and easy to understand. Users can create and customize a model in the editor. The components that can be created are listed in order, in the margin of the editor. When they are created, they dynamically adapt to the model. For example, when creating declarations, they are automatically displayed at

the top of the corresponding superstate or root state. The same applies to deleting components. This allows users to create and edit models more efficiently.

4. Errors in models are indicated by validation with in the editor. If models do not comply with the syntax of SCCharts, the corresponding erroneous components are indicated to the user. In this way, the user can correct the errors and no incorrect models are created.

5. The editor provides code generation from designed models to integrable Java or C code. This allows the created model to be directly integrated into software systems, which improves the efficiency of development with the editor of such systems.

Based on these core requirements, designs are now being created to facilitate subsequent implementation.

5.2 Data Structure

The data structure must contain all relevant attributes and associations to adequately represent the elements of the SCCharts language. For this purpose, the individual elements of SCCharts that are to be implemented within the scope of the editor are first examined in more detail, then the relationship between them is considered.

5.2.1 Model Elements

Figure 5.1 shows all elements of the data structure as classes and subclasses that shall be implemented in the MGL.

SCChart, Root State and Region

The basis is the SCChart model itself, which must have a unique name in form of a string and no other attributes. In addition, there is the root state, of which exactly one exists and which must also have a unique name. Optionally, a label can be added that is displayed as a name when it is set. The same applies to regions, with the exception that here both the name and the label are optional.

Superstate and Simple State

Superstates and simple states are additionally extended by initial, final and initial final states. In the case of simple states, additional connectors are added in the form of a subclass. Each of these states has an individual name and optionally a label, similar to the previously described root state class.

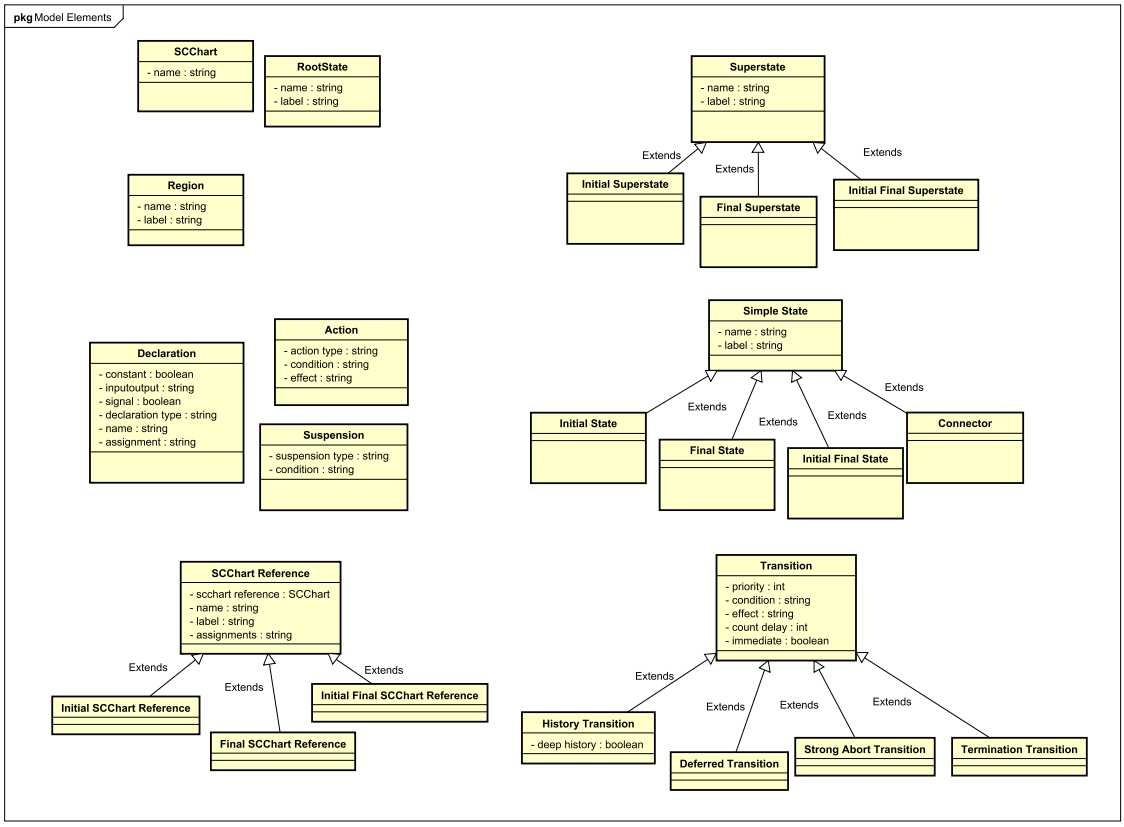

Figure 5.1: Classes of the individual components

Declaration, Action and Suspension

Declarations can occur on the one hand as constants or signals, which is represented by a boolean value. In addition, they can be declared as input, output, input output or local variable. This is made possible by the string data type, as there are more than two options. In addition, they have a name, a declaration type (for datatype) and optionally an assignment, all three of which are represented as strings. Actions and suspensions each have a type and a condition (also known as a trigger), which are defined as strings. In addition, actions have an effect, which is also a string.

Transition

Transitions have a priority and a count delay, both of which are represented as integers. In addition, they have a condition (trigger) and an effect (action), which are represented as strings. Furthermore, it is specified whether they are immediate or delayed, whereby this is represented with a boolean value.

The transition class is extended by four subclasses: history transition, deferred transition, termination transition and strong abort transition. The history transition also has a boolean attribute that is used to display deep or shallow history.

In addition to the transition subclasses mentioned, there are other combinations of transition types that are made up of the transition subclasses. An example would be a history transition that is also deferred, i.e., a deferred history transition. This results in a total of twelve different possible transitions. Namely: *transition*, *termination transition*, *strong abort transition*, *deferred transition*, *history transition*, *deferred termination transition*, *deferred strong abort transition*, *termination history transition*, *strong abort history transition*, *deferred history transition*, *deferred strong abort history transition* and *deferred termination history transition*.

SCChart Reference

The last component of the data structure is the SCChart reference class. Similar to the superstate class, it has subclasses including initial, final, initial final SCChart references, along with same attributes of the superstate class. Furthermore, it contains a SCChart reference with a SCChart type, and assignments in the form of strings that are necessary for the inputs and outputs of the referenced SCChart.

5.2.2 Model Elements Associations and Relations

Now that the individual components of the data structure have been defined, the relationships and associations between them can be considered. Figure 5.2 represents the class diagram of the data structure that is to serve as the basis for the SCCharts model editor. All associations between the components are illustrated. The fundamental element for this diagram is the SCChart class, which serves as the root element for all other elements.

The SCChart class contains exactly one root state, which, in turn, contains at least one region and can have an arbitrary number of declarations, suspensions, and actions. Within regions, there can be an arbitrary number of simple states or their subclasses, from which any number of transitions can originate and enter. Furthermore, regions can contain multiple superstates, each of which can have an arbitrary number of regions, declarations, suspensions, and actions. Additionally, an arbitrary number of transitions can connect these elements with others. Lastly, regions can contain any amount of SCChart references, which can, in turn, be linked with an arbitrary number of transitions. Additionally, these SCChart references reference one SCChart.

The created data structure can now be used for the model transformation.

5.3 Model Transformation

The model transformation of each Cinco Product is based on the MGL and the MSL, in which all components and their visual representation are defined. In addition to the graph model, the MGL consists of further components of the type node, edge and container.

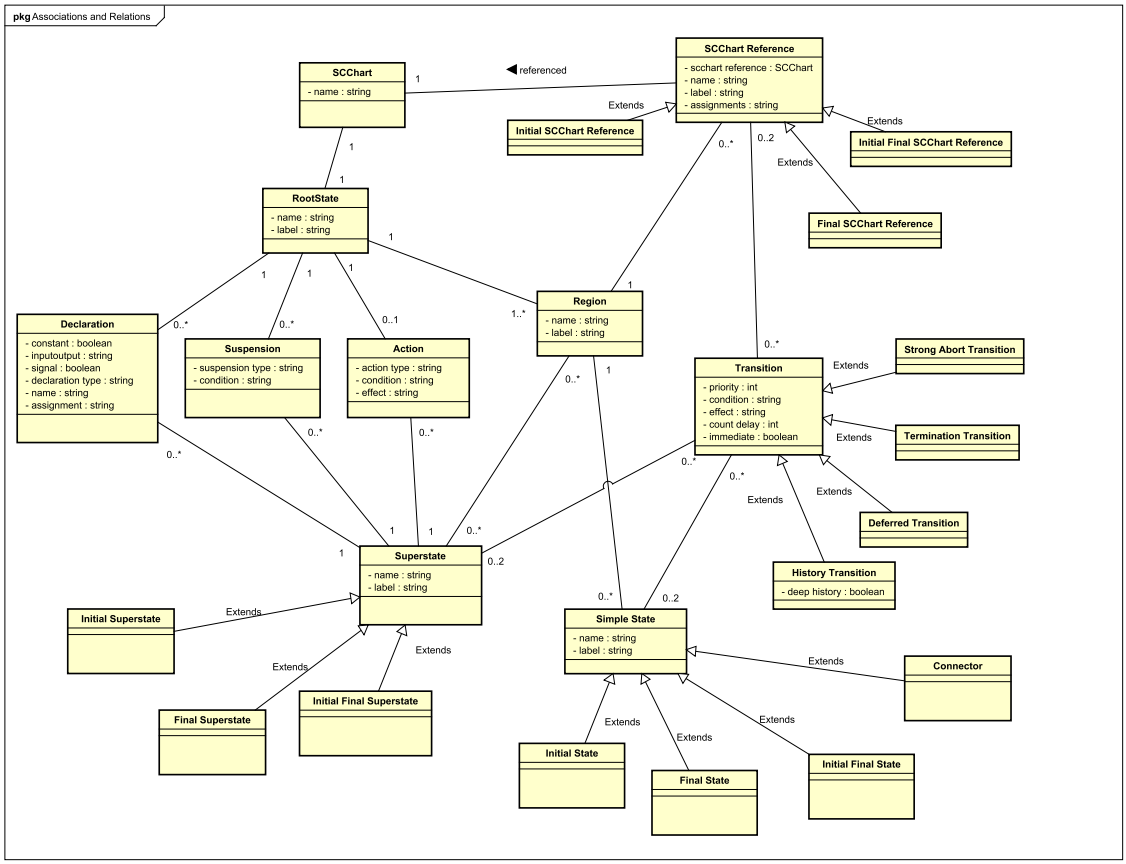

Figure 5.2: The class diagram of the designed data structure

The elements of the designed data structure have to be mapped to the types of the meta graph model of Cinco for the implementation of the MGL. It is crucial to determine which component of the data structure can best be represented by which type of the Cinco meta graph model.

In Figure 5.3, the meta graph model types of Cinco were mapped to the designed SCChart data structure. The class diagram of Cinco's meta graph model elements can be seen at the top left, with the components graph model (blue), container (green), node (red) and edge (yellow). Below this is the designed SCChart data structure, whose classes have been colour-coded according to Cinco's meta graph model elements. The graph model was assigned to the SCChart class. All components with inner behaviour, i.e. those that can contain further containers and nodes, are elements of the type container. These include root state, region, superstate and SCChart references. Declaration, suspension, action and simple states, on the other hand, are components of the node type. Transitions were assigned accordingly as edges.

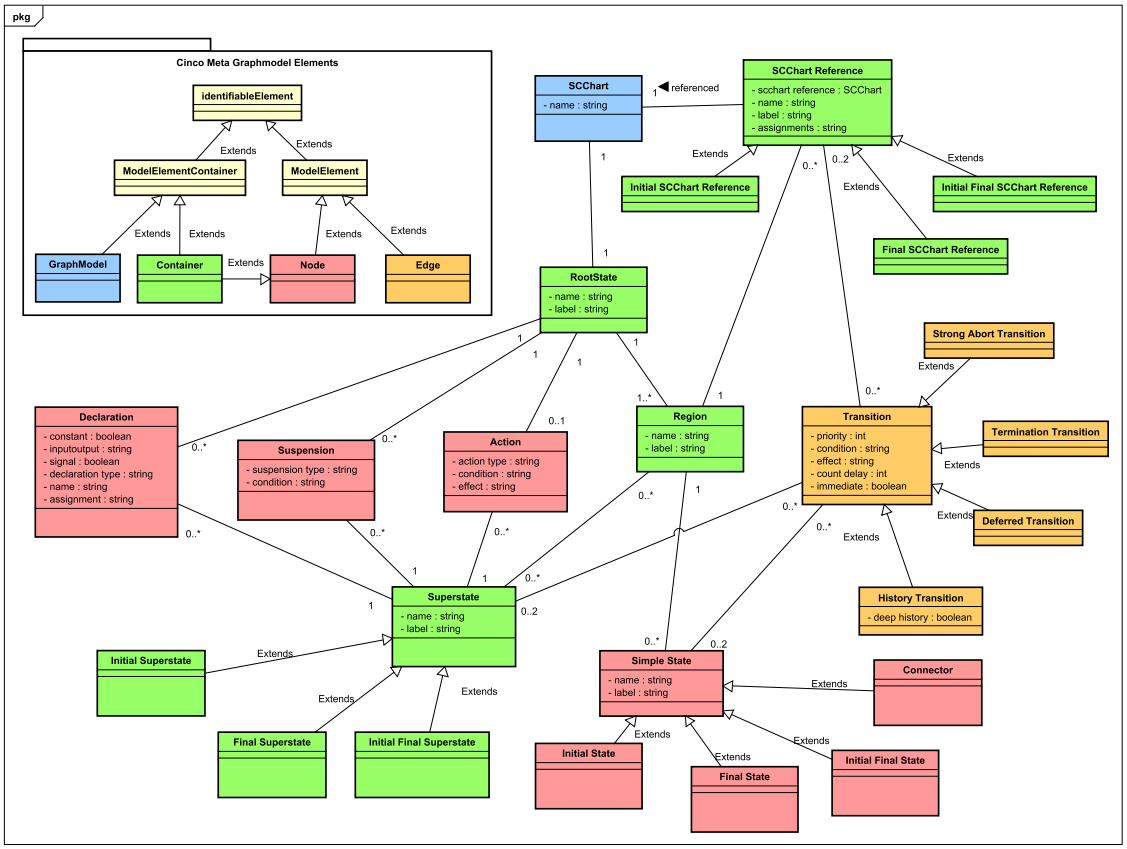

Figure 5.3: Mapping of Cinco's meta graph model elements to the designed data structure for the SCCharts editor

5.3.1 Implementation of the Meta Graph Language

With the mapping as a basis, all components of the data structure are now implemented with the corresponding Cinco's meta graph model type. In this section, only one representative for each meta graph model type is considered in more detail, as the implementation for the same types is similar. For the entire implementation of the SCChart MGL, see Appendix A.1.

Graph Model

In Listing 5.1 is the implemented graph model of which there is only one in each MGL. After the keyword *graphModel* the name of the component is given. In this case it is *SCChart*. Then *diagramExtension* is set in each graph model. This determines which file format the created graph model has. Here it is **.scchart*. *ContainableElements* specifies which components SCChart graph model can contain. The data structure indicates that SCChart model has only one root state. Limits for the number of components can be realised in the MGL using square brackets. In addition, attributes can be defined in graph model, which is done here with name.

```
graphModel SCChart {
  diagramExtension "scchart"
  containableElements(RootState[1,1])
  attr EString as name
}
```

Listing 5.1: The SCChart graph model implementation from the SCChart MGL

Container

Superstate was selected here as a representative for a component of the type container. The implementation can be seen in Listing 5.2. All components except the graph model have a style, which is shown here with *style* followed by the name of the style defined in the MSL. The string is an Xtext expression that ensures that if the label of Superstate is set, the label is passed along, otherwise the name. This allows the label for the component to be displayed in the editor if it is specified. Subsequently, in container, as with graph model, *containableElements* can be used to determine which components superstate can contain. According to the data structure for superstate these are regions, declarations, actions, suspensions. *IncomingEdges* and *outgoingEdges* define which transitions can enter and exit. If these are not specified, no transitions can enter or exit. For Superstate, these are all transitions. This could actually be realised with '*', for all edges. However, since an abstract transition is defined in the MGL for the implementation and this should not be included in the model, all transitions must be entered here individually. With a design decision without abstract transion, the MGL would be significantly shorter at this point due to the '*'. Finally, the attributes for name and label are defined. For containers, nodes and edges, a class can be extended as usual in programming using the keyword *extends*. In the MGL, the defined properties of the superclass are inherited. In this way, initial superstate, final superstate and initial final superstate are realised, whereby a style must still be specified here.

```
container SuperState {
  style superStateStyle("${empty label ? name : label}")
  containableElements(Region, Declaration, Action, Suspension)
  incomingEdges(Transition, TerminationTransition, StrongAbortTransition,
    DeferredTransition, TerminationDeferredTransition,
    StrongAbortDeferredTransition, HistoryTransition,
    TerminationHistoryTransition, StrongAbortHistoryTransition,
    DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
    TerminationDeferredHistoryTransition)
  outgoingEdges(Transition, TerminationTransition, StrongAbortTransition,
    DeferredTransition, TerminationDeferredTransition,
    StrongAbortDeferredTransition, HistoryTransition,
    TerminationHistoryTransition, StrongAbortHistoryTransition,
```

```
    DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
    TerminationDeferredHistoryTransition)
  attr EString as name := "<set name>"
  attr EString as label
}
```

Listing 5.2: The superstate container implementation from the SCChart MGL

Node

In Listing 5.1 the simple state component is shown as a representation for the type node. Nodes have similar properties to containers, except that they cannot contain elements. After the style is referenced, *incomingEdges* and *outgoingEdges* can be set, which are fewer compared to superstates. Although this is not directly evident from the data structure, it makes sense when considering the functions that some transitions have when entering or exiting a state. Since simple states have no internal behaviour, such transitions have no effect on them and would therefore be rather confusing. For this reason, the deferred and history transitions are missing from the incoming edges, and the strong abort and termination transitions are missing from the outgoing edges. Subsequently, attributes can be specified, in this case the name, which is already initialised, and the label. Initialising name with *<set name>*, which has already been done for superstates, is to ensure later that the user gives the state a unique name.

```
node SimpleState {
  style simpleStateStyle("${empty label ? name : label}")
  incomingEdges(Transition, TerminationTransition, StrongAbortTransition)
  outgoingEdges(Transition, DeferredTransition, HistoryTransition,
    DeferredHistoryTransition)
  attr EString as name := "<set name>"
  attr EString as label
}
```

Listing 5.3: The simple state node implementation from the SCChart MGL

Edge

To better distinguish (normal) transition from other transition types, the superclass abstract transition was added, from which all twelve transition types inherit. Listing 5.4 shows the implementation of the deferred history transition. The string contained in the referenced style is an Xtext expression with the function that the label of the edge has the format '[priority:] [[count delay] condition] [/ effect]'. Only the attributes of the label that have been set are displayed within the label. A second string is given for history transitions that is used for displaying deep history. Besides style, only attributes for edges can be defined.

```
edge DeferredHistoryTransition extends AbstractTransition {
  style deferredHistoryTransitionStyle("${priority.concat(': ').concat(
    condition == '<No condition>' || empty condition ? '' : (empty
    count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
    effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
    '*' : ' '}")
  attr EString as condition := '<No condition>'
  attr EString as effect := '<No effect>'
  attr EString as count_delay
  attr EBoolean as deepHistory
  attr EBoolean as immediate
}
```

Listing 5.4: The deferred history transition edge implementation from the SCChart MGL

Prime Viewer Meta Plug-In

Listing 5.5 shows the Prime Viewer plug-in. It is used to implement the reference in the data structure from the SCChartReference to an SCChart model. This must be activated before the graph model with the annotation *@primeviewer*. Afterwards the annotation *@pvFileExtension(...)* can be used to search for files of this format which contain the referenced model element. In this case the file extension is *scchart* and the model element is SCChart. This is then assigned to *reference*, which basically behaves like an attribute and contains all the information of the referenced SCChart.

```
container SCChartReferece {
  style sCChartRefereceStyle("${empty label ? name : label} @ ${reference.
    name} (${assignments})")
  containableElements(RegionRef)
  incomingEdges(Transition, TerminationTransition, StrongAbortTransition,
    DeferredTransition, TerminationDeferredTransition,
    StrongAbortDeferredTransition, HistoryTransition,
    TerminationHistoryTransition, StrongAbortHistoryTransition,
    DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
    TerminationDeferredHistoryTransition)
  outgoingEdges(Transition, TerminationTransition, StrongAbortTransition,
    DeferredTransition, TerminationDeferredTransition,
    StrongAbortDeferredTransition, HistoryTransition,
    TerminationHistoryTransition, StrongAbortHistoryTransition,
    DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
    TerminationDeferredHistoryTransition)
  @pvFileExtension("scchart")
  prime this::SCChart as reference
  @readOnly
  attr EString as inputsOutputsOfRef := ""
  attr EString as name := "<set name>"
```

```
176   attr EString as assignments := ""
      attr EString as label
178 }
```

Listing 5.5: The Prime Viewer meta plug-in for SCChart reference

5.3.2 Implementation of the Style Graph Language

The visual appearance of the components from the MGL is defined in the MSL. In node styles, the visual representation of components of the type node or container can be defined and in edge style for components of the type edge. In this section, only one representative each of the node style and the edge style will be examined in more detail, as the implementation process of these styles is highly repetitive. For the entire implementation of the SCChart MSL, see Appendix A.2. In general, it is tried to stay as close as possible to the original syntax and visual representation of the SCCharts language with the given graphical elements and the settings of the appearance of the MSL.

NodeStyle

In node style, any number of container shapes, such as rounded rectangle or ellipse, or shapes such as text or polylines, can be defined. Container shapes can also contain other container shapes and shapes. Listing 5.6 shows the node style of the component initial final superstate. The number in the bracket after the name determines the number of string parameters that must be provided by the components of the MGL. The name or label of the initial final superstate from the MGL is passed here. Then a container shape is implemented with *roundedRectangle* and its appearance is determined with *initialFinalStateOuterCircle*. In this case, it is a little wider line to indicate the initial feature. Then the size is set with *size()* and the rounded corners with *corner()*. In addition, another inner rounded rectangle is defined whose appearance determines the fore and background colour. With *position* the location of the inner rectangle can be set in the outer rectangle. The size and the rounded corners can be adjusted as for the outer rectangle. Thus, the component has a double border indicating the final feature. In the inner rectangle, a text is defined that receives the parameter string and displays it in the middle at the top of the inner rectangle. The visual representation in the editor of the initial final superstate is shown in Figure 5.4 left.

```
nodeStyle initialFinalSuperStateStyle(1) {
164   roundedRectangle {
        appearance initialFinalStateOuterCircle
166     size (140,80)
        corner (15,15)
168     roundedRectangle {
          appearance superStateAppearance
170       position (CENTER,MIDDLE)
```

```
      size (134,74)
172   corner (15,15)
      text {
174     appearance textFont
        position (CENTER,TOP 3)
176     value "%1$s"
      }
178 }
  }
180 }
```

Listing 5.6: The initial final superstate style implementation from the SCChart MSL

EdgeStyle

In edge styles, the appearance of the connecting line and the decorators can be defined. As with node style, a string must be given as argument for termination transition, as shown in Listing 5.7. The appearance provider is used for every transition type to change the line of the edge from solid to dashed on runtime if the transition is immediate, i.e. the boolean *immediate* is true. In this way, it is not necessary to define an equivalent immediate transition type in the MGL for each transition type. Furthermore, termination transitions are also displayed as dashed transitions if they are unconditional. The referenced class of the Appearance Provider is an Xtend class for each transition type, in which the function when to change the appearance is implemented. After that, the default appearance is set. In addition, decorators are defined that lie on or at the line. These can have predefined shapes like the first decorator (arrow). The location also specifies the relative position of the decorator to the edge. An appearance can also be specified for each decorator. The second decorator is a self-defined green triangle at the beginning of the transition, which indicates the termination of the source state. The last decorator displays the text, i.e. the label of the edge, which includes priority, condition, count delay and effect. This is displayed in the centre and can be moved because of the *movable* keyword. The representation of the termination transition in the editor is shown right in Figure 5.4.

```
edgeStyle terminationTransitionStyle(1) {
348   appearanceProvider("info.scce.cinco.product.scchart.appearance.
        TerminationTransitionAppearance")
      appearance transitionAppearance
350   decorator {
        location (1.0)
352     ARROW
        appearance transitionAppearance
354   }
      decorator {
356     location (0)
```

```
    polygon {
358   appearance greenTriangle
      points [(0,0)(0,7)(15,0)(0,-7)]
360 }
  }
362 decorator {
    location (0.5)
364 movable
    text {
366   value "%s"
    }
368 }
}
```

Listing 5.7: The termination transition style implementation from the SCChart MSL

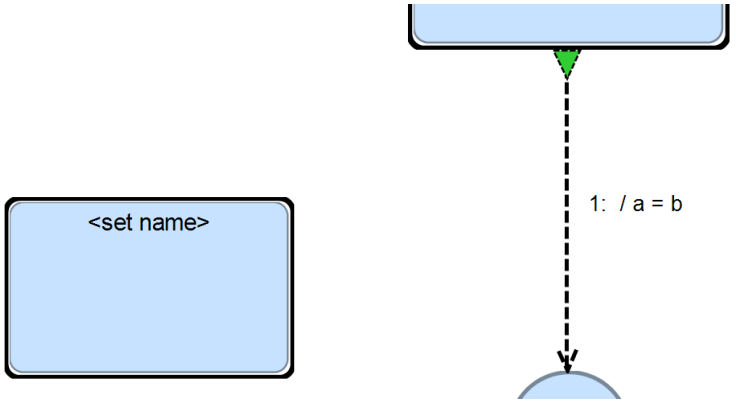

Figure 5.4: Visual representation of the initial final superstate (left) and the termination transition (right) in the editor

5.4 User Interface and Model Validation

Although it is already possible to create SCChart models with the implementation of MGL and MSL, the editor, in its current state, would not provide a good user experience. The issue is illustrated in Figure 5.5. All components of the type container and node are listed under the *node* tab on the right-hand side. A model with several components has been created in the middle. When components are created by drag and drop, they are created at the location where they are dropped with the specified size defined in the MSL. This may mean that the component created is too large for the container or that the component should not be displayed in this position. This can lead to the creation of confusing diagrams. It is also tedious to have to manually adjust the nodes and containers every time when creating new components or deleting old ones. The KIELER SCChart tool suit (Sec. 4.3) has an automatic layout which, when designing SCCharts after changes

to the SCT, automatically arranges the model appropriately. Within the scope of this bachelor thesis it is not possible to realise such a comprehensive layout program. But with the meta plug-ins from CINCO, the usability of the editor can be improved with little effort. Therefore, it would be useful if declarations, suspensions and actions were arranged in a superstate or root state directly in the upper area when they are created and the remaining declarations, suspensions and actions are rearranged when they are deleted in order to close any gaps that occur. The same applies to regions. These are to be integrated next to other regions with the same distance to them or also to the border when they are created. In addition, not all functions such as resizing, deleting or moving should be possible for all components, as e.g. simple states should have a constant size. In the next sections, meta plug-ins and their implementation are presented, which improve the user-friendliness of the editor.

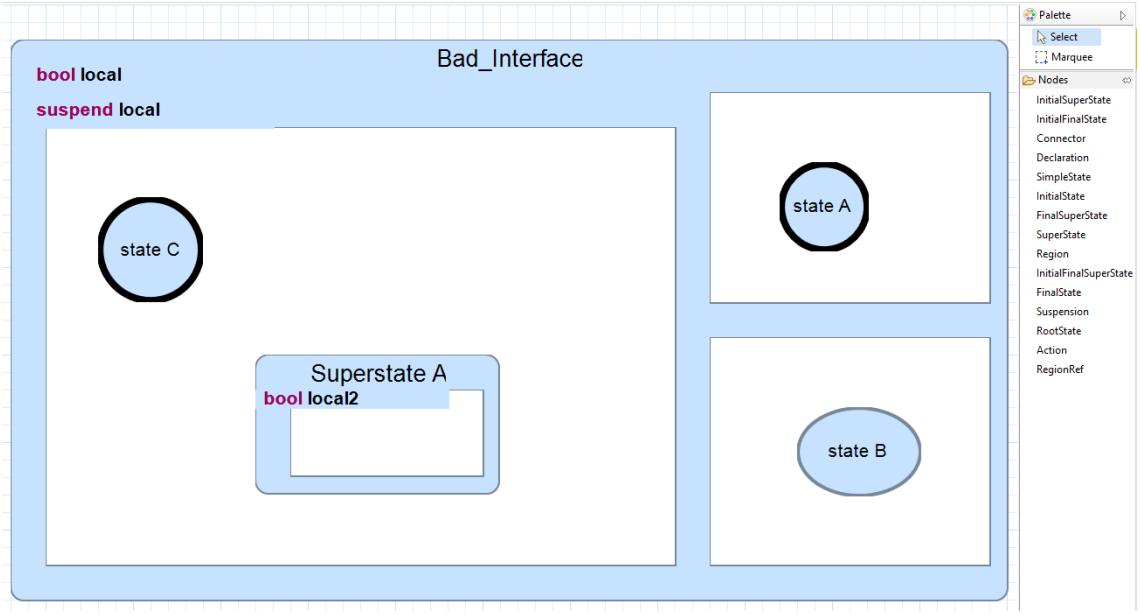

Figure 5.5: The SCCharts editor without plug-ins

5.4.1 Event API

The Event API can be used to react to various events in the editor. These events can be, for example, the creation or moving of a component. The Event API can be activated for a component in the MGL by annotation and can be applied to all component types. In the corresponding event class of the component, the functionality can be implemented for an event.

For the SCChart graph model, the Event API is used to automatically create a root state with a region after creation. This is because every SCChart has a root state. Furthermore, for superstates and the root state, regions, declarations, suspensions and actions

are automatically adjusted to the new size of the superstate or root state after a change in size.

In order for declarations, suspensions and actions to be listed in the upper left corner of their associated states, within the Event API a function is implemented for each component, which is called when the respective component is created. Listing 5.8 shows, as an example, a part of the code implemented for the *postCreate* event of action. For the arrangement of the elements in the respective states, these must be accessed. Since the superstate or root state cannot be accessed directly as the respective instance, the SCChart model is called via the method *rootElement* and then the state that contains the created action node is identified via a tree-like search. To identify the state containing the created action node, a random UUID is assigned via the Java UUID (Universally Unique Identifier) class when the action node is created (Line 33). This requires an additional attribute to be defined in the MGL for actions. To make it invisible to the user of the editor, it is hidden in the editor with the annotation *@propertiesViewHidden*. The algorithm first iterates the root state and searches in the list of actions (if not null) for an element with the UUID of the created element (lines 35-37). If the action node with the UUID is found in the root state, action nodes are reordered with the created action (lines 38-51). To do this, the number of declarations and suspensions are first determined (lines 38-44) in order to be able to place the action nodes below them. Then a check is made to see if a region has been obscured by the placement of the action node. If this is the case, the region is reduced accordingly (lines 53-60). If the created action is not in the root state, the superstates in the regions of the root state are searched for the element (lines 65-73). The method *postCreateAction()* (Line 69) is similar in structure to *postCreate()* (Line 33). The only difference is that the action node created is searched in a superstate and instead in a root state. This method is also recursive, i.e. in each superstate in which the element was not found and in whose regions there are still superstates, the function is called again for the superstates contained. Similar methods were used for post creations of declarations or suspensions. And similar procedures are also implemented for post deletion of corresponding components, which ensure that no gaps exist within declarations suspensions and actions, and regions resize accordingly when space is freed up at the top of the state. For the entire implementation of the Xtend ActionEvent class, see Appendix A.3.

```
32  override postCreate(Action element) {
      element.uuid = UUID.randomUUID.toString
34    var boolean continue = false
      if (element.rootElement.rootStates.head.actions !== null) {
36      for (action : element.rootElement.rootStates.head.actions) {
          if (action.uuid == element.uuid) {
38          var int declarationCount = 0
            if (element.rootElement.rootStates.head.declarations !== null) {
```

```
40            declarationCount = element.rootElement.getRootStates.head.
    declarations.size
          }
42        if (element.rootElement.rootStates.head.suspensions !== null) {
            declarationCount += element.rootElement.rootStates.head.
    suspensions.size
44        }
          for (var i = 0; i < element.rootElement.rootStates.head.actions.
    size; i++) {
46          element.rootElement.rootStates.head.actions.get(i).x = 10
            element.rootElement.rootStates.head.actions.get(i).y = 30 + 13 *
     declarationCount + 13 * i
48          element.rootElement.rootStates.head.actions.get(i).width =
    element.rootElement.rootStates.head.
              width - 20
50          element.rootElement.rootStates.head.actions.get(i).height = 13
          }
52        declarationCount += element.rootElement.getRootStates.head.actions
    .size
          if (element.rootElement.rootStates.head.regions !== null) {
54          for (region : element.rootElement.rootStates.head.regions) {
              if (region.y < 30 + declarationCount * 13) {
56              region.y = region.y + 13
                region.height = region.height - 13
58            }
            }
60        }
          continue = true
62      }
      }
64    }
      if (!continue) {
66      for (region : element.rootElement.rootStates.head.regions) {
          if (region.superStates !== null) {
68          for (superState : region.superStates) {
              postCreateAction(superState, element)
70          }
          }
72      }
      }
74  }
```

Listing 5.8: Part of the implementation in Xtend for the postCreate method of the action event class

The Event API is also used to arrange regions when they are created in a state. Again, the root state or superstate in which the region is created must be accessed, so a similar search algorithm is used as for actions. For this, region also receives a hidden UUID

attribute in the MGL. If state containing the created region is found, the region next to it (if exists) is halved (minus the distance between them) and the region created is placed on the area that becomes free with the same size as the halved region. For better understanding an example for this is shown in 5.6. The arrows indicate on which side the new region was created and what the superstate looks like after the regions have been arranged. If a region is created between two regions, the left or upper region is first used to create the new region. If there is no region in the state, the created region will have the entire size of the state minus the distance to the borders. During the implementation, cases occurred where the region was simply placed but not arranged, which could lead to the creation of incorrect models. For this case, the algorithm was extended so that the region is deleted automatically if it is not arranged, so that the user can try to create a new region again. For the post delete function of regions, the create function is basically reversed. Again, the state that contained the deleted region is searched first, and then a neighbouring region of similar size to the deleted region is searched for. In this way, the region takes up the area of the deleted region without covering other regions or leaving free areas in the state. With these simple methods of the event classes, components can be dynamically created, deleted and resized.

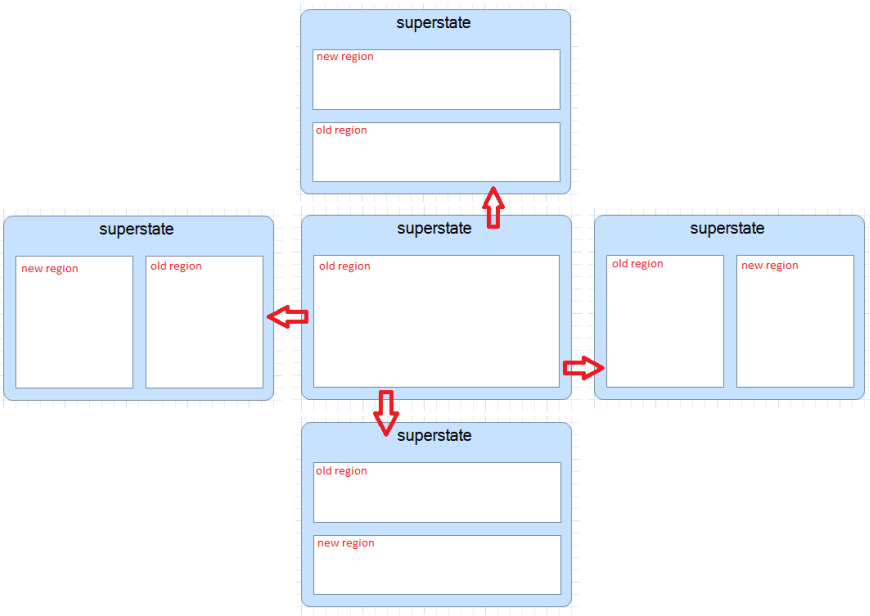

Figure 5.6: Ordering mechanism of created regions in states

5.4.2 Palette, Disable, Possible Value Provider

Besides the Event API, which can be used to implement many different functions, there are also smaller plug-ins that can be applied to improve the user interface. These are briefly presented below.

With the palette plug-in, containers and nodes can be grouped under their own term on the right-hand side of the editor, which improves clarity for the user. For example, superclasses can be combined with subclasses under the *Superstates* tab. In addition, components can be hidden if no term is specified in the palette name. This is used for the root state, as it only exists once in SCChart and is created directly when an SCChart model is created via the Event API.

In the editor it is desirable that certain functions, such as resize or move, are not available for some components. Otherwise, the user can make changes to the model that are not intended. *@disable* can be used for this purpose by disabling the functions move, select, create, resize and delete in the editor for certain components. Thus, resize is disabled for simple states and their subclasses, as their size should be adjustable. Furthermore, move and resize are disabled for region and declarations, suspensions and actions, as their arrangement and resizing is controlled by the Event API. In addition, delete has been disabled for root state, as the user should not be able to delete it.

The Possible Value Provider plug-in can be used to specify certain values for attributes of components to ensure that users cannot make invalid entries. This is used in declarations, actions and suspensions, e.g. to define the type. In addition, the possible selection of the priority of outgoing transitions for states is defined, whereby the selection option is bound to the number of outgoing edges.

5.4.3 Model Compare and Merge Framework

The Model Compare and Merge (MCaM) framework is a plug-in that allows validations to be made within Cinco editors during the creation of models. To do this, it is activated with *@mcam("check")* in the MGL. Subsequently, with *@mcam_ checkmodule()* the classes in which the checks are implemented are registered in the MGL. In the classes, functions can be created that check the model in the editor according to the specifications defined in the functions, for example that each region contains exactly one initial state or superstate, or that the priorities of the outgoing transitions of a state have a valid order. An example of such a function shows Listing 5.9. The function is part of the class for checking transitions. It can be used to check whether the source element and the target element of each transition of the model are in the same region. If this is not the case, an error message is displayed for the corresponding transition. These are displayed to the user in the editor in the *Model Validation View* window. Other, but certainly not all, checks are implemented. For a better overview during implementation, it makes sense to create a separate check module for each class or superclass of the MGL.

```
62  def checkTransitionInRegion(SCChart scchart) {
      scchart.find(AbstractTransition).forEach [
64      if (it.getSourceElement.getContainer != it.getTargetElement.
    getContainer) {
```

```
        it.addError("source and target element of transition must have the
    same region")
66      }
    ]
68  }
```

Listing 5.9: Function of transition check class

5.5 Implementation of the Code Generator

After the user interface has been adapted and validation functions realised, models can now be created. To be able to convert these models into C or Java code, a code generator must be implemented.

The Generatable meta plug-in of Cinco is a tool, that utilises template expressions and can be used to translate SCChart models into Java or C code. However, this would be elaborately, as it is not so easy to translate the features of the SCCharts language into corresponding Java and C code artefacts. A simpler method is to use the KIELER Compiler CLI from Section 4.3, which can generate C or Java code from SCT. In addition, this reduces error-proneness, since on the one hand the tool only compiles syntactically correct SCT and on the other hand the tool has a proven correct compiler. For the generator this means that it would have to convert the components of the model into the SCT format and then pass them to the KIELER Compiler CLI.

In Figure 5.7 Core SCCharts and Extended SCCharts are shown with SCT translations for each component in the margin. Most of the SCT translations are already illustrated here. The only missing component are the SCChart references, which need to be called at the beginning of an SCT file with an import statement, and further defined in a second call.

For the model to text transformation, a function is defined for each component in the code generator plug-in, as shown in Figure 5.8. This function is responsible for placing keywords and attributes of its component at the appropriate location in the SCT file. For components of type container, the containing components are invoked along with their defined functions.

In Listing 5.10 the template function is shown. It is used to translate the root state of the SCChart graph model into SCT. The generation of the other components of the SCChart model originates from here. In Line 60 and 86 the start and end of an template expression is indicated, i.e. text indentation is taken into account outside the french prefixes within this section. In lines 61-63, the imports for possible referenced SCCharts are converted. Then follows the formatting for the root state with *scchart* as the keyword and, separately, the name of the root state. Within the curly brackets, the declarations,

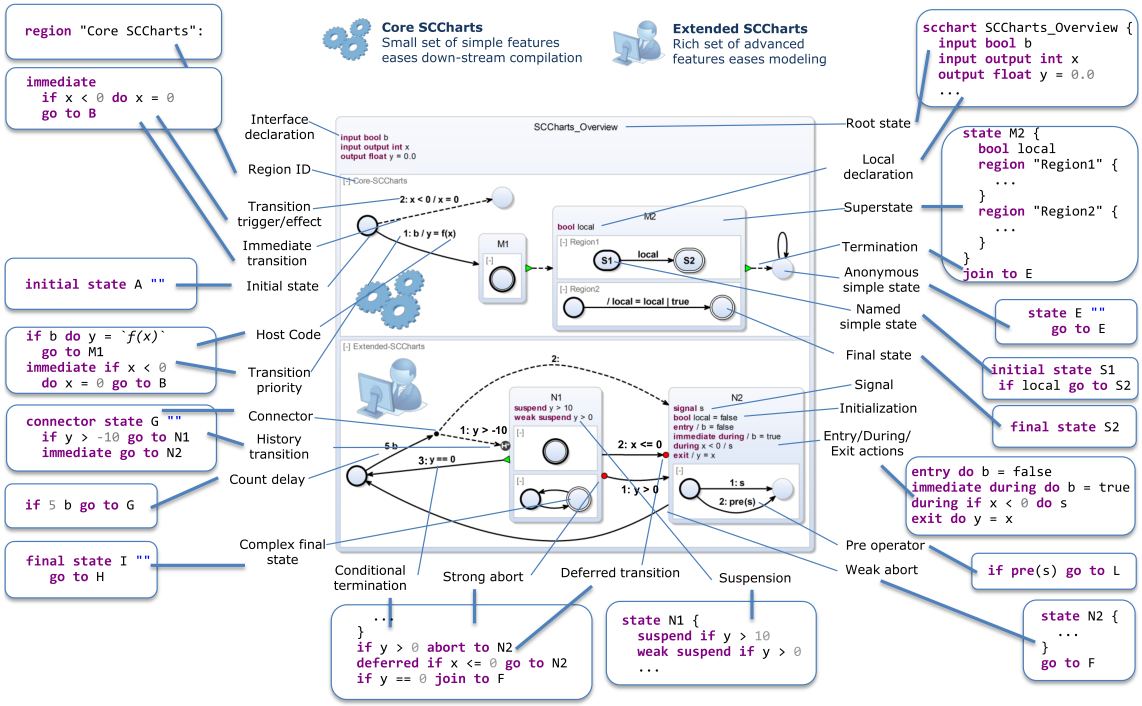

Figure 5.7: Core and Extended SCCharts with SCT annotation

suspensions, actions, and regions contained in the root state in this order are then called with the corresponding functions (lines 65-84).

```
60  def template(SCChart scchart) '''
    «FOR sCChartReferece :scchart.find(SCChartReferece)»
62  import "«sCChartReferece.reference.file.name.substring(0, sCChartReferece.
      reference.file.name.lastIndexOf('.'))»"
    «ENDFOR»
64  scchart «scchart.rootStates.head.name» {
      «IF scchart.rootStates.head.declarations!=null»
66    «FOR declaration : scchart.rootStates.head.declarations»
      «declaration.genDeclaration»
68    «ENDFOR»
      «ENDIF»
70    «IF scchart.rootStates.head.suspensions!=null»
      «FOR Suspension : scchart.rootStates.head.suspensions»
72    «Suspension.genSuspension»
      «ENDFOR»
74    «ENDIF»
      «IF scchart.rootStates.head.actions!=null»
76    «FOR action : scchart.rootStates.head.actions»
      «action.genAction»
78    «ENDFOR»
      «ENDIF»
80    «IF scchart.rootStates.head.regions!=null»
```

```
    «FOR region : scchart.rootStates.head.regions»
82  «region.genRegion»
    «ENDFOR»
84  «ENDIF»
  }
86  ''',
```

Listing 5.10: Template function of the SCChart code generator

In SCT, priority is regulated so that transitions with the highest priority are placed directly after the source element, followed by transitions with the second-highest priority, and so on. For this purpose, an extra function, *genEdgesOrder(...)*, is implemented that outputs the transitions for components with outgoing edges in the correct order after the component. The generated file with the file extension *.sctx is saved in the folder specified in the MGL in the workspace.

In addition, a command line call with the method *commandLineParser* is implemented, that invokes the KIELER Compiler CLI and passes the generated file in SCT format along with instructions for the target language as parameters and target directory. The target directory is the folder in which the SCT file is located. The KIELER Compiler CLI offers different compilation variants for Java and C. It also offers the possibility to convert the SCT file into a diagram, which can be useful for validating the created model with the editor. In total, there are seven different types of compilation for files with SCT format. For this reason, another attribute is defined in the MGL for root state, which is used in the editor to select the target language of the code generator. The plug-in Possible Value Provider is used for this purpose. With this, the user can select one of the possible compilation types. This code generator is written for the Windows operating system, as it calls the *cmd.exe*. However, it can easily be implemented for other operating systems by calling the appropriate terminal application.

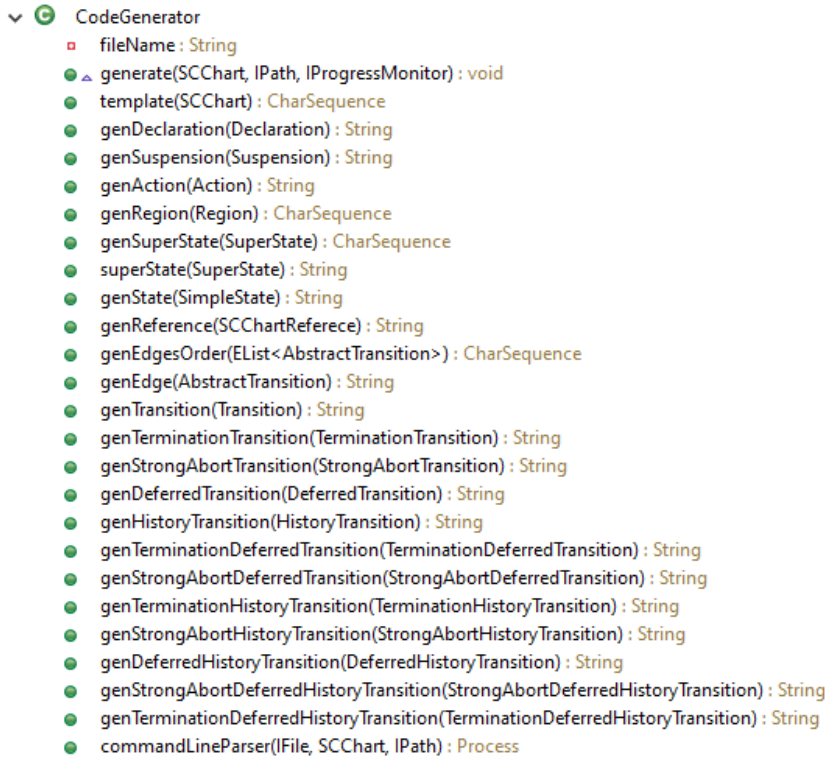

Figure 5.8: Methods of the implemented Xtend class from code generator plug-in

Chapter 6

Evaluation of the Model Editor

In this chapter, the created editor is evaluated. It is checked whether the requirements defined at the beginning of the development have been successfully implemented. For this purpose, components of the editor, such as the user interface, the visual syntax of the models or the code generator are examined more closely. Finally, a conclusion is drawn.

6.1 Result of the Editor Development

In this section, various parts of the editor are evaluated. These should fulfill the requirements defined at the beginning of the development process. In short, these are a correct data structure, a clearly visual syntax, a user-friendly interface, validation when creating models, and the ability to generate integrable codes from the model.

6.1.1 User Interface Evaluation

Figure 6.1 shows a screenshot of the user interface of the created SCCharts editor. In the centre is a SCChart model, which can be edited and customised. By clicking on an element in the editor, its attributes are displayed under Cinco Properties here in the lower area and can thus be adjusted. Names of states are displayed on them and declarations of transition are displayed next to the connection line and can be moved freely. On the right-hand side, the SCChart components are listed by category, such as superstates or states. These can be dragged and dropped into the model and the appropriate containers. Highlighting indicates whether the container can accommodate the component being created. Transitions can also be created by dragging and dropping from source to target element. Superstates and root states can be resized and the regions and declarations they contain are adjusted accordingly. When declarations, actions or suspensions are created, they are placed in the upper right area of the state, in which they are added. The regions are arranged correctly in the states when they are created. When declarations or regions are deleted, the remaining elements in the state are reordered accordingly. In addition, new SCCharts can be created and

existing ones accessed via the project folder in the left window and thus created as an SCChart reference via drag and drop. This makes the editor a good user experience.

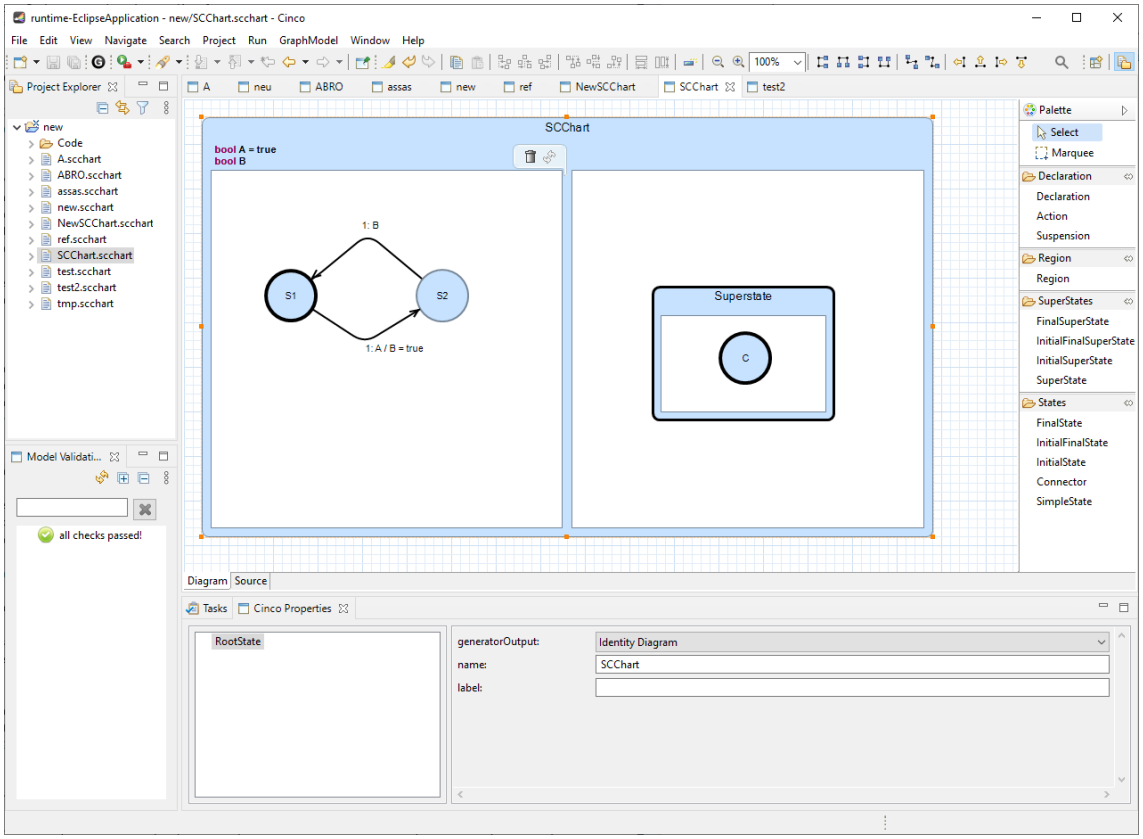

Figure 6.1: User Interface of the created SCChart Editor

6.1.2 Visual Syntax Evaluation

In Figure 6.2, the upper part shows the *SCCharts Overview* image of Section 2.2 but without labels. Almost all the features of SCCharts can be seen in this picture. Below that is an SCCharts model created using the editor developed in this bachelor thesis, with a design similar to the *SCCharts_ Overview* image. It can be clearly seen that these are almost identical models. Only the host code, which is declared last in fourth place in the root state in the original SCChart, is missing in the lower SCChart model. In addition, the priorities are indicated for components with only one outgoing transition, while these are not shown in the original SCChart. With deferred history transitions, the red circle of deferred is covered by the black circle of history. This is because only relative values can be assigned to the decorators and therefore no fixed distance to the end of the arrow is possible. In addition, the Appearance Provider plug-in changes the entire appearance in the style so that, for example, the line of the green triangle is also dashed for the termination

transition. Nevertheless, these are only minor deviations. All in all, the components of SCCharts are clearly recognisable.

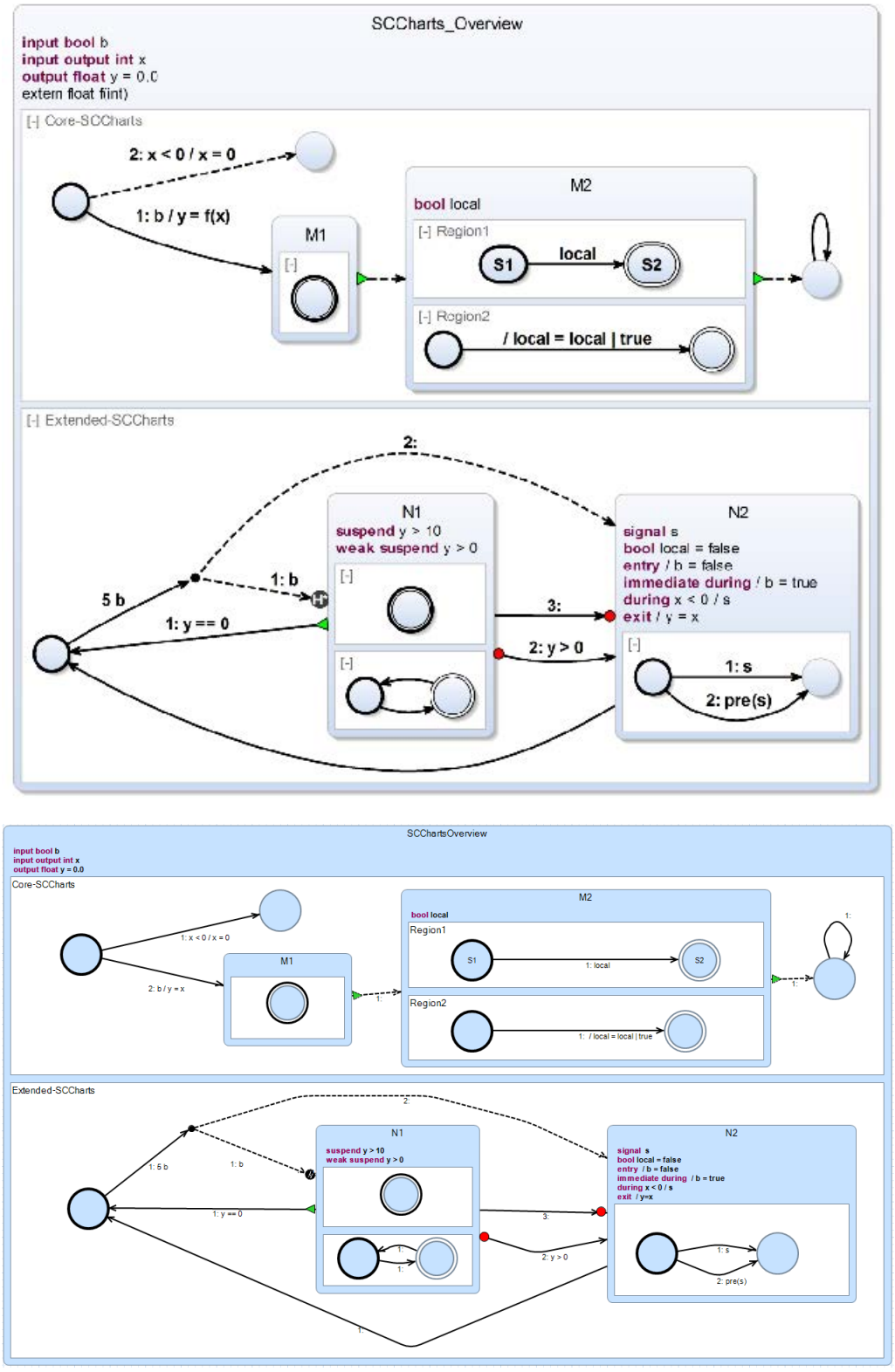

Figure 6.2: Original SCCharts model (top) and editor SCChart model (bottom)

6.1.3 Validation Evaluation

A validation was implemented via the MCaM plugin, which performs checks after saving the model. For a better overview and easier extension, a separate check class was created for each class or superclass of components of the MGL. Several validation functions are implemented.

Figure 6.3 shows some examples of validation functions. In the middle is a model with incorrect syntax. The validations are shown at the bottom left. The first error indicates that the order of outgoing transitions from an initial state, state A, is not valid. This is because both transitions originating from A have priority 1. Next, it indicates that a region does not contain exactly one initial state. This is the case in the second region of the superstate. And the last error message displayed is that for a transition, source and target are not in the same region, which is the transition from A to B. The KIELER SCChart tool would also show this model as incorrect. Firstly, the transition from A to B is not accessible, and it is not possible for two transitions to have the same priority, since this is determined by the order in which they are given first. Second, if a region does not have exactly one initial, it is marked as false. In this respect, the MCaM checks take over the semantics and syntax checks for the model that are made in the KIELER SCChart tool in the SCT editor.

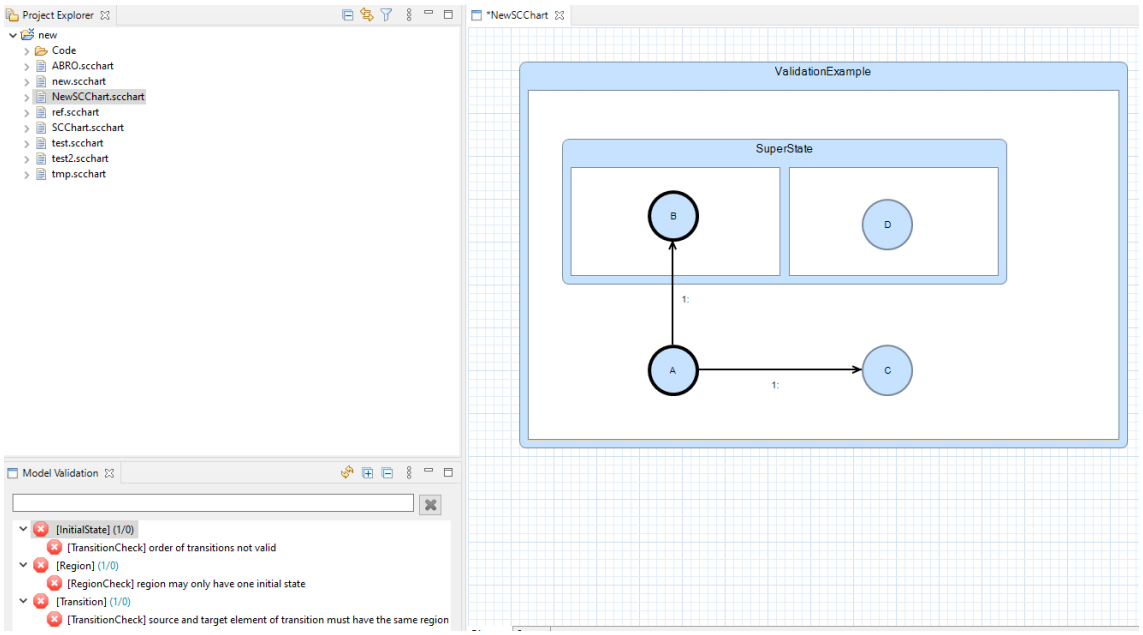

Figure 6.3: Example for MCaM plugin for SCChart Validation

6.1.4 Code Generator Evaluation

Through Cinco's code generator meta plugin it is possible to start the code generation by clicking the generate button in the user interface. This translates the currently opened

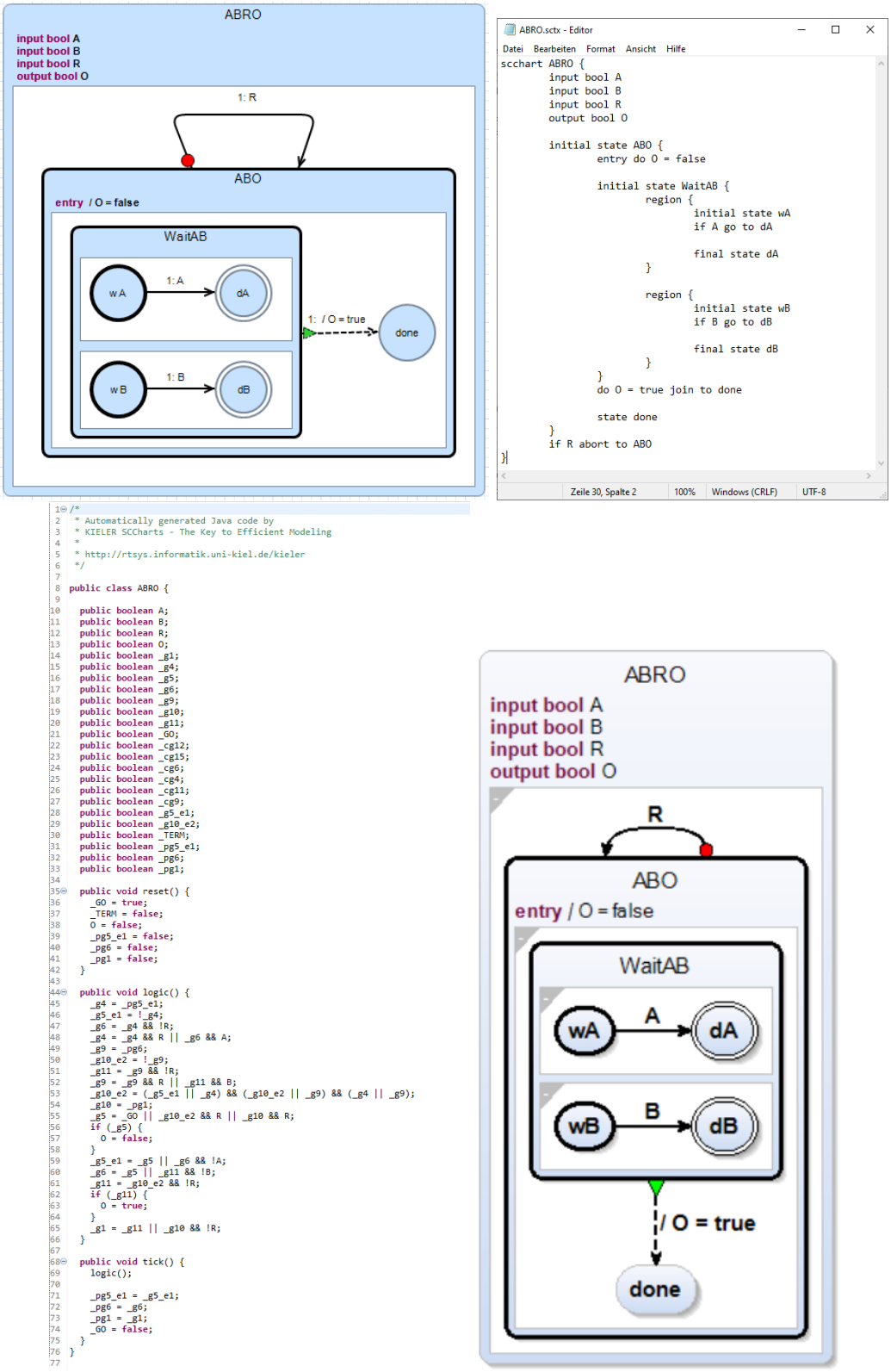

Figure 6.4: Output of the code generator plug-in from the ABRO SCChart

model into SCT and saves it in a file in the workspace. In addition, the KIELER SCCharts CLI is started via the command line. This takes the file in SCT format as input, and a parameter selected by the user via the root state that defines the target language of the output. This can be C or Java code, but also a SCChart diagram, which provides the possibility to compare the with the editor created SCChart model with that one of the KIELER SCChart CLI. In this way, the created model can be checked again to make sure that it meets the original requirements. In the context of this bachelor thesis, this is mainly useful to check if the code generator interprets the components of the model correctly.

An example of the output of the implemented code generator is presented in Figure 6.4. It shows the ABRO SCChart, the "hello world" of synchronous programming [4]. Here are the functions of concurrency and preemption illustrated in a compact example. At the top left is the SCChart model from the developed Cinco SCChart Editor. The input and output booleans are listed in the upper section of ABRO. The strong abort transition of ABO leads to the fact that if input R is true, ABO is immediately reset, the preempt function. Within ABO, another initial superstate with 2 regions to represent concurrency, is shown. The diagram of the KIELER SCCharts CLI (bottom right) can be seen as relatively identical, as the components only differ slightly by position. This means that both the SCT code (top right) and the Java code (bottom left) created from the model (top left) with the code generator of the Cinco SCChart editor can be considered correct.

However, if the model created with the SCChart editor is incorrect, the code generator also creates an incorrect SCT file, which cannot be processed by the KIELER SCCharts CLI, so that neither Java or C code nor a diagram is generated. This can also happen if, for example, a variable is specified in a condition that does not exist in the model. With an input validation that checks if the given variable is initialized, this could be prevented. Nevertheless, the code generator already works quite reliably.

6.2 Result of the Evaluation

With the conception and implementation of a domain specific graphical tool it should be shown that its development does not have to be tedious and complex as often assumed. The Cinco meta tool was the tool for this, as it can be used to create graphical DSL tools, and its main feature is its full generation from meta specifications. This means that tools generated by Cinco can be executed directly without having to adapt the code as is the case with other meta tools with semi-automatic generation. In general, Cinco follows a simplicity-driven approach that imposes restrictions in order to avoid complexity.

As the graphical DSL the SCCharts language was chosen, which was designed to be used as a visual modeling language for safety-critical applications. These bring with them a wide range of features, many of which were realised in the editor that was finally developed. On the basis of previously defined requirements and a data structure containing all the

important components and associations of the SCChart language that should be included in the editor, the implementation of the editor's MGL was started. Through the preliminary work with the data structure, the implementation of the MGL could be completed quickly. The MSL was oriented as closely as possible to the visual syntax of SCCharts. As a result, the visual representation of components was limited to the runtime of Cinco. Since every small difference in the visual representation can almost only be implemented by defining a new component in the MGL with its own style, which meant that many transitions of SCCharts differing only slightly still had to be defined in the MGL as own component. However, the Appearance provider, which allowed a few changes to be made at runtime, reduced the number of transitions to be defined in the MGL. After MGL and MSL were implemented, SCCHart models could be created, but it was neither intuitive nor dynamic, as all lay-outing was left to the user and the interface was not very user-friendly. The various Cinco meta plug-ins made it easy to modify the interface of the editor and adapt it to the DSL. They also provide a large number of functions that improve the layout of the SCChart model and thus make it possible to create more dynamic and improved models. Subsequently, the generation of integrable Java and C code from the models created with the Cinco SCChart editor was realised via a further plug-in from Cinco and with the help of an SCChart Compiler CLI. The evaluation of the editor has shown that although it is not yet perfect, it is already a great support for the graphical creation of SCChart models. And the code generator is already a very useful tool.

All in all, it was shown that a powerful and useful tool for modeling DSLs can be developed with relatively little effort. In addition, the Cinco meta tool has shown that it can be used both for DSLs that offer many features and for DSLs in the security-critical area.

Chapter 7

Conclusion

The last chapter summarizes the results of this bachelor thesis and gives an outlook on possible improvements and further extensions of the SCCharts editor developed with Cinco.

7.1 Summary

Cinco is a tool designed for the easy creation of domain-specific graphical modeling tools. To show this, the development process was demonstrated using SCCharts, a visual modeling language specifically designed for specifying safety-critical reactive systems, serving as a representative of a graph-based modeling language. First, the requirements for the editor to be developed were defined. Subsequently, a data structure was designed based on the SCCharts syntax and including all essential attributes and associations of SCCharts components. This data structure was easily mapped to the MGL component types of Cinco, which made it simple to implement the MGL. Subsequently, the visual representation of the components in the editor was defined with the MSL. To increase usability, the easily customizable meta plug-ins from Cinco were used to improve the handling of the editor. Additionally, a code generator was implemented, making use of the KIELER Compiler CLI, allowing for the generation of Java or C code and diagrams from the created model. The evaluation of the user interface, visual syntax, and code generator of the developed editor showed that the created DSL tool is a useful asset for SCCharts modeling. All in all it could be shown that with Cinco a graphical DSL tool can be created with little effort.

7.2 Outlook

In this section extensions are presented that could increase the scope or improve the usability of the editor.

7.2.1 Possible Improvements of the Developed Editor

The evaluation showed that the editor already provides a good basis for creating SCCharts. Nevertheless, the editor could be improved by a few functionalities to increase the usability.

During the code generation, it turned out that the model transformation fails if incorrect entries are made for the attributes of components. For example, if a transition has a condition with an invalid format or a variable is specified in the condition that was never declared. The file in SCT format is created, but it has a wrong syntax and cannot be compiled by the KIELER Compiler CLI. To ensure that an input of an attribute is valid, the Grammar meta plug-in of Cinco could be used. With this plug-in, an Xtext editor can be integrated into the Cinco Properties View. It can be activated in the MGL by annotation *@grammar(...)* over an attribute. In addition, an Xtext grammar must be created in which it is specified which inputs are allowed. This would reduce the error-proneness of the code generator.

In addition, a simulator similar to the one implemented in the Cinco statechart project (Figure 3.1) could be programmed. This could be used to simulate inputs to the created models in the editor, leading to a better understanding of the system and helping to avoid errors. The KIELER SCChart tool also has a simulator, but it is probably easier to refer to another Cinco editor, as the structure is more similar.

Finally, a function similar to the KIELER SCCharts tool suit could be built in, enabling the collapsing and expanding of regions within the editor. When a region is collapsed, its inner behavior is hidden, and upon expanding it, everything becomes visible again. This functionality could be implemented using the Event API, for example, allowing users to minimize a region with a double-click and restore it to full size with another double-click. Such a feature would enhance the usability, especially for larger systems with numerous superstates.

7.2.2 Extensions of the SCCharts Editor by further Features of the SCCharts Language

With the implementation of the editor developed in this bachelor thesis, many but not all features of the SCCharts language were realised. The few functions that were not included in the development process are mostly minor functions such as the integration of host code or vectors as another data type.

A major feature that was added to SCCharts with the extension of data-flow constructs based on the semantic of Lustre, a synchronous data-flow language, [3] is nevertheless missing in the editor. This extension allows to create data-flow regions within superstates and root states in addition to standard control-flow regions, as seen in Figure 7.1. For this, a separate type of region, data-flow region, would have to be defined in the MGL. Furthermore, additional components, such as for inputs, outputs and operators, as well as

a linkage of data-flow elements had to be defined in the MGL and corresponding styles created in the MSL. For code generation, a separate function can simply be created for each additional component, which is responsible for its formatting in the SCT file. This should allow the editor to be expanded to include this major feature and thus broaden the editor's scope of application.

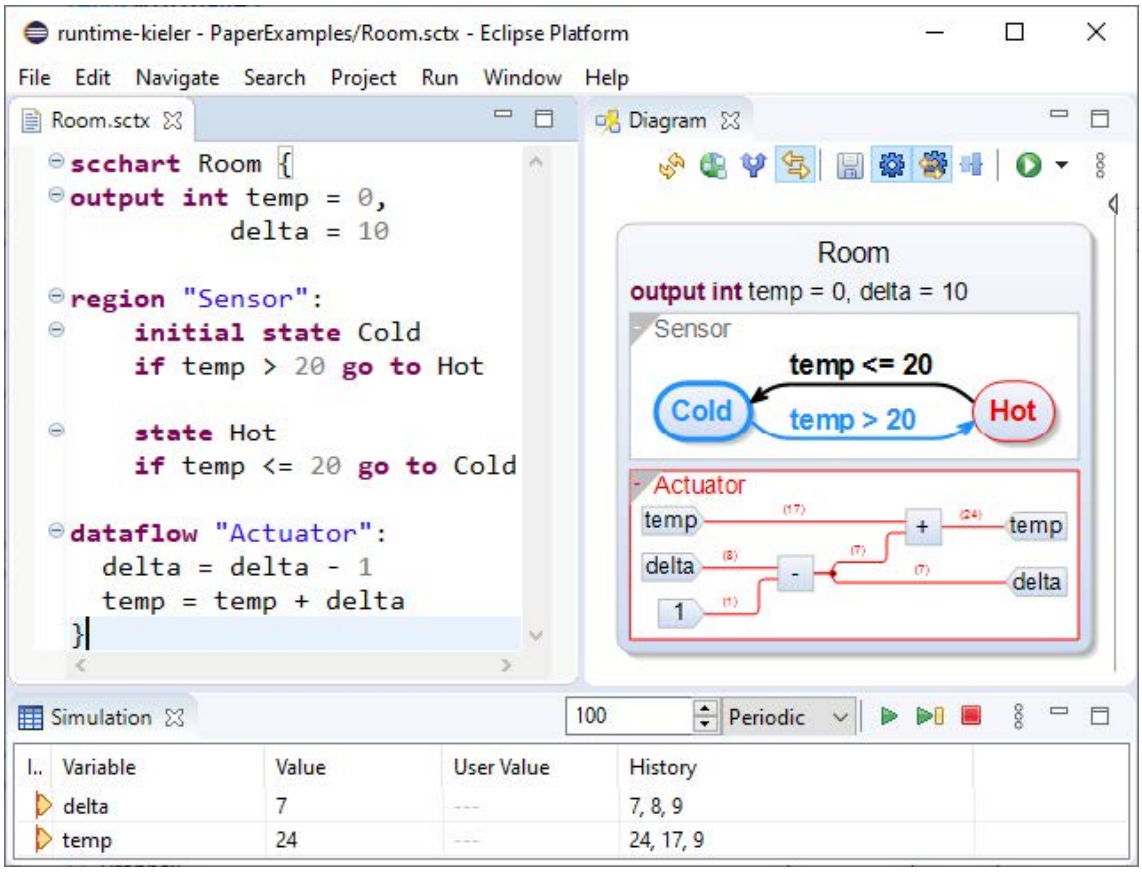

Figure 7.1: A Screenshot of the KIELER SCChart tool suit containing a SCChart with control-flow and data-flow regions [3]

Appendix A

Code of the SCCharts Editor

```
id info.scce.cinco.product.scchart.mglid
stylePath "model/SCChart.style"

@primeviewer

@mcam("check")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.RootStateCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.RegionCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.SuperStateCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.DeclarationCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.SuspensionCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.ActionCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.StateCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.
    SCChartRefereceCheck")
@mcam_checkmodule("info.scce.cinco.product.scchart.checks.TransitionCheck")

@generatable("info.scce.cinco.product.scchart.generator.CodeGenerator", "
    Code")

@event("info.scce.cinco.product.scchart.events.SCChartEvent")
graphModel SCChart {
  diagramExtension "scchart"
  containableElements(RootState[1,1])
  attr EString as name
}

@palette
@disable(delete)
@event("info.scce.cinco.product.scchart.events.RootStateEvent")
container RootState {
  style rootStateStyle("${empty label ? name : label}")
  containableElements(Region, Declaration, Suspension, Action)
```

```
32    outgoingEdges(*)//without, no edge properties
      attr EString as name := "<set name>"
34    attr EString as label
      @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
        GeneratorOutputProvider")
36    attr EString as generatorOutput := "Identity Diagram"
}
38
/* Regions */
40 @palette("Region")
@disable(resize,move)
42 @event("info.scce.cinco.product.scchart.events.RegionEvent")
container Region {
44    style regionStyle("${empty label ? (empty name ? '' : name) : label}")
      containableElements(SuperState, InitialSuperState[0,1], FinalSuperState,
        InitialFinalState[0,1], SimpleState, InitialState[0,1], FinalState,
        InitialFinalSuperState[0,1], Connector, SCChartReferece,
        InitialSCChartReferece, FinalSCChartReferece)
46    attr EString as name
      attr EString as label
48    @propertiesViewHidden
      attr EString as uuid
50 }

52 @palette
@disable(resize,move)
54 container RegionRef {
      style regionStyle("")
56    containableElements(RootState)
}
58
/* SuperStates */
60 @palette("SuperStates")
@event("info.scce.cinco.product.scchart.events.SuperStateEvent")
62 container SuperState {
      style superStateStyle("${empty label ? name : label}")
64    containableElements(Region, Declaration, Action, Suspension)
      incomingEdges(Transition, TerminationTransition, StrongAbortTransition,
        DeferredTransition, TerminationDeferredTransition,
        StrongAbortDeferredTransition, HistoryTransition,
        TerminationHistoryTransition, StrongAbortHistoryTransition,
        DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
        TerminationDeferredHistoryTransition)
66    outgoingEdges(Transition, TerminationTransition, StrongAbortTransition,
        DeferredTransition, TerminationDeferredTransition,
        StrongAbortDeferredTransition, HistoryTransition,
        TerminationHistoryTransition, StrongAbortHistoryTransition,
```

```
    DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
    TerminationDeferredHistoryTransition)
  attr EString as name := "<set name>"
  attr EString as label
}

@palette("SuperStates")
container InitialSuperState extends SuperState {
  style initialSuperStateStyle("${empty label ? name : label}")
}

@palette("SuperStates")
container FinalSuperState extends SuperState {
  style finalSuperStateStyle("${empty label ? name : label}")
}

@palette("SuperStates")
container InitialFinalSuperState extends SuperState {
  style initialFinalSuperStateStyle("${empty label ? name : label}")
}

/* Declarations */
@palette("Declaration")
@disable(move, resize)
@event("info.scce.cinco.product.scchart.events.DeclarationEvent")
node Declaration {
  style declarationStyle("${(empty inputOutput || inputOutput == ' ' ? '' :
    (inputOutput.concat(' '))).concat(empty constant || !constant ? '' : '
    const ').concat(empty signal || !signal ? '' : 'signal ').concat(empty
    declarationType || declarationType == ' ' ? '' : declarationType)}", "${(
    empty name || name == ' ' ? '' : (name.concat(empty assignment ||
    assignment == ' ' ? '' : (' = '.concat(assignment)))))}")
  @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
    DeclarationInputOutputProvider")
  attr EString as inputOutput := ' '
  attr EBoolean as constant
  attr EBoolean as signal
  @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
    DeclarationTypeProvider")
  attr EString as declarationType := ' '
  attr EString as name := '<set name>'
  attr EString as assignment
  @propertiesViewHidden
  attr EString as uuid
}

@palette("Declaration")
@disable(move, resize)
```

```
@event("info.scce.cinco.product.scchart.events.SuspensionEvent")
node Suspension {
  style declarationStyle("${(suspensionType)}", "${(suspensionType == '<set
    suspension>' ? ' ' : empty condition ? ' ' : condition)}")
  @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
    SuspensionTypeProvider")
  attr EString as suspensionType := '<set suspension>'
  attr EString as condition
  @propertiesViewHidden
  attr EString as uuid
}

@palette("Declaration")
@disable(move,resize)
@event("info.scce.cinco.product.scchart.events.ActionEvent")
node Action {
  style declarationStyle("${actionType}", "${(actionType == '<set action>' ?
     ' ' : ((empty condition ? '' : condition).concat(empty effect ? '' : ' /
     '.concat(effect))))}")
  @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
    ActionTypeProvider")
  attr EString as actionType := '<set action>'
  attr EString as condition
  attr EString as effect
  @propertiesViewHidden
  attr EString as uuid
}

/* States */
@disable(resize)
@palette("States")
node SimpleState {
  style simpleStateStyle("${empty label ? name : label}")
  incomingEdges(Transition,TerminationTransition,StrongAbortTransition)
  outgoingEdges(Transition,DeferredTransition,HistoryTransition,
    DeferredHistoryTransition)
  attr EString as name := "<set name>"
  attr EString as label
}

@disable(resize)
@palette("States")
node InitialState extends SimpleState {
  style initialStateStyle("${empty label ? name : label}")
}

@disable(resize)
@palette("States")
```

```
148 node FinalState extends SimpleState {
      style finalStateStyle("${empty label ? name : label}")
150 }

152 @disable(resize)
    @palette("States")
154 node InitialFinalState extends SimpleState {
      style initialFinalStateStyle("${empty label ? name : label}")
156 }

158 @disable(resize)
    @palette("States")
160 node Connector extends SimpleState {
      style connectorStyle
162 }

164 /* SCChart References */
    @event("info.scce.cinco.product.scchart.events.SCChartRefereceEvent")
166 container SCChartReferece {
      style sCChartRefereceStyle("${empty label ? name : label} @ ${reference.
        name} (${assignments})")
168   containableElements(RegionRef)
      incomingEdges(Transition, TerminationTransition, StrongAbortTransition,
        DeferredTransition, TerminationDeferredTransition,
        StrongAbortDeferredTransition, HistoryTransition,
        TerminationHistoryTransition, StrongAbortHistoryTransition,
        DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
        TerminationDeferredHistoryTransition)
170   outgoingEdges(Transition, TerminationTransition, StrongAbortTransition,
        DeferredTransition, TerminationDeferredTransition,
        StrongAbortDeferredTransition, HistoryTransition,
        TerminationHistoryTransition, StrongAbortHistoryTransition,
        DeferredHistoryTransition, StrongAbortDeferredHistoryTransition,
        TerminationDeferredHistoryTransition)
      @pvFileExtension("scchart")
172   prime this::SCChart as reference
      @readOnly
174   attr EString as inputsOutputsOfRef := ""
      attr EString as name := "<set name>"
176   attr EString as assignments := ""
      attr EString as label
178 }

180 container InitialSCChartReferece extends SCChartReferece {
      style initialSCChartRefereceStyle("${empty label ? name : label} @ ${
        reference.name} (${assignments})")
182   @pvFileExtension("scchart")
      prime this::SCChart as reference
```

```
184 }

186 container FinalSCChartReferece extends SCChartReferece {
      style finalSCChartRefereceStyle("${empty label ? name : label} @ ${
        reference.name} (${assignments})")
188   @pvFileExtension("scchart")
      prime this::SCChart as reference
190 }

192 container InitialFinalSCChartReferece extends SCChartReferece {
      style initialFinalRefereceStyle("${empty label ? name : label} @ ${
        reference.name} (${assignments})")
194   @pvFileExtension("scchart")
      prime this::SCChart as reference
196 }

198 /*Transitions */
    edge AbstractTransition {
200   style abstractTransitionStyle
      @possibleValuesProvider("info.scce.cinco.product.scchart.provider.
        AbstractTransitionPriorityProvider")
202   attr EString as priority := '1'
    }
204

    edge Transition extends AbstractTransition {
206   style transitionStyle("${priority.concat(': ').concat(condition == '<No
        condition>' || empty condition ? '' : (empty count_delay ? '' :
        count_delay.concat(' ')).concat(condition)).concat(effect == '<No effect
        >' ? '' : ' / '.concat(effect))}")
      attr EString as condition := '<No condition>'
208   attr EString as effect := '<No effect>'
      attr EString as count_delay
210   attr EBoolean as immediate
    }
212

    edge TerminationTransition extends AbstractTransition {
214   style terminationTransitionStyle("${priority.concat(': ').concat(condition
        == '<No condition>' || empty condition ? '' : (empty count_delay ? '' :
        count_delay.concat(' ')).concat(condition)).concat(effect == '<No effect
        >' ? '' : ' / '.concat(effect))}")
      attr EString as condition := '<No condition>'
216   attr EString as effect := '<No effect>'
      attr EString as count_delay
218   attr EBoolean as immediate
    }
220

    edge StrongAbortTransition extends AbstractTransition {
```

```
222   style strongAbortTransitionStyle("${priority.concat(': ').concat(condition
        == '<No condition>' || empty condition ? '' : (empty count_delay ? '' :
      count_delay.concat(' ')).concat(condition)).concat(effect == '<No effect
      >' ? '' : ' / '.concat(effect))}")
      attr EString as condition := '<No condition>'
224   attr EString as effect := '<No effect>'
      attr EString as count_delay
226   attr EBoolean as immediate
}
228
edge DeferredTransition extends AbstractTransition {
230   style deferredTransitionStyle("${priority.concat(': ').concat(condition ==
        '<No condition>' || empty condition ? '' : (empty count_delay ? '' :
      count_delay.concat(' ')).concat(condition)).concat(effect == '<No effect
      >' ? '' : ' / '.concat(effect))}")
      attr EString as condition := '<No condition>'
232   attr EString as effect := '<No effect>'
      attr EString as count_delay
234   attr EBoolean as immediate
}
236
edge HistoryTransition extends AbstractTransition {
238   style historyTransitionStyle("${priority.concat(': ').concat(condition ==
      '<No condition>' || empty condition ? '' : (empty count_delay ? '' :
      count_delay.concat(' ')).concat(condition)).concat(effect == '<No effect
      >' ? '' : ' / '.concat(effect))}","${deepHistory ? '*' : ' '}")
      attr EString as condition := '<No condition>'
240   attr EString as effect := '<No effect>'
      attr EString as count_delay
242   attr EBoolean as deepHistory
      attr EBoolean as immediate
244 }

246 edge TerminationDeferredTransition extends AbstractTransition {
      style terminationDeferredTransitionStyle("${priority.concat(': ').concat(
        condition == '<No condition>' || empty condition ? '' : (empty
        count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
        effect == '<No effect>' ? '' : ' / '.concat(effect))}")
248   attr EString as condition := '<No condition>'
      attr EString as effect := '<No effect>'
250   attr EString as count_delay
      attr EBoolean as immediate
252 }

254 edge StrongAbortDeferredTransition extends AbstractTransition {
      style strongAbortDeferredTransitionStyle("${priority.concat(': ').concat(
        condition == '<No condition>' || empty condition ? '' : (empty
```

```
    count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
    effect == '<No effect>' ? '' : ' / '.concat(effect))}")
  attr EString as condition := '<No condition>'
  attr EString as effect := '<No effect>'
  attr EString as count_delay
  attr EBoolean as immediate
}

edge TerminationHistoryTransition extends AbstractTransition {
  style terminationHistoryTransitionStyle("${priority.concat(': ').concat(
    condition == '<No condition>' || empty condition ? '' : (empty
    count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
    effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
    '*' : ' '}")
  attr EString as condition := '<No condition>'
  attr EString as effect := '<No effect>'
  attr EString as count_delay
  attr EBoolean as deepHistory
  attr EBoolean as immediate
}

edge StrongAbortHistoryTransition extends AbstractTransition {
  style strongAbortHistoryTransitionStyle("${priority.concat(': ').concat(
    condition == '<No condition>' || empty condition ? '' : (empty
    count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
    effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
    '*' : ' '}")
  attr EString as condition := '<No condition>'
  attr EString as effect := '<No effect>'
  attr EString as count_delay
  attr EBoolean as deepHistory
  attr EBoolean as immediate
}

edge DeferredHistoryTransition extends AbstractTransition {
  style deferredHistoryTransitionStyle("${priority.concat(': ').concat(
    condition == '<No condition>' || empty condition ? '' : (empty
    count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
    effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
    '*' : ' '}")
  attr EString as condition := '<No condition>'
  attr EString as effect := '<No effect>'
  attr EString as count_delay
  attr EBoolean as deepHistory
  attr EBoolean as immediate
}

edge StrongAbortDeferredHistoryTransition extends AbstractTransition {
```

```
290   style strongAbortDeferredHistoryTransitionStyle("${priority.concat(': ').
        concat(condition == '<No condition>' || empty condition ? '' : (empty
        count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
        effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
        '*' : ' '}")
      attr EString as condition := '<No condition>'
292   attr EString as effect := '<No effect>'
      attr EString as count_delay
294   attr EBoolean as deepHistory
      attr EBoolean as immediate
296 }

298 edge TerminationDeferredHistoryTransition extends AbstractTransition {
      style terminationDeferredHistoryTransitionStyle("${priority.concat(': ').
        concat(condition == '<No condition>' || empty condition ? '' : (empty
        count_delay ? '' : count_delay.concat(' ')).concat(condition)).concat(
        effect == '<No effect>' ? '' : ' / '.concat(effect))}","${deepHistory ?
        '*' : ' '}")
300   attr EString as condition := '<No condition>'
      attr EString as effect := '<No effect>'
302   attr EString as count_delay
      attr EBoolean as deepHistory
304   attr EBoolean as immediate
    }
```

Listing A.1: The SCChart MGL

```
1 appearance default {
    lineWidth 2
3   background (198,226,255) //LightSlateGrey
    foreground (119,136,153) //LightSlateGrey
5 }

7 appearance declarationAppearance {
    lineWidth 0
9   background (198,226,255) //SlateGrey2
    foreground (198,226,255) //LightSlateGrey
11 }

13 appearance declarationText {
    font ("Sans",BOLD,8)
15 }

17 appearance textHighlight {
    font ("Sans",BOLD,8)
19  foreground (166, 0, 99)
    background (166, 0, 99)
21 }
```

```
23 appearance finalStateOuterCircle {
     lineWidth 2
25   foreground (119,136,153) //LightSlateGrey
   }
27
   appearance initialFinalStateOuterCircle {
29   lineWidth 3
   }
31
   appearance superStateAppearance {
33   background (198,226,255) //SlateGrey2
     foreground (119,136,153) //LightSlateGrey
35 }

37 appearance initialSuperStateAppearance {
     background (198,226,255) //SlateGrey2
39   foreground (0,0,0) //Black
     lineWidth 3
41 }

43 appearance regionAppearance {
     foreground (119,136,153) //LightSlateGrey
45 }

47 appearance initialStateAppearance {
     background (198,226,255) //LightSlateGrey
49   lineWidth 4
   }
51
   appearance connector {
53   background (0,0,0) //Black
   }
55
   appearance sCChartRefereceAppearance {
57   background (255,207,241) //purple
   }
59
   appearance initialSCChartRefereceAppearance {
61   background (255,207,241) //purple
     lineWidth 4
63 }

65 appearance transitionAppearance {
     lineStyle SOLID
67   lineWidth 2
   }
69
```

```
appearance greenTriangle {
  background (50,205,50)
}

appearance redCircle {
  background (255,0,0) //Red
}

appearance historyCircle {
  background (0,0,0)
}

appearance historyH {
  font ("Sans",10)
  foreground (255,255,255)
}

appearance textFont {
  font ("Sans",10)
}

/*rootstate style */
nodeStyle rootStateStyle(1) {
  roundedRectangle {
    appearance superStateAppearance
    size (800,1000)
    corner (15,15)
    text {
      appearance textFont
      position (CENTER,TOP 3)
      value "%1$s"
    }
  }
}

/*region style */
nodeStyle regionStyle(1) {
  rectangle {
    appearance regionAppearance
    text {
      appearance textFont
      position (LEFT 3,TOP 3)
      value "%1$s"
    }
  }
}

/*superstate style */
```

```
nodeStyle superStateStyle(1) {
  roundedRectangle {
    appearance superStateAppearance
    size (140,80)
    corner (15,15)
    text {
      appearance textFont
      position (CENTER,TOP 3)
      value "%1$s"
    }
  }
}

nodeStyle initialSuperStateStyle(1) {
  roundedRectangle {
    appearance initialSuperStateAppearance
    size (140,80)
    corner (15,15)
    text {
      appearance textFont
      position (CENTER,TOP 3)
      value "%1$s"
    }
  }
}

nodeStyle finalSuperStateStyle(1) {
  roundedRectangle {
    appearance regionAppearance
    size (140,80)
    corner (15,15)
    roundedRectangle {
      appearance superStateAppearance
      position (CENTER,MIDDLE)
      size (134,74)
      corner (15,15)
      text {
        appearance textFont
        position (CENTER,TOP 3)
        value "%1$s"
      }
    }
  }
}

nodeStyle initialFinalSuperStateStyle(1) {
  roundedRectangle {
    appearance initialFinalStateOuterCircle
```

```
    size (140,80)
    corner (15,15)
    roundedRectangle {
      appearance superStateAppearance
      position (CENTER,MIDDLE)
      size (134,74)
      corner (15,15)
      text {
        appearance textFont
        position (CENTER,TOP 3)
        value "%1$s"
      }
    }
  }
}

/*declaration style */
nodeStyle declarationStyle(2) {
  rectangle {
    appearance declarationAppearance
    size (140,80)
    text {
      appearance declarationText
      position (LEFT 5,MIDDLE)
      value "%1$s %2$s"
    }
    text {
      appearance textHighlight
      position (LEFT 5,MIDDLE)
      value "%1$s"
    }
  }
}

/*state style */
nodeStyle simpleStateStyle(1) {
  ellipse {
    appearance default
    size (60,60)
    text {
      position (CENTER,MIDDLE)
      value "%s"
    }
  }
}

nodeStyle initialStateStyle(1) {
  ellipse{
```

```
    appearance initialStateAppearance
    size (60,60)
    text {
      position (CENTER,MIDDLE)
      value "%s"
    }
  }
}

nodeStyle finalStateStyle(1) {
  ellipse{
    appearance finalStateOuterCircle
    size (60,60)
    ellipse {
      appearance default
      position (CENTER,MIDDLE)
      size (50,50)
      text {
        position (CENTER,MIDDLE)
        value "%s"
      }
    }
  }
}

nodeStyle initialFinalStateStyle(1) {
  ellipse {
    appearance initialFinalStateOuterCircle
    size (60,60)
    ellipse {
      appearance default
      position (CENTER,MIDDLE)
      size(50,50)
      text {
        position (CENTER,MIDDLE)
        value "%s"
      }
    }
  }
}

nodeStyle connectorStyle {
  ellipse {
    appearance connector
    size (10,10)
  }
}

```

```
nodeStyle sCChartRefereceStyle(1) {
  roundedRectangle {
    appearance sCChartRefereceAppearance
    size (140,80)
    corner (15,15)
    text {
      position (CENTER,TOP)
      value "%s"
    }
  }
}

nodeStyle initialSCChartRefereceStyle(1) {
  roundedRectangle {
    appearance initialSCChartRefereceAppearance
    size (140,80)
    corner (15,15)
    text {
      appearance textFont
      position (CENTER,TOP 3)
      value "%1$s"
    }
  }
}

nodeStyle finalSCChartRefereceStyle(1) {
  roundedRectangle {
    appearance regionAppearance
    size (140,80)
    corner (15,15)
    roundedRectangle {
      appearance sCChartRefereceAppearance
      position (CENTER,MIDDLE)
      size (134,74)
      corner (15,15)
      text {
        appearance textFont
        position (CENTER,TOP 3)
        value "%1$s"
      }
    }
  }
}

nodeStyle initialFinalRefereceStyle(1) {
  roundedRectangle {
    appearance initialFinalStateOuterCircle
    size (140,80)
```

```
    corner (15,15)
    roundedRectangle {
      appearance sCChartRefereceAppearance
      position (CENTER,MIDDLE)
      size (134,74)
      corner (15,15)
      text {
        appearance textFont
        position (CENTER,TOP 3)
        value "%1$s"
      }
    }
  }
}

/*transitionStyles */
edgeStyle abstractTransitionStyle {
  appearance transitionAppearance
}

edgeStyle transitionStyle(1) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    TransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    ARROW
    appearance transitionAppearance
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
}

edgeStyle terminationTransitionStyle(1) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    TerminationTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    ARROW
    appearance transitionAppearance
  }
  decorator {
```

```
    location (0)
    polygon {
      appearance greenTriangle
      points [(0,0)(0,7)(15,0)(0,-7)]
    }
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
}

edgeStyle strongAbortTransitionStyle(1) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    StrongAbortTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    ARROW
    appearance transitionAppearance
  }
  decorator {
    location (0)
    CIRCLE
    appearance redCircle
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
}

edgeStyle deferredTransitionStyle(1) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    DeferredTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    polyline{
      appearance transitionAppearance
      points [(-22,-5)(-12,0)(-22,5)]
    }
```

```
  }
  decorator {
    location (1.0)
    CIRCLE
    appearance redCircle
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
}

edgeStyle historyTransitionStyle(2) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    HistoryTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    ellipse {
      appearance historyCircle
      size(15,15)
    }
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
  decorator {
    location (1.0)
    text {
      appearance historyH
      value "%2$s"
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance historyH
      points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
    }
  }
  decorator {
```

```
449     location (1.0)
        polyline {
451       appearance transitionAppearance
          points [(-25,-5)(-15,0)(-25,5)]
453     }
    }
455 }

457 edgeStyle terminationDeferredTransitionStyle(1) {
    appearanceProvider("info.scce.cinco.product.scchart.appearance.
      TerminationDeferredTransitionAppearance")
459   appearance transitionAppearance
    decorator {
461     location (1.0)
        polyline {
463       appearance transitionAppearance
          points [(-22,-5)(-12,0)(-22,5)]
465     }
    }
467   decorator {
        location (1.0)
469     CIRCLE
        appearance redCircle
471   }
    decorator {
473     location (0.5)
        movable
475     text {
          value "%s"
477     }
    }
479   decorator {
        location (0)
481     polygon {
          appearance greenTriangle
483       points [(0,0)(0,7)(15,0)(0,-7)]
        }
485   }
}
487

edgeStyle strongAbortDeferredTransitionStyle(1) {
489   appearanceProvider("info.scce.cinco.product.scchart.appearance.
      StrongAbortDeferredTransitionAppearance")
    appearance transitionAppearance
491   decorator {
        location (1.0)
493     polyline {
          appearance transitionAppearance
```

```
495       points [(-22,-5)(-12,0)(-22,5)]
      }
497   }
    decorator {
499     location (1.0)
      CIRCLE
501     appearance redCircle
    }
503   decorator {
      location (0.5)
505     movable
      text {
507       value "%s"
      }
509   }
    decorator {
511     location (0)
      CIRCLE
513     appearance redCircle
    }
515 }


517
  edgeStyle terminationHistoryTransitionStyle(2) {
519   appearanceProvider("info.scce.cinco.product.scchart.appearance.
        TerminationHistoryTransitionAppearance")
    appearance transitionAppearance
521   decorator {
      location (1.0)
523     ellipse {
        appearance historyCircle
525       size(15,15)
      }
527   }
    decorator {
529     location (0.5)
      movable
531     text {
        value "%s"
533     }
    }
535   decorator {
      location (1.0)
537     text {
        appearance historyH
539       value "%2$s"
      }
541   }
```

```
  decorator {
    location (1.0)
    polyline {
      appearance historyH
      points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance transitionAppearance
      points [(-25,-5)(-15,0)(-25,5)]
    }
  }
  decorator {
    location (0)
    polygon {
      appearance greenTriangle
      points [(0,0)(0,7)(15,0)(0,-7)]
    }
  }
}

edgeStyle strongAbortHistoryTransitionStyle(2) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    StrongAbortHistoryTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    ellipse {
      appearance historyCircle
      size(15,15)
    }
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
  decorator {
    location (1.0)
    text {
      appearance historyH
      value "%2$s"
    }
  }
```

```
589   decorator {
        location (1.0)
591     polyline{
          appearance historyH
593       points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
        }
595   }
      decorator {
597     location (1.0)
        polyline {
599       appearance transitionAppearance
          points [(-25,-5)(-15,0)(-25,5)]
601     }
      }
603   decorator {
        location (0)
605     CIRCLE
        appearance redCircle
607   }
}
609
edgeStyle deferredHistoryTransitionStyle(2) {
611   appearanceProvider("info.scce.cinco.product.scchart.appearance.
        DeferredHistoryTransitionAppearance")
      appearance transitionAppearance
613   decorator {
        location (1.0)
615     CIRCLE
        appearance redCircle
617   }
      decorator {
619     location (1.0)
        ellipse {
621       appearance historyCircle
          size (15,15)
623     }
      }
625   decorator {
        location (0.5)
627     movable
        text {
629       value "%s"
        }
631   }
      decorator {
633     location (1.0)
        text {
635       appearance historyH
```

```
      value "%2$s"
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance historyH
      points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance transitionAppearance
      points [(-25,-5)(-15,0)(-25,5)]
    }
  }
}

edgeStyle strongAbortDeferredHistoryTransitionStyle(2) {
  appearanceProvider("info.scce.cinco.product.scchart.appearance.
    StrongAbortDeferredHistoryTransitionAppearance")
  appearance transitionAppearance
  decorator {
    location (1.0)
    CIRCLE
    appearance redCircle
  }
  decorator {
    location (1.0)
    ellipse {
      appearance historyCircle
      size(15,15)
    }
  }
  decorator {
    location (0.5)
    movable
    text {
      value "%s"
    }
  }
  decorator {
    location (1.0)
    text {
      appearance historyH
      value "%2$s"
    }
```

```
683   }
      decorator {
685     location (1.0)
        polyline {
687       appearance historyH
          points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
689     }
      }
691   decorator {
        location (1.0)
693     polyline {
          appearance transitionAppearance
695       points [(-25,-5)(-15,0)(-25,5)]
        }
697   }
      decorator {
699     location (0)
        CIRCLE
701     appearance redCircle
      }
703 }

705 edgeStyle terminationDeferredHistoryTransitionStyle(2) {
      appearanceProvider("info.scce.cinco.product.scchart.appearance.
        TerminationDeferredHistoryTransitionAppearance")
707   appearance transitionAppearance
      decorator {
709     location (1.0)
        CIRCLE
711     appearance redCircle
      }
713   decorator {
        location (1.0)
715     ellipse {
          appearance historyCircle
717       size (15,15)
        }
719   }
      decorator {
721     location (0.5)
        movable
723     text {
          value "%s"
725     }
      }
727   decorator {
        location (1.0)
729     text {
```

```
      appearance historyH
      value "%2$s"
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance historyH
      points [(-5,-5)(-5,5)(-5,0)(-10,0)(-10,-5)(-10,5)]
    }
  }
  decorator {
    location (1.0)
    polyline {
      appearance transitionAppearance
      points [(-25,-5)(-15,0)(-25,5)]
    }
  }
  decorator {
    location (0)
    polygon {
      appearance greenTriangle
      points [(0,0)(0,7)(15,0)(0,-7)]
    }
  }
}
```

Listing A.2: The SCChart SGL

```
// Generated by de.jabc.cinco.meta.plugin.event.generator.template.
    EventUserClassTemplate
package info.scce.cinco.product.scchart.events

import graphmodel.Direction
import graphmodel.ModelElementContainer
import info.scce.cinco.product.scchart.mglid.scchart.Action
import java.util.UUID
import info.scce.cinco.product.scchart.mglid.scchart.SuperState

/*
 * About this class:
 * - This is a default implementation for info.scce.cinco.product.scchart.
    mglid.scchart.event.ActionEvent.
 * - This class was generated, because you added an "@event" annotation to
 *   Node "Action" in "SCChart.mgl".
 * - This file will not be overwritten on future generation processes.
 *
 * Available event methods:
```

```
 * - preCreate(Class<? extends Action> elementClass, ModelElementContainer
    container, int x, int y, int width, int height)
 * - preDelete(Action element)
 * - preMove(Action element, ModelElementContainer newContainer, int newX,
    int newY)
 * - preResize(Action element, int newWidth, int newHeight, int newX, int
    newY, Direction direction)
 * - postAttributeChange(Action element, String attribute, Object oldValue)
 * - postCreate(Action element)
 * - postDelete(Action element)
 * - postDoubleClick(Action element)
 * - postMove(Action element, ModelElementContainer oldContainer, int oldX,
    int oldY)
 * - postResize(Action element, int oldWidth, int oldHeight, int oldX, int
    oldY, Direction direction)
 * - postSelect(Action element)
 */
final class ActionEvent extends info.scce.cinco.product.scchart.mglid.
    scchart.event.ActionEvent {

  override postCreate(Action element) {
    element.uuid = UUID.randomUUID.toString
    var boolean continue = false
    if (element.rootElement.rootStates.head.actions !== null) {
      for (action : element.rootElement.rootStates.head.actions) {
        if (action.uuid == element.uuid) {
          var int declarationCount = 0
          if (element.rootElement.rootStates.head.declarations !== null) {
            declarationCount = element.rootElement.getRootStates.head.
    declarations.size
          }
          if (element.rootElement.rootStates.head.suspensions !== null) {
            declarationCount += element.rootElement.rootStates.head.
    suspensions.size
          }
          for (var i = 0; i < element.rootElement.rootStates.head.actions.
    size; i++) {
            element.rootElement.rootStates.head.actions.get(i).x = 10
            element.rootElement.rootStates.head.actions.get(i).y = 30 + 13 *
     declarationCount + 13 * i
            element.rootElement.rootStates.head.actions.get(i).width =
    element.rootElement.rootStates.head.
              width - 20
            element.rootElement.rootStates.head.actions.get(i).height = 13
          }
          declarationCount += element.rootElement.getRootStates.head.actions
    .size
          if (element.rootElement.rootStates.head.regions !== null) {
```

```
            for (region : element.rootElement.rootStates.head.regions) {
              if (region.y < 30 + declarationCount * 13) {
                region.y = region.y + 13
                region.height = region.height - 13
              }
            }
          }
          continue = true
        }
      }
    }
    if (!continue) {
      for (region : element.rootElement.rootStates.head.regions) {
        if (region.superStates !== null) {
          for (superState : region.superStates) {
            postCreateAction(superState, element)
          }
        }
      }
    }
  }

  def postCreateAction(SuperState superState, Action action) {
    var boolean continue = false
    if (superState.actions !== null) {
      for (actionList : superState.actions) {
        if (actionList.uuid == action.uuid) {
          var int declarationCount = 0
          if (superState.declarations !== null) {
            declarationCount = superState.declarations.size
          }
          if (superState.suspensions !== null) {
            declarationCount += superState.suspensions.size
          }
          for (var i = 0; i < superState.actions.size; i++) {
            superState.actions.get(i).x = 10
            superState.actions.get(i).y = 30 + 13 * declarationCount + 13 *
    i
            superState.actions.get(i).width = superState.width - 20
            superState.actions.get(i).height = 13
          }
          declarationCount += superState.actions.size
          if (superState.regions !== null) {
            for (region : superState.regions) {
              if (region.y < 30 + declarationCount * 13) {
                region.y = region.y + 13
                region.height = region.height - 13
              }
```

```
            }
          }
          continue = true
        }
      }
    }
    if (!continue && superState.regions !== null) {
      for (region : superState.regions) {
        if (region.superStates !== null) {
          for (superStateList : region.superStates) {
            postCreateAction(superStateList, action)
          }
        }
      }
    }
  }

  override postDelete(Action element) {
    var boolean continue = false
    if (element.rootElement.rootStates.head.actions !== null) {
      for (var j = 0; j < element.rootElement.rootStates.head.actions.size;
    j++) {
        if (element.rootElement.rootStates.head.actions.get(j).uuid ==
    element.uuid) {
          var int declarationCount = 0
          if (element.rootElement.rootStates.head.declarations !== null) {
            declarationCount = element.rootElement.getRootStates.head.
    declarations.size
          }
          if (element.rootElement.rootStates.head.suspensions !== null) {
            declarationCount += element.rootElement.rootStates.head.
    suspensions.size
          }
          for (var i = j; i < element.rootElement.rootStates.head.actions.
    size; i++) {
            element.rootElement.rootStates.head.actions.get(i).y = element.
    rootElement.rootStates.head.
              actions.get(i).y - 13
          }
          declarationCount += element.rootElement.getRootStates.head.actions
    .size
          if (element.rootElement.rootStates.head.regions !== null) {
            for (region : element.rootElement.rootStates.head.regions) {
              if (region.y == 33 + declarationCount * 13) {
                region.y = region.y - 13
                region.height = region.height + 13
              }
            }
```

```
          }
143         continue = true
        }
145     }
    }
147   if (!continue) {
      for (region : element.rootElement.rootStates.head.regions) {
149     if (region.superStates !== null) {
          for (superState : region.superStates) {
151         postDeleteAction(superState, element)
          }
153     }
      }
155   }
    // Set up your post delete Runnable here.
157 // This will be executed pre delete.
    return [
159   // This is your post delete Runnable.
      // This will be executed post delete.
161 ]
}
163
def postDeleteAction(SuperState superState, Action action) {
165   var boolean continue = false
    if (superState.actions !== null) {
167   for (var j = 0; j < superState.actions.size; j++) {
        if (superState.actions.get(j).uuid == action.uuid) {
169       var int declarationCount = 0
          if (superState.declarations !== null) {
171         declarationCount = superState.declarations.size
          }
173       if (superState.suspensions !== null) {
            declarationCount += superState.suspensions.size
175       }
          for (var i = j; i < superState.actions.size; i++) {
177         superState.actions.get(i).y = superState.actions.get(i).y - 13
          }
179       declarationCount += superState.actions.size
          if (superState.regions !== null) {
181         for (region : superState.regions) {
              if (region.y == 33 + declarationCount * 13) {
183             region.y = region.y - 13
                region.height = region.height + 13
185           }
            }
187       }
          continue = true
189     }
```

```
      }
    }
    if (!continue && superState.regions !== null) {
      for (region : superState.regions) {
        if (region.superStates !== null) {
          for (superStateList : region.superStates) {
            postDeleteAction(superStateList, action)
          }
        }
      }
    }
  }
}
```

Listing A.3: The ActionEvent Xtend class

List of Figures

Bibliography

[1] André, C.: *Semantics of SyncCharts.* Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[2] Brambilla, Marco, Jordi Cabot and Manuel Wimmer: *Model-Driven Software Engineering in Practice: Second Edition.* Synthesis Lectures on Software Engineering, 3(1):1–207, 2017.

[3] Grosse, Daniel (editor): *Proceedings of the 2020 Forum on Specification & Design Languages (FDL): Kiel (Germany), 15-17 September 2020.* IEEE, Piscataway, NJ, 2020.

[4] Hanxleden, Reinhard von, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer and Owen O'Brien: *SCCharts: sequentially constructive statecharts for safety-critical applications.* ACM SIGPLAN Notices, 49(6):372–383, 2014.

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

[6] Lybecait, Michael, Dawid Kopetzki, Philip Zweihoff, Annika Fuhge, Stefan Naujokat and Bernhard Steffen: *A Tutorial Introduction to Graphical Modeling and Metamodeling with CINCO.* pages 519–538. Springer, Cham, 2018.

[7] Motika, Christian, Reinhard von Hanxleden and Florence Maraninchi: *SCCharts - Language and Interactive Incremental Compilation.* 2017.

[8] Naujokat, Stefan, Michael Lybecait, Dawid Kopetzki and Bernhard Steffen: *CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools.* International Journal on Software Tools for Technology Transfer, 20(3):327–354, 2018.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kentlich gemacht habe.

Dortmund, den October 12, 2023

Kristopher Kettler