Christian-Albrechts-Universität zu Kiel

Master Thesis

A SyncCharts Editor based on YAKINDU SCT

cand. inform. Wahbi Haribi

March 14, 2013

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

SyncCharts are a synchronous Statecharts dialect, a graphical formalism to describe complex reactive systems. In order to allow modeling with SyncCharts, the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) framework offers an interactive editor. However, it is based on Graphical Modeling Framework (GMF) and its generation needs a lot of modified and added templates. YAKINDU Statechart Tools (SCT) is a modeling environment based on the concept of Statecharts, which is designed for extensibility. Since the representation of SyncCharts is similar to Statecharts, the YAKINDU SCT Editor can be extended to implement a SyncCharts Editor in order to minimize the effort for maintenance and to allow KIELER developers to focus on pragmatics or semantics.

In this master thesis an approach is presented that allows the implementation of a generic SyncCharts Editor based on YAKINDU. It is shown that the presented approach can be reused for similar editors like an SCCharts Editor with minimal effort. We evaluate the approach by comparing the maintenance effort with the standard GMF implementation.

Contents

List of Figures

Abbreviations

KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEL	Kiel Integrated Environment for Layout
KIML	KIELER Infrastructure for Meta Layout
KLay	KIELER Layout Algorithms
KAOM	KIELER Actor Oriented Modeling
KARMA	KIELER Advanced Rendering of Model Appearance
KEG	KIELER Editor for Graphs
KEX	KIELER Example Management
FSM	Finite State Machines
GUI	Graphical User Interface
SCADE	Safety Critical Application Development Environment
SCT	Statechart Tools
RCP	Rich Client Platform
IDE	Integrated Development Environment
SWT	Standard Widget Toolkit
OS	Operating System
HTML	Hypertext Markup Language
OLE	Object Linking and Embedding
COM	Component Object Model
UI	User Interface
OSGi	Open Services Gateway initiative
JVM	Java Virtual Machine
KIEM	KIELER Execution Manager

EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
GEF	Graphical Editing Framework
XMI	XML Metadata Interchange
XML	Extensible Markup Language
UML	Unified Modeling Language
MVC	Model View Controller
DSL	Domain-Specific Language
KIePto	KIELER leveraging Ptolemy semantics
EBNF	Extended Backus-Naur Form
GME	Generic Modeling Environment
RSA	Rational Software Architect
SM	Stereotype Mechanism
MEM	Metamodel Extension Mechanism
MOF	MetaObject Facility
KIVI	KIELER View Management
SCXML	State Chart eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
ThinKCharts	Thin Kieler SyncCharts
SC	SyncCharts in C
SJ	Synchronous Java
YAKINDU	YAKINDU

1 Introduction

1.1 Reactive Systems

A *reactive system* is a system that continuously reacts to events coming from the environment as depicted in Figure 1.1. A reaction consists of three phases. The system reads input signals from the environment, computes the reaction, and performs the outputs to the environment [4]. A reactive system is often composed of parallel interacting subsystems. Communication and synchronization are essential for these interactions. Safety critical systems often are reactive systems. A typical example

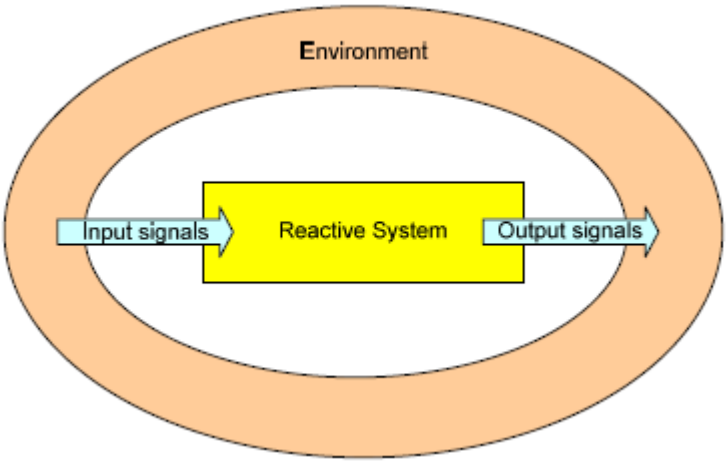


Figure 1.1: A reactive system that reacts to events coming from the environment [4]

is a flight control system of modern airplanes. It allows one plane to take-off, stay in the air, and land. The system could cause injury or loss of human life if it fails or encounters errors. Therefore, such systems have to fulfil the timing constraints and must have a deterministic behavior.

1.1.1 The Synchronous Approach

Modeling and programming reactive systems is not satisfied within traditional programming languages. *Synchronous languages* simplify the describing and analyzing task by allowing the simple expression of deterministic timing predictable concurrency and preemption. Synchronous languages rely on the *synchrony hypothesis*. In the synchrony hypothesis, computations are considered to be atomic and assumed

to be instantaneous [13]. The system reacts to its environment in no time. The notion of physical time is replaced by a notion of clock cycle (logical ticks). Time is divided into ordered sequences of logical ticks. In each tick, the program reads the inputs, computes its reaction and suspends. The *perfect synchrony hypothesis* is defined as below:

Definition 1 (Perfect Synchrony) *A system works in perfect synchrony, if all reactions of the system are executed in zero time. Hence, outputs are generated at the same time, when the inputs are read.*

The communication between the reactive system and its environment is done through signals. Each signal is only present or absent in a tick, never both and each signal is present in a tick only if it is produced in this tick.

1.1.2 The Graphical Approaches

Synchronous languages like Esterel [8] resolve the problems of ensuring the timing constraints and providing a deterministic behavior encountered by describing complex reactive systems. However, they are not user-friendly and with large systems the program becomes unclear, which causes that they are not favored in the industry. *Graphical formalisms* like *Statecharts*, *SyncCharts*, and *SCCharts* are simpler and more interactive. They let users create programs graphically manipulating visual expressions, spatial arrangements of text and graphic symbols [9].

Statecharts [16], conceived by Harel in the 1980s, are a hierarchical model which enables the expression of complex reactive behavior as well as compositional and modular. They are able to express the parallelism, hierarchy and some forms of preemption. However, Statecharts are non-deterministic and do not completely fulfil the synchrony hypothesis [32].

SyncCharts can be seen as a graphical notation for the Esterel language [6] conceived in the mid nineties. They was created by Charles Andre. SyncCharts are based on the synchronous paradigm and devoted to programming reactive systems. They are a Statecharts dialect disposing additionally of a deterministic behavior and based on the synchrony hypothesis as Esterel is.

The semantics of SyncCharts are restrictive, which causes the increase of rejected programs deemed as non-constructive [7]. SCCharts [35] are a sequentially constructive Statecharts dialect. They extend SyncCharts with the advantage of admitting a larger class of programs while preserving deterministic concurrency.

There are several graphical software tools that integrate these graphical formalisms. Among these tools are in particular the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] Framework and the YAKINDU[2] Open Source project. They are presented in the following two sections.

[1] http://www.informatik.uni-kiel.de/rtsys/kieler

[2] http://www.itemis.com/itemis-ag/products/language=en/43503/yakindu-open-source

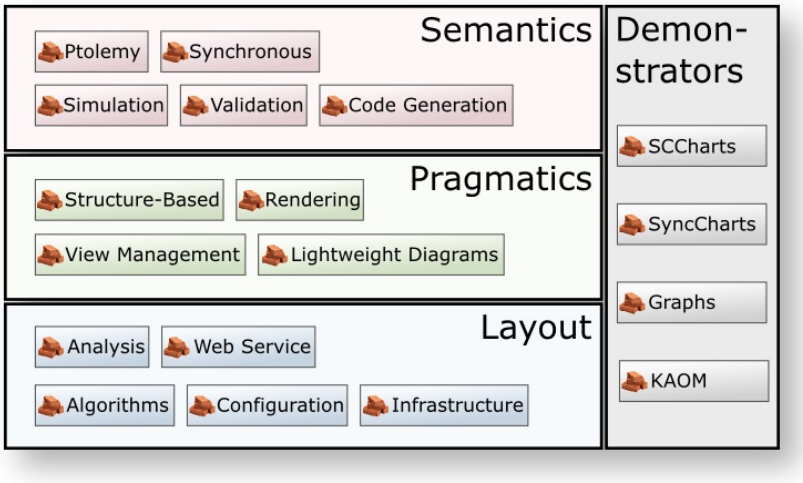


Figure 1.2: The different areas of the KIELER project

1.2 KIELER Framework

KIELER is an academic research project that integrates a variety of modeling languages into the rich client platform Eclipse. KIELER is designed to enhance the graphical model-based design of complex systems by employing automatic layout in all graphical components of the diagrams within the modeling environment. As depicted in Figure 1.2, it is composed of four areas:

- Semantics
- Pragmatics
- Layout
- Demonstrators

The semantics area includes the simulators for modeling languages integrated in the project KIELER and the KIELER Execution Manager (KIEM), which is the infrastructure for simulators, validators, and other facilities [19][20].

The pragmatics area contains a set of practical aspects, which allows the creation and modification of models as well as the synthesis of diverse views.

As mentioned before, KIELER enhances the graphical model-based design of complex systems by employing *automatic layout* [15] in all graphical components of the diagrams. This feature directly allows changing the structure of the model. The automatic layout also permits to expand the important parts of a model and to hide the rest. This technique works immediately without any customisation on each editor integrated into the KIELER framework.

The demonstrators area contains the editors developed in the context of the KIELER framework:

- KIELER SCCharts Editor and SyncCharts Editor, to allow the graphical modeling of reactive systems. They are presented in more detail in Chapter 2.
- KIELER Actor Oriented Modeling (KAOM) [22], to permit the rendering and simulation of actor oriented modeling languages like Ptolemy.
- KIELER Advanced Rendering of Model Appearance (KARMA), a collection of tools, to enable the customisation of the diagram appearance.
- KIELER Editor for Graphs (KEG), to permit the development of graph algorithms and layout algorithms.

The layout and demonstrators areas are relevant for this work, since we implemented the SyncCharts Editor and SCCharts Editor and used the automatic layout to range the modeled diagrams.

1.3 YAKINDU Open Source

YAKINDU[3] Open Source is a collection of Eclipse-based software tools designed for the development of embedded systems. It was developed by the Itemis[4] AG company. YAKINDU Open Source is composed of two main projects: YAKINDU Statechart Tools (SCT) and YAKINDU Damos. YAKINDU SCT allows the development of reactive, event-driven systems using Statecharts. YAKINDU Damos allows the development of reactive, dataflow–oriented systems [5].

1.4 The Aim of this Thesis

The primary task to be undertaken for this thesis was the development of a graphical editor for SyncCharts. A main requirement was the ease of maintenance, in particular compared to the existing GMF based Thin Kieler SyncCharts (ThinKCharts) Editor. A further requirement was the ease of customization, motivated by the ongoing development of SCCharts, a variant of SyncCharts. At the onset of this thesis work, YAKINDU SCT was identified as a suitable basis that should be used in this thesis.

[3] http://www.itemis.com/itemis-ag/products/language=en/43503/yakindu-open-source/
[4] http://www.itemis.de/
[5] http://code.google.com/a/eclipselabs.org/p/yakindu/

1.5 Overview

The rest of this thesis is organized as follows. Chapter 2 presents related work. It gives an overview of the ThinKCharts Editor and introduces the SCCharts concept. Finally, several tools for the development of graphical editors are evaluated.

After that, Chapter 3 presents the technologies used for the implementation of the SyncCharts Editor based on YAKINDU. It presents the Eclipse platform. Afterwards, it introduces the EMF, GEF, and GMF technologies, which are used for the generation of graphical editors. The Xtext and Xtend technologies as well as the Dependency Injection design pattern are also introduced. Finally, it gives an overview of the YAKINDU SCT Editor.

Chapter 4 provides a comparison between the ThinKCharts Editor, the SCCharts Editor, and the YAKINDU SCT Editor. It outlines the requirements to implement the new SyncCharts Editor as well as the SCCharts Editor and gives an abstract overview of the project design.

Chapter 5 presents the contribution of Itemis AG in this work and gives an implementation of the SyncCharts Editor as well as the SCCharts Editor.

Finally, Chapter 6 concludes this thesis by summarizing it, evaluating it, and giving some ideas for future research.

2 Related Work

The interest in designing and developing user-friendly software is increasing. Interactivity of software applications plays an important role in productivity gains for programmers. The aim is to make systems easier to use.

Section 2.1 introduces the ThinKCharts Editor, an interactive editor which is replaced by the resulting SyncCharts Editor of this work. That provides an overview of which functionalities the new editor should offer. Section 2.2 introduces SCCharts, which are the modeling language used by the SCCharts Editor. Finally, in Section 2.3 several tools for the development of graphical editors are evaluated.

2.1 The ThinKCharts Editor

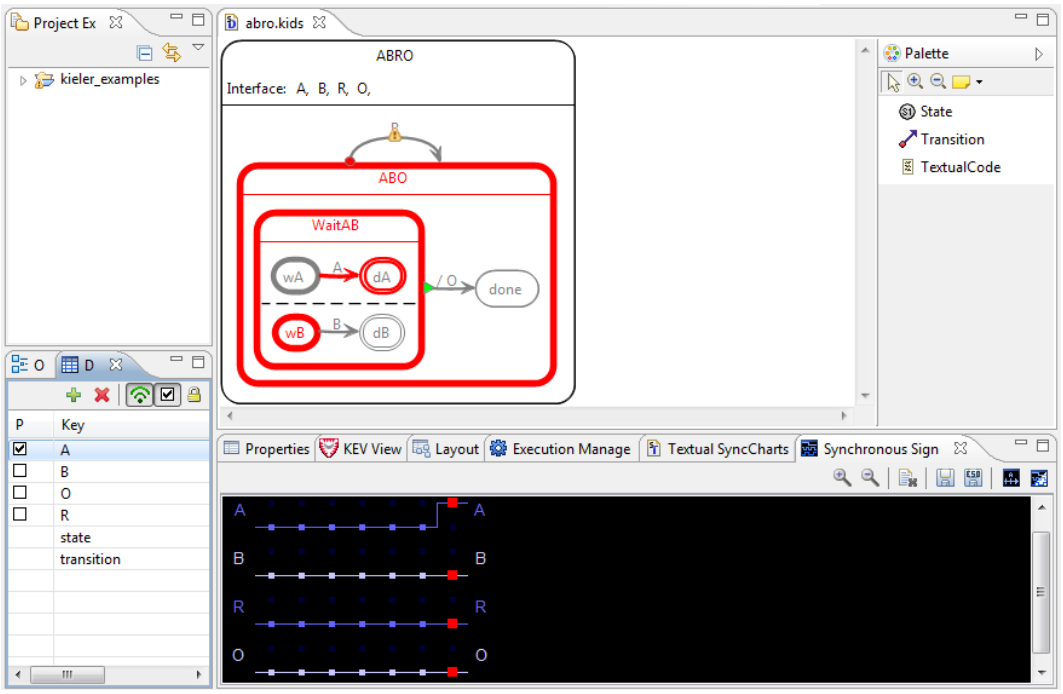


Figure 2.1: Simulation of the ABRO example in the KIELER framework

The ThinKCharts Editor is a GMF based editor that allows the graphical modeling of reactive systems. It was developed and documented by Matthias Schmeling [30] in the context of his diploma thesis and was enhanced and maintained by Hauke

Fuhrmann [14]. The ThinKCharts Editor was generated using the Graphical Modeling Framework (GMF) Tooling project, introduced in Section 3.1.3. The GMF Tooling uses templates to generate a diagram code. However, adding and manipulating these templates is time-consuming. Therefore, this work extends the YAKINDU SCT Editor to implement a SyncCharts Editor. This approach minimizes the effort for maintenance.

2.1.1 Introduction to SyncCharts

The representation of SyncCharts is similar to Statecharts. They use states, transitions, and actions to describe the behavior of a reactive system. A state can be either a simple state or a macro state. A simple state is empty, while a macro state contains a SyncChart. SyncCharts support three types of transitions: *strong abortion*, *weak abortion*, and *normal termination*. A transition is defined by an action, consisting of a *trigger* and an *effect*.

The ABRO example in Figure 2.1 is a typical example, which contains the main elements of SyncCharts. It combines instantaneous signal reaction, preemption, and concurrency. It has three input signals A, B, and R and an output signal O. The system waits concurrently for the input signals A and B before emitting the output signal O. If at any time the input signal R (Reset) is present, the behavior of the system is preempted to start from the beginning.

2.1.2 The Editor

The ThinKCharts Editor has four main functions:

1. **Editing**: The editor combines the graphical and textual modeling of SyncCharts. It permits adding, removing, modifying, and manipulating SyncCharts elements by changing the appearance or arranging them. The palette, located on the right side in Figure 2.1, displays a list of SyncCharts elements to be used in the editor. It is also possible to add elements using the context menu (right-click). The properties tab, located on the bottom in Figure 2.1, allows the manipulation of attributes of the SyncCharts elements. The ThinKCharts Editor uses the *attribute awareness* approach to improve and simplify the creation of SyncCharts. This approach allows to change the appearance of a figure in the editor by manipulating attributes.

2. **Validation**: It permits the syntactic and semantic check of a SyncChart and ensures that no invalid one is created. It consists of a list of rules that must be respected during the modeling. If a rule is violated, the faulty element is marked by an error marker. Examples of validation rules are the recognition of non-reachable states or the reference of a non-existent signal.

3. **Simulation**: The KIELER framework offers a SyncCharts simulator [19][20] based on the KIELER leveraging Ptolemy semantics (KlePto) project [21]. It

offers the user the opportunity to directly execute and simulate the SyncChart. It is also possible to set and check the values and statutes of variables and signals as well as triggers and effects during the simulation. Figure 2.1 shows a SyncChart being simulated. The execution manager offers the opportunity to run an execution, to execute a SyncChart step by step, and to stop it. The duration of a step can be manipulated by the user. The active states are marked with a thick red border on the active diagram editor. Located on the left side, the data table allows to emit a signal. It also permits to assign a value to a signal or a variable. The synchronous signals tab, located on the bottom, shows the signals statuses in each step.

4. **Code Generation**: It allows to generate SyncCharts in C (SC) [33] and Synchronous Java (SJ) [23] code from the SyncChart.

The main function of the derived YAKINDU SyncCharts Editor is the editing of SyncCharts models and the verifying of their validity. Therefore, the editing and validation parts, presented above, are important in the context of this work.

Since this work aims to implement a SyncCharts Editor as well as a SCCharts Editor, the following section introduces the SCCharts modeling language.

2.2 SCCharts

SCCharts are an extension of SyncCharts. They are based on sequential constructiveness, which was introduced by von Hanxleden et al. in [35]. An implementation for the SCCharts concept was presented by Björn Duderstadt in the context of his diploma thesis [12]. It consists of an editing system based on the *NI Diagram SDK*, a Visual Studio solution.

Syntactically, SCCharts do not differ much from SyncCharts. They have the same graphical elements: states, transitions, etc. However they differ in their semantics and the use of variables. In the context of this work, the semantics is ignored since there is no intention to implement a simulator or a code generator for SCCharts, but the goal is to implement the editor.

2.3 Generation of generic visual editors

Programming and adapting modeling visual editors requires a lot of time and effort. There are tools that aim to accelerate the development of such editors by providing a platform for the generation of source code based on a metamodel. An evaluation of several tools is presented by Amyot et al. in [2]. The tools are Generic Modeling Environment (GME), Tau G2, Rational Software Architect (RSA), XMF-Mosaic, and Eclipse with EMF and GEF.

GME [1] is a configurable toolkit for the generation of domain-specific modeling environments. It is destined for programming language that supports Component Object Model (COM) like C++ and C#.

Telelogic Tau G2 [2] is a model-driven development environment. It allows the generation of editors based on Unified Modeling Language (UML) profiles[3]. Telelogic Tau G2 provides two mechanisms for the definition of profiles: Stereotype Mechanism (SM) and Metamodel Extension Mechanism (MEM).

RSA [4] is a software development environment for the Eclipse platform. It also uses profiles and provides the SM to define them.

XMF-Mosaic [5] is a development environment for modeling languages for the Eclipse platform. It is extensible and also provides a metamodel which is defined in a class diagram in MetaObject Facility (MOF)[6]/XCore[7].

Eclipse is introduced in Section 3.1.

Amyot et al. evaluated the tools by creating a simple metamodel that covers interesting element notations. Afterwards, they generated a graphical editor using each tool. The evaluation criteria was the following:

- Graphical completeness: If all the notation elements may be represented.
- Editor usability: If the generated editor supports the simple manipulation of notation elements and properties as well as undo/redo, load/save, etc.
- Effort: The effort (time and effort) required to learn the tool.
- Language evolution: The effort to extend the older models by changes.
- Integration with other languages: The integration with other tools.
- Analysis capabilities: The capability to analyze and transform a produced model.

[1] http://www.isis.vanderbilt.edu/Projects/gme/
[2] http://www-01.ibm.com/software/rational/
[3] http://www.uml.org/#UMLProfiles/
[4] http://www-306.ibm.com/software/awdtools/architect/swarchitect/
[5] http://www.xactium.com/
[6] http://www.omg.org/mof/
[7] http://wiki.eclipse.org/Xcore/

	GME	Tau G2	RSA	XMF-Mosaic	Eclipse
Graphical Completeness	0	−	−−	−	+
Editor Usability	0	0	−	−	++
Effortlessness	0	−	+	−	−−
Language Evolution	+	?	?	?	0
Integration	−	+	+	−	+
Analysis / Transformation	0	0	−	+	0

−− for Very Low, − for Low, 0 for Medium, + for High, ++ for Very High

Table 2.1: Overview of the comparison [2]

Table 2.1 represents the results of the experiment. The comparison shows that Eclipse is the best in the usability (Editor usability) and the representation of notation elements (Graphical completeness). It offers a good integration solution via extension points. The language evolution as well as the analysis capabilities are sufficient (Medium). However, Eclipse is the most difficult to be mastered (Effort).

Amyot et al. concluded that Eclipse appears to be the most viable and mentioned that the development effort will be proportional to the benefits.

3 Used Technologies

In order to implement a SyncCharts Editor based on YAKINDU SCT, several technologies and tools have been used. They are presented in the following.

3.1 Eclipse

The editor developed in this work is implemented as a set of Eclipse plug-ins. Therefore, this section introduces the Eclipse platform and gives an overview of its plug-in concept.

Eclipse[1] is a platform developed by IBM in 2001. It is an open source software, which was used earlier only for the development with the Java programming language. It is now used as an Integrated Development Environment (IDE) for the programming with various languages like Java, PHP, C, and C++.

Eclipse enables the development of rich client applications by providing a Rich Client Platform (RCP) [18]. It was introduced for the first time in Eclipse version 3. It allows to reuse Eclipse components for the development of custom client applications. The RCP is composed of several components, among them there are:

Equinox OSGI[2] implements the Open Services Gateway initiative (OSGi). The task of the OSGi is to run Java code from multiple sources in a single Java Virtual Machine (JVM) [27]. With Equinox, applications can be implemented in a modular way. An application can be considered as a set of bundles, which can be installed, started, and stopped independently.

Core platform controls the life cycle of an Eclipse application. It represents the basic infrastructure. It is composed of a small number of plug-ins[3].

Standard Widget Toolkit (SWT) [25] is a graphical library. It provides the possibility to produce a Graphical User Interface (GUI), that is well integrated in the Operating System (OS). The SWT offers many widgets ready to be used. Among these widgets are buttons, combos, and progress-bars[4]. It is also possible to use the standards dialog boxes of the OS, to display Hypertext Markup Language (HTML) pages using the standard internet browser, and to support Object Linking and Embedding (OLE)/COM on Windows allowing the integration of Windows applications.

[1] http://www.eclipse.org/
[3] http://en.wikipedia.org/wiki/Plug-in_(computing)/
[4] http://www.eclipse.org/swt/widgets/

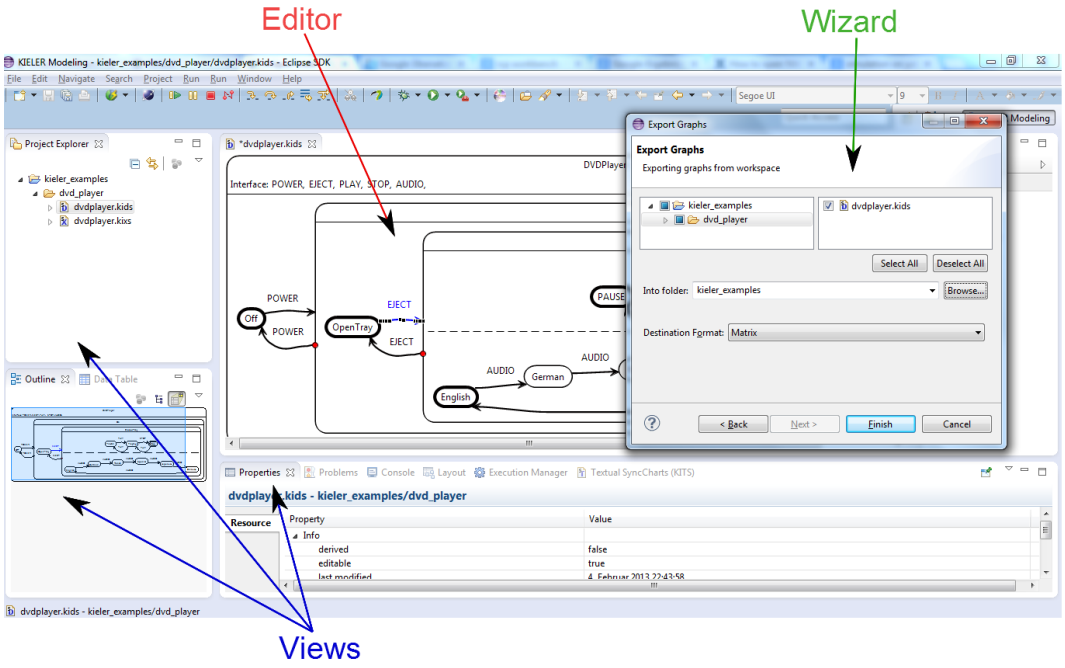


Figure 3.1: The Eclipse workbench (KIELER perspective)

JFace[5] [1] is a User Interface (UI) toolkit designed to work with SWT. It permits to handle the following tasks: Sorting, filtering, and updating of widgets. It provides tools for creating wizards and managing preferences. It allows to manage the connection between graphical components and to display data through the notion of viewer. JFace proposes, in addition to the standards dialog boxes of the OS, to use customized dialog boxes.

Eclipse workbench is the graphical solution proposed by the Eclipse RCP. It offers a collection of elements, which cover the basic needs of programming client applications. Figure 3.1 represents an overview of the Eclipse workbench. The workbench provides views, editors, perspectives, and wizards:

Views are used in many different ways. They are typically used for browsing files or packages, modifying properties, or displaying problems. Besides the typical use, views also serve to control executions of simulations, to set inputs, and to display outputs, as it was used in the KIELER project.

Editors are used to edit or browse a document.

Perspectives determine the collection of visible views and their locations in the user interface. There can only be one selected perspective at any time.

Wizards allow the user to enter information in a structured way (a sequence of dialog boxes), which help to perform a task. As seen in Figure 3.1, the Export Graphs Wizard permits the export of graphical diagrams into the selected folder with the given destination format.

Eclipse provides the concept of extension points and extensions, which is a mechanism that permits the connection between plug-ins. Extension points are defined in a plug-in to permit other plug-ins, which define extensions, to add functionalities in the first. Extensions and extension points are described in an Extensible Markup Language (XML) document, which may be added and modified using the Eclipse manifest editor.

YAKINDU SCT provides extension points, which are used in the context of this work to extend the editor by adding functionalities that are relevant to the SyncCharts Editor.

3.1.1 Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) is used in this work to define the underlying models of the SyncCharts Editor.

EMF[6] is an Eclipse framework that generates from a given metamodel a set of Java classes. It provides a set of adapter classes for the viewing and editing of the model as well as a basic editor. The metamodel can be described via different languages, e.g., UML[7], XML[8], or XML Metadata Interchange (XMI)[9].

Modeling

To specify the metamodel with EMF, a file in an ecore format can be created directly in Eclipse. The ecore modeling elements are presented in the following:

EPackage represents a package.

EClass represents a class, which may contain a set of attributes (EAttribute) and references (EReference).

EAttribute is an attribute having a name and a type.

EReference is used to associate classes. It points to a reference class and may represent a containment.

EDataType represents the type that an attribute can be.

[6] http://www.eclipse.org/modeling/emf/
[7] http://www.uml.org/
[8] http://www.w3.org/XML/
[9] http://www.omg.org/spec/XMI/

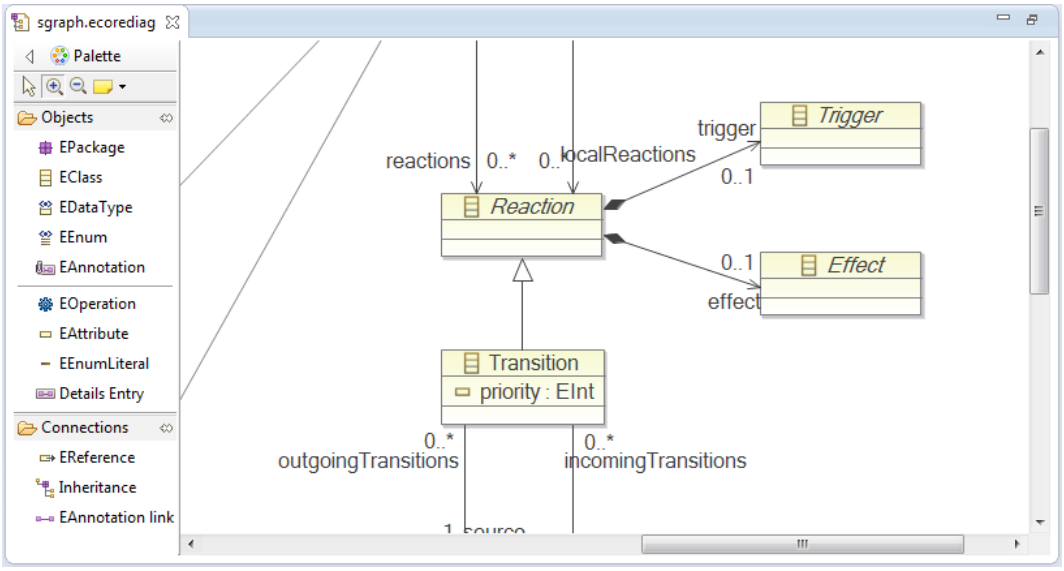


Figure 3.2: The visual editor for creating EMF models

EMF provides a graphical editor, which allows to graphically create and modify the metamodel using UML-like class diagrams. The advantage of this editor is that it offers a view of the metamodel allowing the clear understanding of how objects interact with each other. Figure 3.2 depicts a subset of the YAKINDU SCT Sgraph metamodel, which is fully explained in Section 3.2.1. The palette, located on the left side, offers the different elements to be used for modeling the metamodel. Classes are represented by boxes. The top partition of the box is conceived to define the name of the class. Attributes are defined in the middle of the box. The partition in the bottom is destined for operations. As seen in Figure 3.2, the Transition class has an attribute of type EInt. It inherits from the reaction class. The inheritance is depicted by a solid line having an unfilled arrowhead. A Reaction consists of an optional Trigger and an optional Effect.

Besides the graphical editor, EMF allows to define ecore models using a tree-based editor. Figure 3.3 represents the same subset of the YAKINDU SCT Sgraph metamodel presented in Figure 3.2. Classes, attributes, and references can be created using the context menu. The property view permits to define a name, a type, and other properties.

Ecore models also can be created using the Xcore syntax[10].

In the context of this work, the EMF graphical editor is used to understand YAKINDU SCT metamodels and the EMF tree-based editor is used to create the metamodel of the SyncCharts Editor.

[10] http://wiki.eclipse.org/Xcore

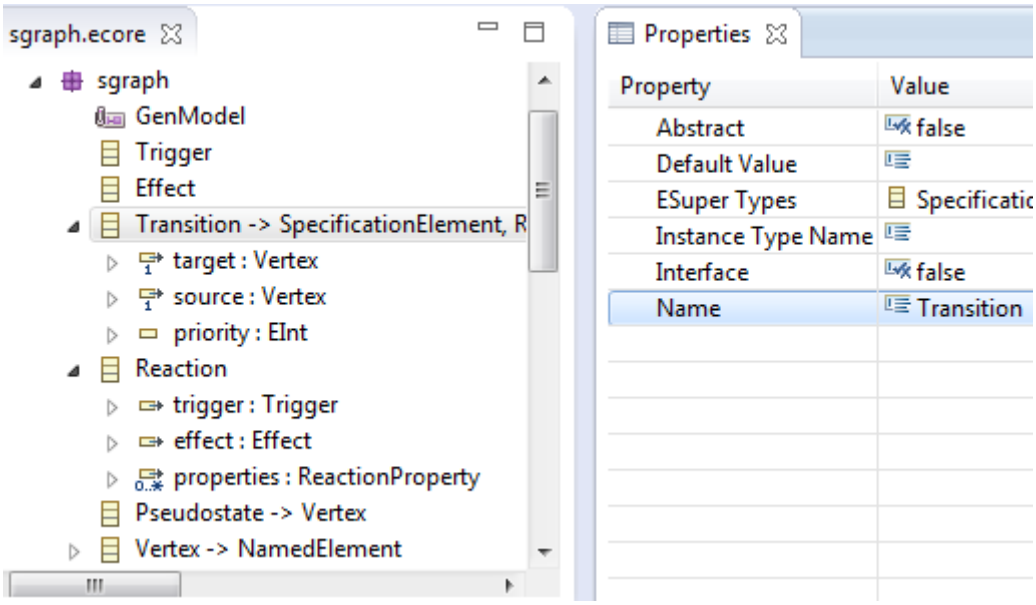


Figure 3.3: The tree-based editor for creating EMF models

Implementing Java Code

As mentioned before, EMF generates Java code based on the metamodel. In the first place, a generator model is generated that allows for implementing Java code. This model contains information and properties for the code generation, e.g. the model plug-in identifier, the model directory, the file extension, and the base package. Once the properties are set, the Generate the Model Core option from the context menu will create the Java implementation of the EMF model into the selected model directory.

3.1.2 Graphical Editing Framework (GEF)

The Graphical Editing Framework (GEF)[11] is an Eclipse plug-in for the creation of graphical editors and views destined for the Eclipse workbench. It uses the Draw2D[12] framework to display the graphics on an SWT[13] canvas. The advantage of using GEF is that it enables simple changes to be applied to the model from the view.

3.1.3 Graphical Modeling Framework (GMF)

EMF defines the underlying models of the SyncCharts Editor and GEF allows to display the graphics. In order to integrate the EMF models with the GEF framework and to provide a generative infrastructure for building modeling applications, the

[11] http://www.eclipse.org/gef/
[12] http://www.eclipse.org/gef/draw2d/
[13] http://www.eclipse.org/swt/

GMF technology is used. GMF[14] is an Eclipse plug-in that bridges between EMF and GEF technologies.

On the one hand, a GMF based editor can be generated using the GMF Tooling project, which provides a model-driven approach for the generation of graphical editors in Eclipse. The GMF Tooling was used to generate the ThinKCharts Editor.

On the other hand, a GMF based editor can be created using the GMF runtime, which provides many features that must be coded by hand. YAKINDU SCT Editor uses the GMF runtime.

GMF provides *figures*, which are basically *nodes* and *edges*. On the one hand, a node can be for example a rectangle, an ellipse, or another geometric object. On the other hand, an edge is a connection between two nodes. A figure may be extended to create a custom one.

As previously stated, GMF is used to generate a graphical editor. The latter is equipped with tools allowing the creation of diagrams (e.g., add figures into the editor or create a connection between two figures). Once the diagram is ready, it is saved in the form of a notation model. That preserves the structure and hierarchy of the diagram as well as the position and size of the modeling objects.

3.1.4 Xtext and Xtend

In the context of this work, Xtext defines the YAKINDU SCT textual language, which is extended to define the SyncCharts textual language. Xtext generates an EMF metamodel. The metamodel for the textual language of YAKINDU SCT (Stext) is presented in Section 3.2.2. Xtend is used in YAKINDU SCT to check the type conformance of *expressions*, which are introduced in Section 3.2.2.

Xtext

Xtext[15] is a framework that allows the user to define his own Domain-Specific Language (DSL). It consists of a set of plug-ins, which can be installed in the Eclipse IDE. The grammar of the DSL is specified using the Xtext textual language.

Xtext generates an ecore model. It also permits to reuse and extend an existing ecore model. Xtext uses EMF for the generation of a DSL editor. The generated editor offers many useful features like syntax coloring, code completion, and static analysis.

Figure 3.4 shows a subset of the Xtext textual language, which generates the EMF metamodel presented in Figure 3.2 and Figure 3.3. The Interface rule specifies that an interface may either be a Variable or a Signal. The Signal rule specifies that a signal definition starts with defining its direction, followed by the literal signal and the name. The Signal direction may be an input, an output, or both. The direction is specified by writing the literals input or output. The name represents the signal identifier. A Variable rule specifies that a variable definition may start with the

[14] http://www.eclipse.org/modeling/gmp/
[15] http://xtext.itemis.com/xtext/language=en/36527/about-xtext

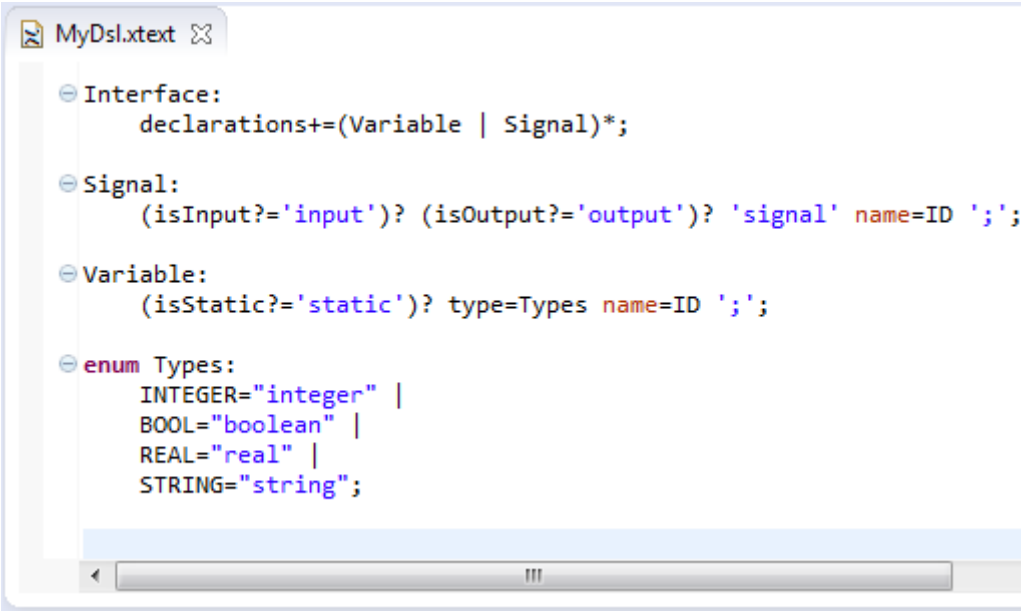


Figure 3.4: The Xtext DSL grammar

literal static, followed by the type and the name. The type is an enumeration. It contains the possible types that a variable may be.

Xtend

Xtend is a programming language. It facilitates the use of Java by making the code easier to read and to write. Further details on how to use Xtend can be found on the Xtend website[16].

3.1.5 Dependency Injection (Google-Guice)

Dependencies between software artifacts (classes, packages, functions, etc.) make it difficult to update and maintain a software. YAKINDU SCT is designed for extensibility. Therefore, it uses the Dependency Injection concept and precisely Google Guice to remove hard-coded dependencies.

Dependency Injection [36] is a design pattern that allows to dynamically create dependencies between different classes. By using this method, the dependencies between classes are no longer hard-coded and expressed in the code statically but dynamically determined at runtime.

Google Guice[17] is a Java software framework for Dependency Injection. It is released by Google under the Apache Licence. Google Guice uses a special module

[16] http://help.eclipse.org/helios/topic/org.eclipse.xpand.doc/help/Xtend_language.html
[17] http://code.google.com/p/google-guice/

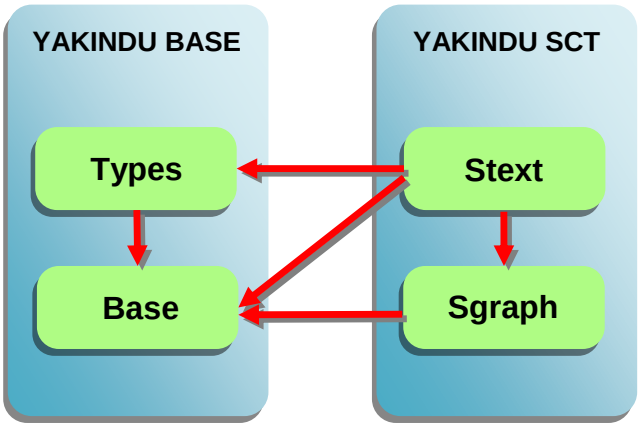


Figure 3.5: Dependencies between YAKINDU metamodels

to define the relations between interfaces and their implementations. To use this module, an injector must be instantiated. The corresponding class has to be passed as a parameter to the injector. The *@Inject* annotation enables to get an instance of any of the bound classes.

3.2 YAKINDU SCT Editor

YAKINDU SCT Editor is a set of Eclipse plug-ins for the modeling with Statecharts [16]. It is based on EMF, GEF, and GMF, which offer sophisticated features for the development of graphical editors. The advantage of using these technologies is the simplicity to adapt the editor to changes [2]. The SyncCharts Editor presented here extends the YAKINDU SCT Editor. The metamodel used by the latter allows to represent Statecharts. Therefore, it should be adapted to the SyncCharts syntax. In the following, the YAKINDU SCT Editor metamodel is presented. This permits to delimit where to make changes in the metamodel.

YAKINDU SCT Editor provides two metamodels for the definition of its syntax. As depicted in Figure 3.5, the *Stext* metamodel extends the *Sgraph* metamodel as well as the *Types* and *Base*[18] metamodels, which are implemented by the YAKINDU Base project. The YAKINDU Base project contains elements used together with the YAKINDU Damos project. The *Stext* defines the textual representation of Statecharts. It implements the interfaces and reactions as well as the expression language, which will be used to write the triggers and effects. The *Sgraph* metamodel extends the Base metamodel. It defines the graphical representation of Statecharts. It specifies how regions, states, and transitions are defined.

[18] org.yakindu.base.types

3.2.1 Sgraph

Sgraph[19] is the metamodel for the graphical representation of the abstract elements of a Statechart (see Figure 3.6).

The basic elements introduced in the *Sgraph* consist of: Statechart, Region, Vertex, State, and Transition. In the following, these are described.

Statechart

The Statechart is the root element of the model. It extends SpecificationElement, ReactiveElement, ScopedElement, and CompositeElement. A SpecificationElement has a String attribute allowing to specify a textual definition in an element. The latter may be, e.g., an interface or a reaction declaration. This part is supported by the *Stext* metamodel introduced in the next subsection. A ReactiveElement has two attributes: reactions and localReactions. Each of these attributes contains zero or more elements of type Reaction. A Reaction contains two attributes. The trigger attribute is of type Trigger and the effect attribute is of type Effect. A ScopedElement comprises a list of elements of type Scope as well as a namespace attribute. A Scope consists of Declarations, Events, and Variables. These are also implemented in the *Stext* metamodel. A CompositeElement is composed of a list of Regions.

Region

A Region extends the NamedElement from the Base metamodel. It comprises Vertices, a priority of type EInt, and exactly one CompositeElement. It means that every element that contains a Region is a composite element. A Region may be contained by a State or a Statechart.

Vertex

A Vertex has the following attributes: parentRegion, incomingTransitions, and outgoingTransitions. The parentRegion attribute is of type Region. It contains the parent region, in which the current vertex is included. The incomingTransition and outgoingTransitions are two lists of Transitions that represent respectively the incoming and outgoing transitions of the current vertex. A Vertex may be extended by a RegularState or a Pseudostate.

State

On the one hand, a RegularState may be a State or a Finalstate. On the other hand, a Pseudostate may be extended by a Choice, a Synchronization (also called junction), an Entry, or an Exit. A State extends SpecificationElement, ReactiveElement, ScopedElement, RegularState, and CompositeElement. It contains several attributes: An

[19] org.yakindu.sct.model.sgraph

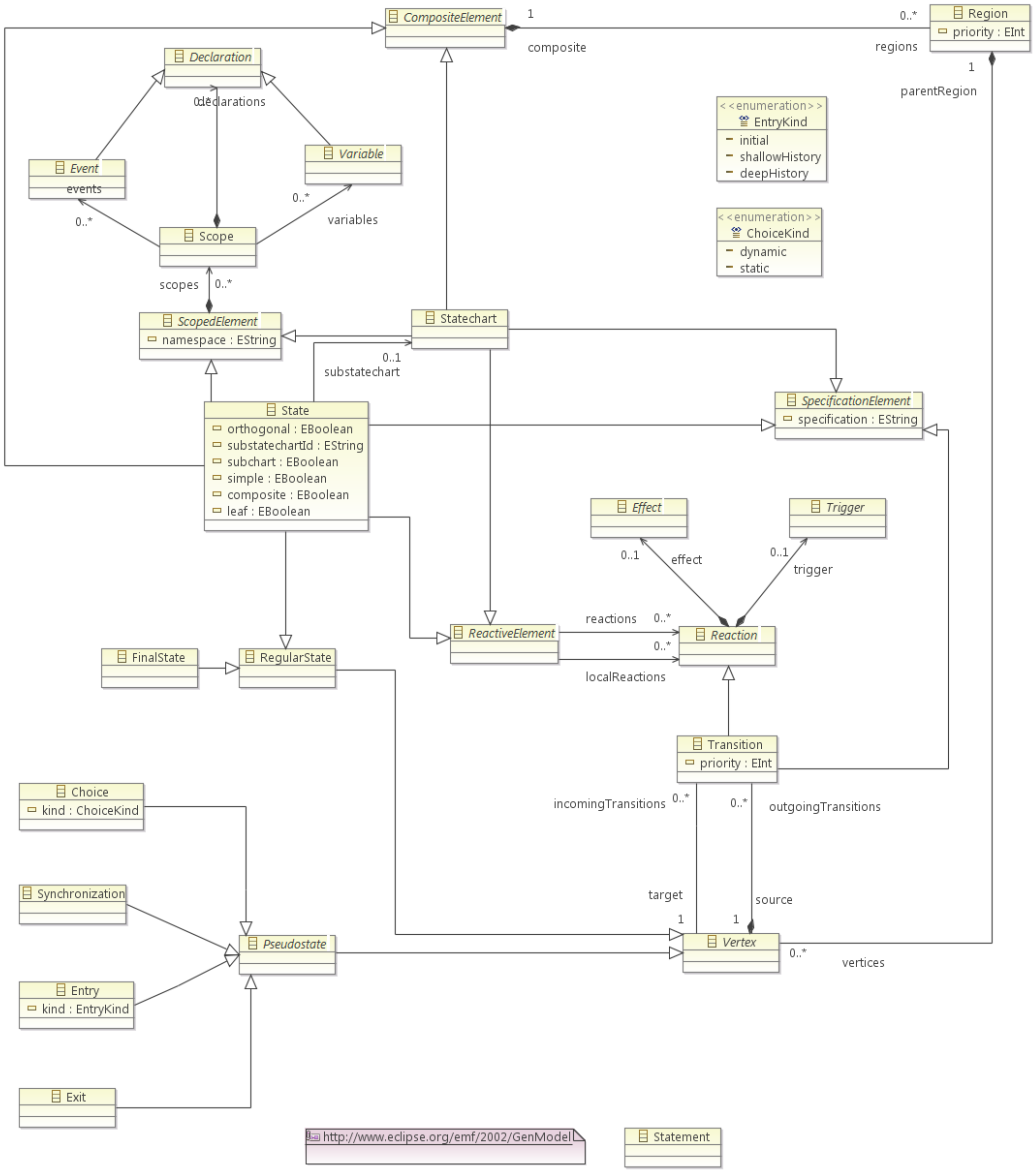


Figure 3.6: YAKINDU Sgraph

orthogonal attribute of type EBoolean, substatechart of type Statechart, substatechartId of type EString, subchart of type EBoolean, simple of type EBoolean, composite of type EBoolean, and leaf of type EBoolean. An Entry has an attribute of type EntryKind. EntryKind is an enumeration. It can be either initial, shallowHistory, or deepHistory. The initial option means that the Entry is an initial state. The shallowHistory and deepHistory options refer to a history state.

Transition

A Transition is a SpecificationElement and at the same time a Reaction. This involves that a Transition is composed of Triggers and Effects. It has an attribute of type EString named specification.

The following subsection presents the Stext metamodel, which represents the textual representation of Statecharts.

3.2.2 Stext

The metamodel presented in Figure 3.7 is the Stext[20] metamodel. It extends the *Sgraph*, *Base*, and *Types* metamodels. The Types metamodel implements the events and variables types.

Starting from the top root element is the best way to understand the diagram. It contains a list of elements of type DefRoot. A DefRoot may be a StatechartRoot, a StateRoot, or a TransitionRoot. The elements StatechartRoot, StateRoot, and TransitionRoot have each one an attribute respectively of type StatechartSpecification, StateSpecification, and TransitionSpecification.

StatechartSpecification

A StatechartSpecification extends the ScopedElement from the *Sgraph* metamodel. It is composed of elements of type StatechartScope. A StatechartScope may be either an InterfaceScope and an InternalScope. The InterfaceScope comprises declarations. A declaration may be of type EventDefinition, VariableDefinition, or OperationDefinition. Note that the InterfaceScope has an identifier attribute, which is used to distinguish between the interfaces defined into a StatechartSpecification. An EventDefinition extends the Event element from the *Sgraph* metamodel. It has several attributes. The direction attribute is of type Direction. The Direction is an enumeration, which is composed of the literals: local, in, and out. The EventDefinition also has an attribute, which represents the type of an event (e.g., integer, boolean, or real). A VariableDefinition extends the Variable element already defined in the *Sgraph* metamodel. It has two attributes of type EBoolean. An attribute expresses whether the variable is only for reading. The other attribute determines whether it is an external variable. The VariableDefinition also has another attribute of type

[20] org.yakindu.sct.model.stext

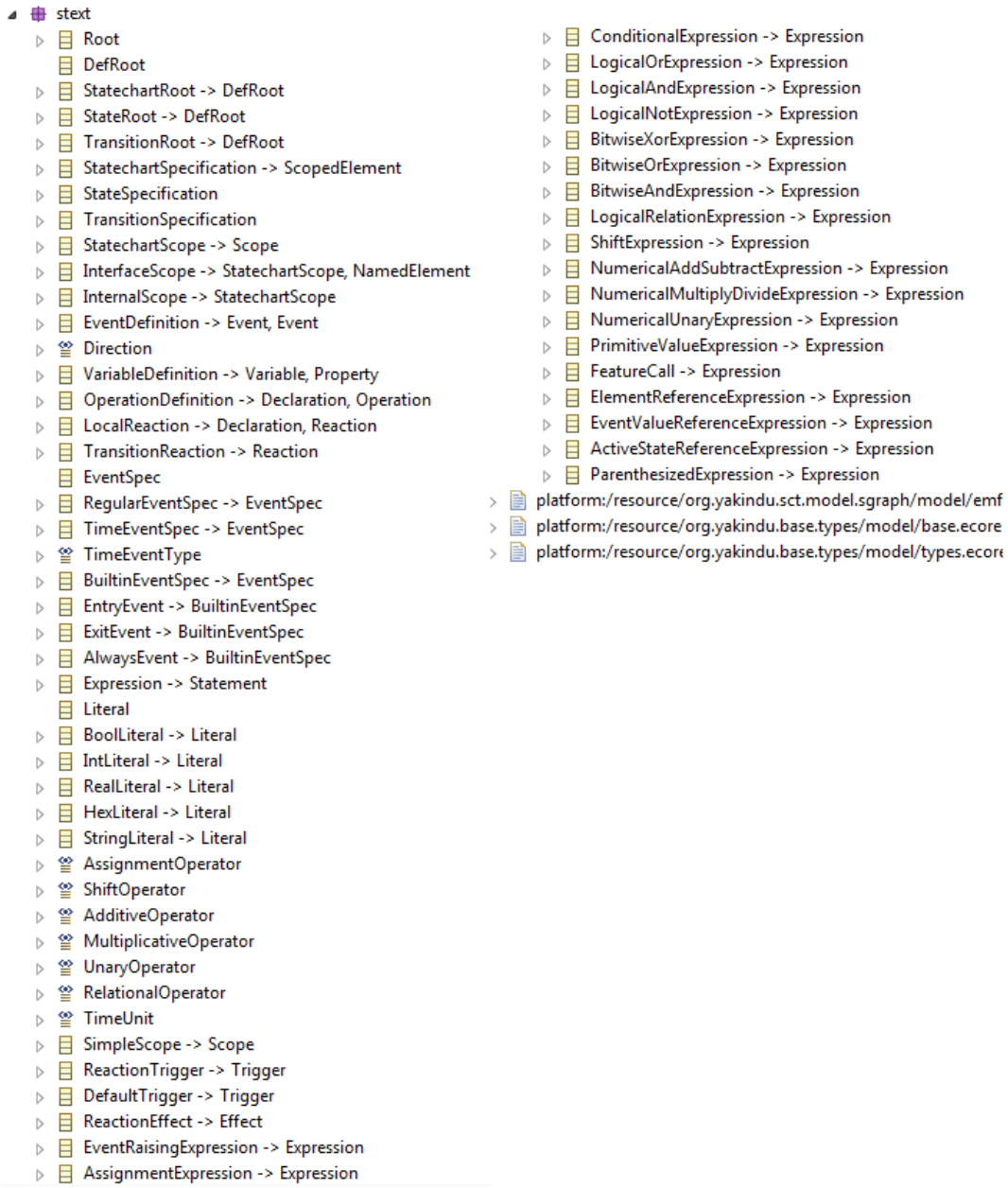


Figure 3.7: YAKINDU Stext (EMF class elements)

Expression. The Expression element will be explained later. An OperationDefinition allows to declare an operation in an InterfaceScope.

StateSpecification

A StateSpecification contains a Scope attribute. The Scope element is already defined in the *Sgraph* metamodel. As mentioned in the previous subsection, the Scope element consists of Declarations, Events, and Variables. Hence, the deduction of the possibility to specify Declarations, Events, and Variables in a State and therefore a State extends SpecificationElement. A StateScope enables to define a collection of declarations within a state.

TransitionSpecification

A TransitionSpecification comprises a TransitionReaction. The latter is a Reaction, which has already been introduced in the previous subsection. The conclusion here is that a TransitionSpecification permits to specify the ReactionTrigger and ReactionEffect of a Transition. A ReactionTrigger contains a list of triggers of type EventSpec. A ReactionTrigger also contains a guardExpression attribute of type Expression. An EventSpec may either be a BuiltinEventSpec, a TimeEventSpec, or a RegularEventSpec. A BuiltinEventSpec is a trigger, which denotes when to fire the ReactionEffect (entry, exit, always, or oncycle). The TimeEventSpec consists of a TimeEventType, a value, and an unit attributes. The TimeEventType is an enumeration and contains the values after and after. The TimeEventSpec expresses the physical time when the ReactionEffect is to be fired (e.g., after 5 seconds). The RegularEventSpec contains an event of type Expression. A ReactionEffect comprehends a list of actions of type Expression or EventRaisingExpression.

Expression

Expressions in YAKINDU SCT are similar to Java programming languages. In the following, several examples are presented followed by the different kinds of expressions that are provided by the Stext metamodel.

```
A || B
cond ? A : B
((A && B) || (!C && V==30))
```

Several elements compose an Expression. Figure 3.8 shows an excerpt of its definition. A PrimitiveValueExpression has an attribute of type Literal. A Literal may be a BoolLiteral, an IntLiteral, a RealLiteral, a HexLiteral, or a StringLiteral. The Expression permits to express conditions (ConditionalExpression). LogicalAndExpression, LogicalOrExpression, LogicalNotExpression, and ShiftExpression allow to express the logical and shift expressions (&&, ||, !, <<, >>). BitwiseXorExpression, BitwiseOrExpression, and BitwiseAndExpression permit to define the bitwise arithmetic expressions. A LogicalRelationExpression expression has three elements. The leftOperand

and rightOperand are of type Expression. The RelationalOperator is an enumeration, which is composed of literals that enables the comparison of the left and right operands (e.g., <, >, or ==). The Expression enables the definition of statements, which can be either an assignment expression (AssignmentExpression), operation call, or raising an event (EventRaisingExpression). The elements NumericalAddSubtractExpression, NumericalMultiplyDivideExpression, and NumericalUnaryExpression permit to express the addition, subtraction, multiplication, division, and the unary arithmetic operations (positive, negative, and complement). ParenthesizedExpression implies that an expression may be surrounded by parenthesis.

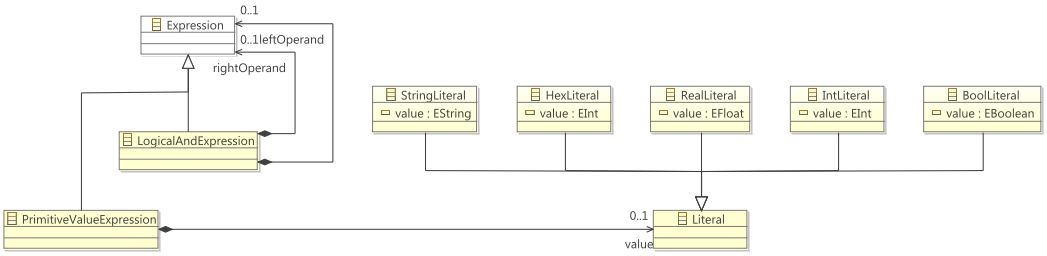


Figure 3.8: An excerpt of the Stext expressions definition

4 ThinKCharts/SCCharts Editor vs YAKINDU SCT Editor

As mentioned in Chapter 1, the aim of this work is to implement a SyncCharts Editor as well as a SCCharts Editor that extend the YAKINDU SCT Editor. However, the latter allows the modeling based on the concept of Statecharts. This chapter presents the ThinKCharts Editor, the SCCharts Editor, and the YAKINDU SCT Editor. Then, a comparison between the editors is provided. The aim of this comparison is to search out which features of YAKINDU SCT Editor can be reused, which features should be adapted, and which features are to be added. This approach allows to minimize the effort for maintenance. Finally, the project design is presented.

4.1 Modeling with ThinKCharts Editor

The ThinKCharts Editor offers several elements, which allow the modeling of SyncCharts. Besides these elements, the editor provides several features allowing the manipulation of the modeled SyncChart. This section represents the different SyncCharts elements as well as the provided features: Palette, pop-up balloons, context menu, properties view, and validator. Finally, it gives an overview of the automatic layout.

4.1.1 SyncCharts elements

As mentioned in Chapter 2, SyncCharts use states, transitions, and actions to describe the behavior of a reactive system.

Regions A *region* is included in a state. It allows grouping children states and transitions. A state may have many parallel *regions*. A *region* may have a name. It has exactly one *initial state*. It is represented by a rectangle having a dashed line border. Figure 4.3 shows two parallel *regions* included in a *macro state*.

States According to its use, there are several ways to represent a state, as depicted in Figure 4.1. A *simple state* is a state without internal behavior and regions. It may have a name. A simple state is represented by a circle or an ellipse with a black outline. A *macro state* contains at least one *region*. It also may have a name. The *macro state* is drawn with a rounded rectangle, which is composed of three main compartments. The top compartment, marked in Figure 4.3 with a rectangle having a green dashed border line, allows to display

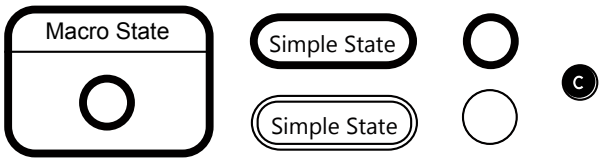


Figure 4.1: Different ThinKCharts Editor states

the name. The middle compartment, surrounded with a red dashed border lined rectangle, allows to define the *interface declaration* and *action declaration*, which are subsequently introduced. This compartment is constituted by an arbitrarily number of sub-compartments, which may be of type *interface compartment* or *action compartment*. An interface compartment contains the interface declaration. It is marked in the figure with a purple dashed rectangle. An action compartment contains the action declaration. It is surrounded with an orange dashed border line in the figure. The bottom compartment, with the blue dashed border lined rectangle, contains a set of parallel *regions*.

A state may be set to an *initial* or a *final state* by manipulating its attributes. An initial state is represented by a thick border line. A final state has a double border line. Note that a state may be initial, final, or both at the same time. In the latter case, it is represented by a thick double border line.

The *conditional state* is a pseudo state. It is drawn as a filled black circle that contains the letter C in the middle.

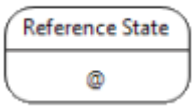


Figure 4.2: ThinKCharts Editor reference states

A state may also be a *textual* or a *reference state*. A textual state contains statements of the Esterel language. A reference state [5] uses instances of an another state, which may belong to another SyncChart (see Figure 4.2).

Interface declarations As mentioned previously, a state contains an *interface declaration*. An *interface declaration* is composed of either *signals* and *variables* declarations.

A signal may be local or it may belong to an interface by setting the attributes input and output. The attributes define the signal direction. A signal can be an input signal, an output signal, or both (input output signal). Besides the

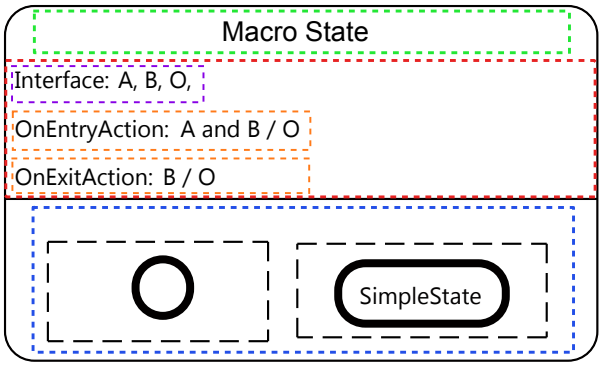


Figure 4.3: Compartments overview of the KIELER SyncCharts macro state

direction, a signal has a name and may have a type in the case of a *valued signal*. The type can be one of the following literals: pure, bool, int, unsigned, float, host, double, and string. The pure type means that a signal has no value (*pure signal*). The host type is provided by the host system. A combine operator may also be chosen for a signal. It serves to combine multiple values of a signal in case it is emitted more than one in the same tick [3]. The possible combine operators are NONE, +, *, max, min, or, and, and host. A host combine operator implies that an operator from the host language is selected. A signal may be initialized.

A variable has a name and should have a type. Like a valued signal, a variable may also be initialized. The difference is that a variable does not have a direction and a combine operator. However, a variable may be set as a constant.

State action/suspension There are three types of *state actions* [26]: *OnEntry actions*, *OnInside actions*, and *OnExit actions*. The *state action* is followed by other actions, which may be expressed in the same way as the *transition actions*. In Figure 4.3 the *OnEntry* and *OnExit actions* are used.

The *suspension* may not contain an effect. How it may be expressed is shown in the following example:

```
(A and B) / Suspend;
```

Transitions A *transition* is a connection that permits to link two states. The editor supports three types of transitions: *weak abortion*, *strong abortion*, and *normal termination* [5]. Figure 4.4 represents the different types of a transition. The first transition type, shown in the figure, is the *weak abortion transition*. It is represented by a simple black arrow. The second transition type is the *strong abortion transition*. It is represented by an arrow, which has a red circle source

Figure 4.4: SyncCharts transition types and the history transition

decorator. The transition type, with the green triangle source decorator, is the *normal termination transition*.

A transition can additionally be declared as a *history transition*, in which case, it is represented by an arrow that ends with a rounded target decorator. The history transition target decorator is filled with a grey color. It contains the letter H inside. Note that a history transition may be a weak abortion, a strong abortion, or a normal termination transition.

A state can have more than one enabled outgoing transition. Therefore, a transition must have a priority number (1 for the highest priority).

Transition actions A *transition action* is composed of a *trigger*, *effects*, and other options. It can be expressed as follows:

```
# <delay> <Trigger> / <Effects>;
```

The sharp symbol (#) is optional. It means that the transition is an *immediate transition* [5]. The delay may be an integer number. For example, 2 S waits for the second strictly future presence of S [5]. The trigger is a boolean expression. Effects are operations executed when the trigger is satisfied. An effect may be a signal emission or a variable assignment

The following examples show how a *transition action* is represented:

```
I / O;
3 I / O;
# I / A,B;
(A and (B or V=30));
pre(I) / O;
?S=5 / O
```

These examples illustrate the use of simple actions, the delay, the immediate transition, and complex boolean expression as a trigger. The pre() operator returns the presence status of a signal in the previous tick. As mentioned in Chapter 1, a signal is either present or absent in a tick. ? is the value operator. It returns the value of a valued signal. In the last example, the trigger is satisfied if the value of the signal S is equal to 5. Note that the trigger and effects are optional.

4.1.2 Tooling

Palette

The ThinKCharts Editor provides a *palette* in order to manually edit a SyncChart. Figure 4.5 shows the different elements offered by the palette, which allows to add a state, a transition, or a textual code [31].

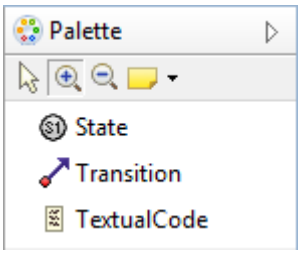


Figure 4.5: ThinKCharts Editor palette

Note that the *palette* does not offer all the required elements for the modeling of SyncCharts, such as regions. Other elements can be added using the *context menu* and the pop-up balloons menu (See Figure 4.7).

Structure-Based Editing

Structure-Based Editing is a KIELER feature, which is based on the technique of model transformation [17]. It is used in the SyncCharts Editor to modify the semantic structure of a SyncChart, without using any graphical information.

The interaction between the user and the Structure-Based Editing in KIELER framework is enabled by using the pop-up balloons menu and the context menu.

Pop-up balloons Pop-up balloons appear when hovering over a SyncCharts element in the diagram. Figure 4.6 shows the pop-up balloon that appears when a state is selected. It allows to

- create a successor for the selected state.
- create a predecessor for the selected state.
- add a choice element.
- add a region with an initial state.
- create a self loop.
- create a new state and move all existing regions to the new one.
- move all regions to the parent state and remove the selected state.
- switch between a state, an initial state, a final state, and an initial final state.

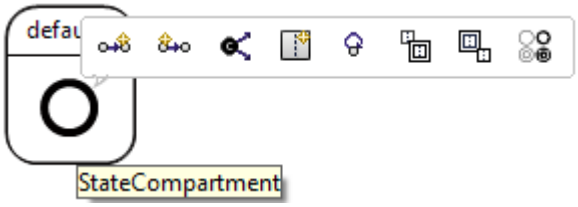


Figure 4.6: ThinKCharts Editor pop-up balloons

Context menu As depicted in Figure 4.7, the context menu can be used to add states, transitions, signals, variables, regions, and state actions/suspensions. The context menu can be opened by a right-click mouse operation on the appropriate element (e.g. a right-click on a region to create a state or on a state to add a region).

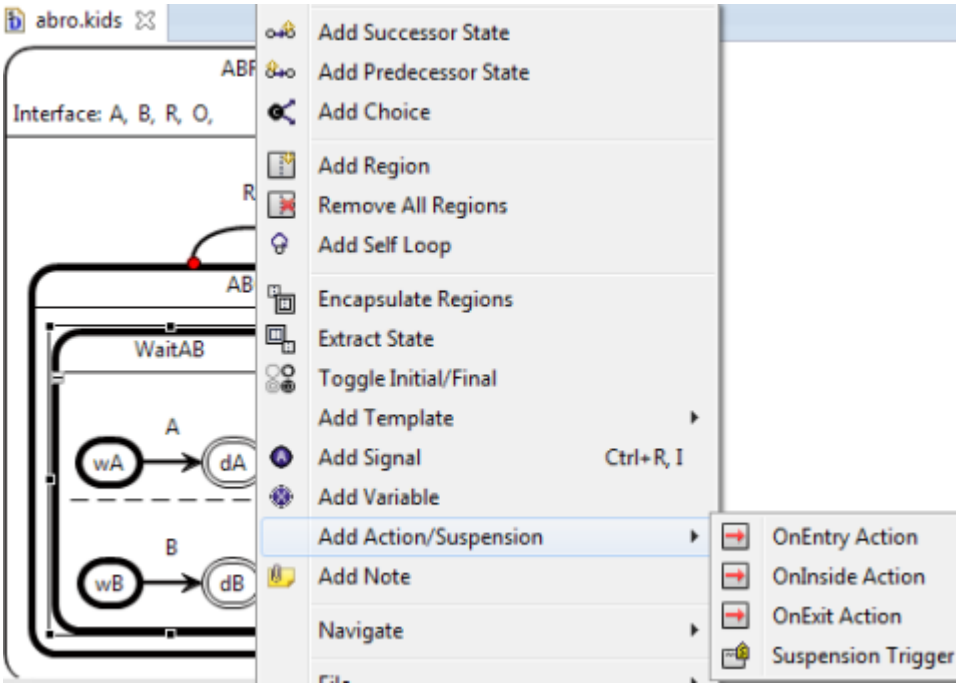


Figure 4.7: ThinKCharts Editor context menu

Once the user adds an element to the SyncChart, it is no longer possible to modify it using the palette and the context menu. Therefore, there is a properties view.

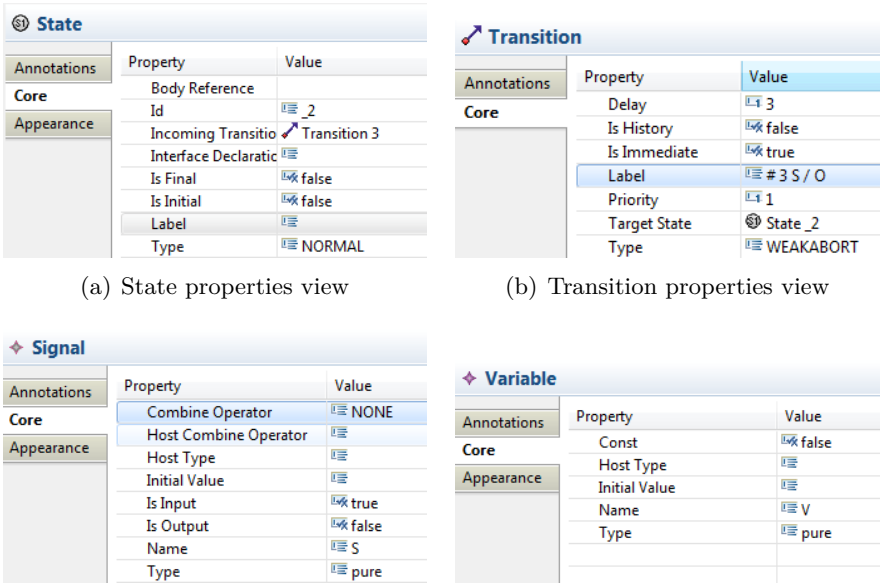
Properties view

The *properties view* allows the user to modify an element of the SyncChart (e.g. the user used a weak abortion transition and wants to change it to a strong abortion transition). Figure 4.8(a) represents the properties of a state. The properties view allows the manipulation of the Label and the Interface Declaration. The type of a state may be changed by choosing one of the following options: NORMAL, CONDITIONAL, REFERENCE, or TEXTUAL. The properties view also permits to set a state as initial or final.

The properties of a transition are depicted in Figure 4.8(b). The type of a transition may be selected via a combo-box: weak abortion, strong abortion, or normal termination. The Label property allows to define the transition action. The delay and immediate options of the transition action may also be set using the Delay and Is Immediate properties. The properties view also permits to set the transition to a history via the Is History property and to modify the transition priority via the Priority property.

As previously stated, a signal has a name, a direction, a type, an initial value, and a combine operator. All these options can be modified by the properties view (see Figure 4.8(c)).

Figure 4.8(d) represents the properties of a variable, which consist of a Name, a Type, an Initial Value, a Host Type, and a Const (Set to true if the variable should be a constant).



(a) State properties view

(b) Transition properties view

(c) Signal properties view

(d) Variable properties view

Figure 4.8: ThinKCharts Editor properties views

4.1.3 Validator

To statically validate a SyncChart, KIELER offers a mechanism for syntax and semantics checks. These permit to avoid inconsistent and/or erroneous models. Therefore, the editor provides some restrictions, which are represented in the form of validity rules. These are:

- Conditional states must not be initial or final.
- Conditional states must not contain regions.
- Conditional states must not contain signals.
- Conditional states must not contain actions.
- Conditional states must not contain a suspension trigger.
- Conditional states must have outgoing transitions.
- Only textual or reference states may contain body text.
- Every state needs at least one incoming transition.
- States need an ID.
- States within a region need to have different IDs.
- IDs should be of standard identifier type.
- A state can only have one outgoing normal termination.
- Signal name may not be empty.
- The interface declaration is not empty but there are no signals and variables defined.
- The root state should have at least one region containing at least one state.
- There should be exactly one root state which represents the whole SyncChart.
- Every region should have exactly one initial state.
- A hierarchical state with an outgoing normal termination has to contain at least one final state in every parallel region.
- Only signals in the root state may be global inputs or outputs.
- Only valued signals may have a combine type other than NONE.
- Priorities have to be at least 1.
- Immediate transitions must not have a delay.

- Delays have to be at least 1.
- Normal termination may not have a trigger.
- Normal termination must have the lowest priority.
- Strong aborts must have the highest priority.
- Simple states may not have a normal termination transition.
- Simple states cannot have strong aborts.
- Priorities of outgoing transitions need to be distinct.
- Transitions of conditionals must be immediate.
- SyncCharts do not support inter-level transitions.
- Pure signals cannot be assigned a value.
- Valued signals must be assigned a value.
- Trigger must have an effect.

4.1.4 Automatic layout

As mentioned in Section 1.2, KIELER provides automatic layout mechanisms to arrange the modeled diagrams elements. The ThinKCharts Editor uses the *dot*[1] layout [29], which is the ideal layout for the SyncCharts models. It also offers the option to apply the automatic layout after the model has been changed.

These were the main features that the ThinKCharts Editor offers. The following section introduces the SCCharts Editor.

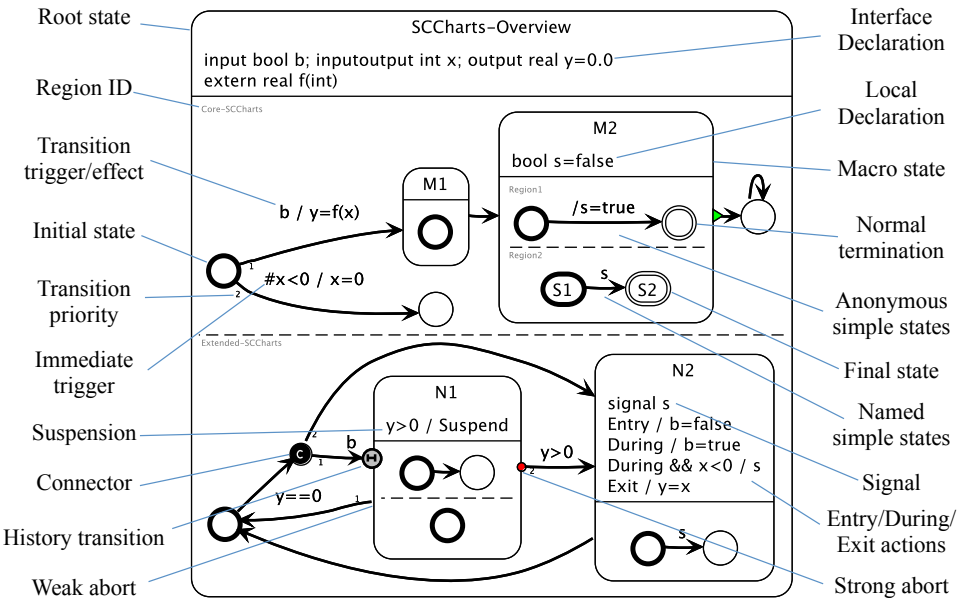
4.2 SCCharts Editor

The SCCharts syntax is introduced by von Hanxleden et al. in [34]. Valid SyncCharts are also valid SCCharts, which means that the SyncCharts syntax introduced in Section 4.1.1 must be supported by the SCCharts Editor.

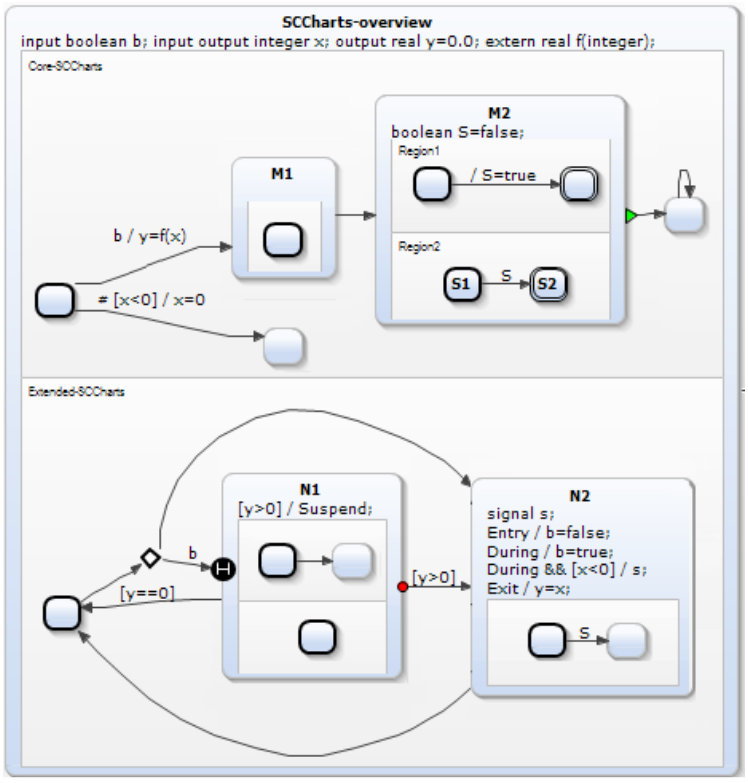
As mentioned in Section 2.2, SyncCharts and SCCharts syntactically only differ by variables. Variables in SCCharts may have a direction. Some examples are presented in the following:

```
input bool x
output real y=1.0
inputoutput int z
bool l
```

[1] http://www.graphviz.org/



(a) SCCharts-overview from [34]



(b) SCCharts-overview modeled using the SCCharts Editor based on YAKINDU SCT

Figure 4.9: Overview of SCCharts syntax

x and y are respectively input and output variables. y has initially the value 1.0. The variable z is an input and output variable at the same time. l is a local variable.

Figure 4.9 shows the SCCharts-overview example introduced in [34] and its modeling using the SCCharts Editor based on YAKINDU SCT.

The following section deals with the features provided by the YAKINDU SCT Editor.

4.3 YAKINDU SCT Editor

YAKINDU SCT is a software tool that allows the development of embedded systems. It is also a framework that allows to define tailored Statechart dialects. More details about the semantics of Statecharts are in [16]. YAKINDU SCT provides an editor, a simulator, and a code generator (see Figure 4.10). The editor is relevant for this work. It is presented in this section, beginning with its provided elements which allow the modeling of Statecharts, followed by the Palette, the context menu, the assistant provider, the properties view, and finally the validator.

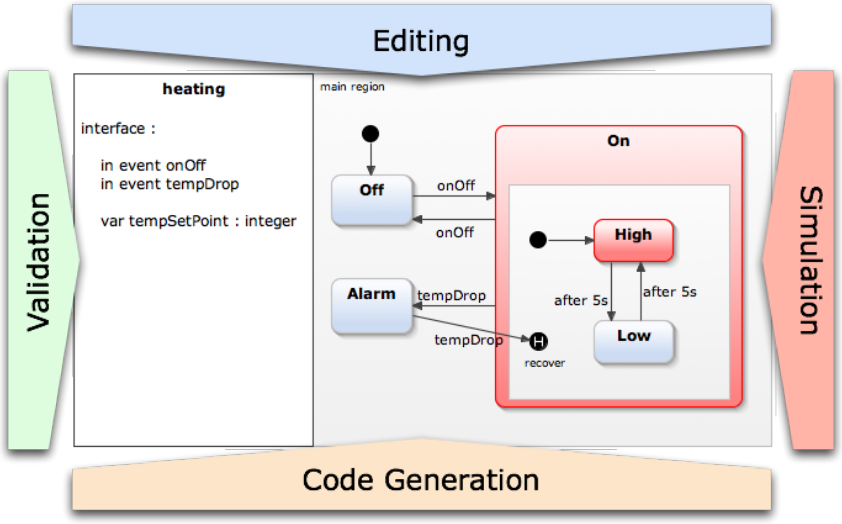


Figure 4.10: Overview of the YAKINDU SCT features[2]

4.3.1 Statecharts Elements

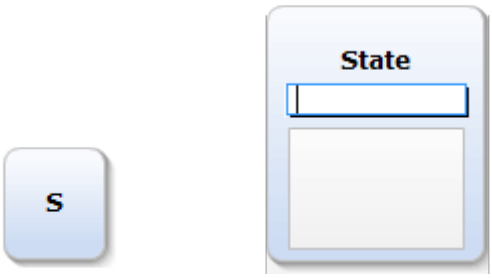
YAKINDU SCT Editor provides several elements for the modeling of Statecharts. This subsection shows how the elements are represented in the editor. For more information on the semantics see the YAKINDU SCT documentation.

[2] http://statecharts.org/documentation.html

Regions A region enables to group other Statecharts elements in a *state*. A state may have an arbitrary number of regions. Every region has a priority and an optional name. The state On in Figure 4.10 contains a region, which includes other elements.

States There are two kinds of states in YAKINDU SCT: *regular states* and *pseudo states*.

On the one hand, a regular state can be a normal state (A normal state will just be called state), or a *final state*. A *state* is represented by a rounded rectangle. Figure 4.11 shows two overviews of states. The difference is that the state on the right side contains a region, whereas the state on the left side contains no regions. A state is composed of three main compartments. The top compartment contains the name of the state. Note that a name must be set. The middle compartment is the text compartment, which enables the definition of *local reactions*. The bottom compartment may contain an arbitrary number of regions.



(a) A state without regions (b) A state having a region

Figure 4.11: YAKINDU SCT states

A *final state* cannot have outgoing transitions and must have one incoming transition. The incoming transition may carry *events* and *actions*. Figure 4.12 shows how final states are drawn in YAKINDU SCT.

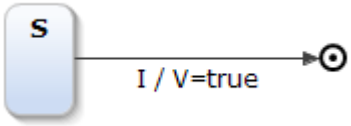


Figure 4.12: YAKINDU SCT final state

On the other hand, a pseudo state may be an *initial state*, a *choice*, or a *junction*. A pseudo state does not contain other elements. It has no name and

may not define *local reactions* compared with a state. An initial state must have one outgoing transition and should not have incoming transitions. The outgoing transition may not carry *events* and *actions*. Figure 4.13 depicts a Statechart, which consists of an initial state, a state, and a transition connecting them. The initial state is represented by a filled black circle. The *choice* and the *junction* elements are presented later.

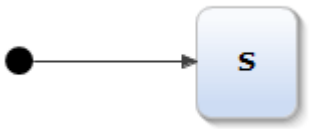


Figure 4.13: YAKINDU SCT initial state

Transitions A transition connects two elements, which can be a *state*, an *initial state*, a *final state*, a *choice*, a *junction*, a *composite state*, a *shallow history*, or a *deep history*. YAKINDU SCT supports only one type of transition. A transition can carry *events* and *actions*. It has also a priority. The transition that was modeled first has the higher priority.

Choice A choice is a pseudo state. It can have an arbitrary number of incoming transitions and outgoing transitions. It is drawn as a diamond without filling (see Figure 4.14).

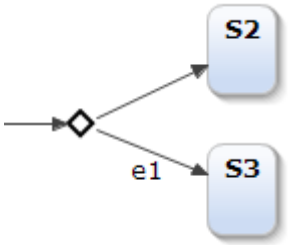


Figure 4.14: YAKINDU SCT choice

Junction A junction is a pseudo state. It permits to join transitions to improve the arrangement of the Statecharts. Figure 4.14 depicts the graphical representation of a junction.

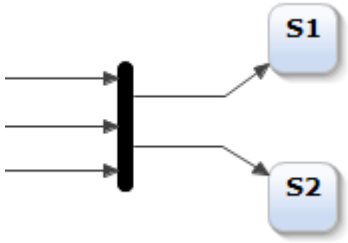
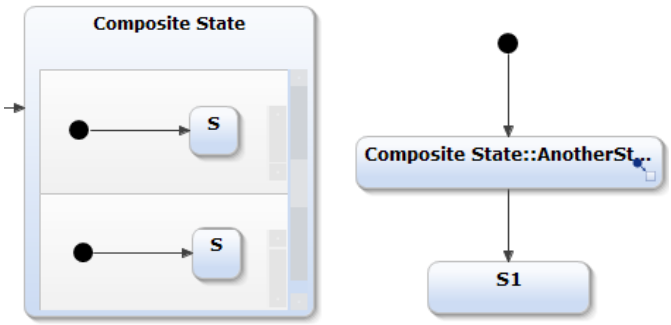


Figure 4.15: YAKINDU SCT junction

Composite state A composite state is a state that contains other Statecharts. On the one hand, a composite state may contain *orthogonal states* as depicted in Figure 4.16(a). Each of the orthogonal states is included in a region and is independent from the other. On the other hand, a composite state can contain *submachine states*. A submachine state includes another entire Statechart diagram. Figure 4.16(b) represents a submachine state that includes a Statechart named AnotherStatechart.



(a) Orthogonal States (b) Submachine States

Figure 4.16: YAKINDU SCT composite state

Shallow history and deep history The shallow history and the deep history are pseudo states. They can be placed inside a region. Figure 4.17(a) represents a shallow history, which is drawn as a filled circle having the letter H. The deep history, shown in Figure 4.17(b), is also represented by a filled circle having the letter H followed additionally by the symbol *.

(a) Shallow history (b) Deep history

Figure 4.17: YAKINDU SCT history state

Scopes A scope allows to define a *namespace*, *interface scopes* and *internal scopes*. The namespace is used to qualify references to the Statechart. The interface scope enables the definition of *declarations* (*event declarations*, *variable declarations*, etc.) that can be shared with the environment. One global (without a name) and an arbitrary number of named interface scopes may be declared in a scope. The internal scope allows to define declarations, which are visible only for contained states.

In Figure 4.18 a scope example is presented. The scope has a namespace. It contains a global interface, a named interface, and an internal scope.

```
namespace MyNameSpace

interface:
   in event I
   out event O:boolean
   var V:integer=10

interface MyInterface:
   in  event I:integer
   out event O
   operation Op(A :boolean,B:integer): boolean

internal:
   event localE
   var localV:boolean=false
```

Figure 4.18: YAKINDU SCT scope

Declarations As previously stated, interface and internal scopes allow to define *declarations*. A declaration may be an *event declaration*, a *variable declaration*, an *operation*, or a *local reaction*.

Event declarations An *event declaration* must have an identifier (unique name). It may have a direction (input or output) and a type. The typesystem of YAKINDU SCT contains the following simple types: boolean, integer, real, boolean,

string, and void. Note that an event declaration defined by an interface scope must have a direction and an event declaration defined by an internal scope has no direction. Here are some examples of events declarations:

```
in event e1
out event e2
in event e3:boolean
local event e4
```

e1 and e2 are respectively input and output events. They have no values. e3 is an input event, which has the boolean type. e4 is a local event.

Variable declarations Like an event declaration, a *variable declaration* has an identifier. However, it has no direction, must have a type, and may be initialized. Some examples are presented in the following:

```
var V:boolean
var V:integer=10
```

Operations An *operation* consists of a name, multiple parameters (optional), and a return type.

```
operation Op(A:boolean): boolean
operation Op(A:boolean, B:integer, C:real): boolean
```

Local reactions A *local reaction* is composed of a *reaction trigger* and a *reaction effect*. It is declared as below:

```
LocalReaction: ReactionTrigger '/' ReactionEffect
```

Reaction triggers A *reaction trigger* may contain *regular events*, *time events*, *built-in events*, and boolean *expressions*.

```
ReactionTrigger: (Event ("," Event)*
        (=> '[' Expression ']')?) | '[' Expression ']'
```

- Regular events: A regular event is simply represented by an event.
- Time events: A time event is either an after or an every trigger.

  ```
  after time (unit)?
  after 5 s
  every time (unit)?
  every 100 ms
  ```

- Built-in events: A built-in event may be an *entry*, *exit*, or *always trigger*.
- Expressions: Expressions are represented below.

	Operator	Symbol
logical relations	less than	<
	equal or less than	<=
	greater than	>
	equal or greater than	>=
	equal	==
	not equal	!=
shift operators	shift left	<<
	shift right	>>
binary arithmetic operators	plus	+
	minus	−
	multiply	*
	divide	/
	modulo	%
unary arithmetic operators	positive	+
	negative	−
	complement	~

Table 4.1: YAKINDU SCT operators

Expressions Within expressions one may express logical expressions, bitwise arithmetic, arithmetic expressions, and bit shifting. The operators offered by YAKINDU SCT are represented in Table 4.1.

Assignment	Symbol
simple assignment	=
multiply and assign	*=
divide and assign	/=
calculate modulo and assign	%=
add and assign	+=
subtract and assign	−=
bitshift left and assign	<<=
bitshift right and assign	>>=
bitwise AND and assign	&=
bitwise XOR and assign	∧=
bitwise OR and assign	|=

Table 4.2: YAKINDU SCT assignments

Reaction Effect A *reaction effect* has a list of *statements*:

```
ReactionEffect:  Statement (';' Statement )* (';')?
Statement: Assignment | EventRaising | OperationCall
```

A statement may be an *assignment*, an *event raising*, or an *operation call*. YAKINDU SCT offers the assignments depicted in Table 4.2.

Event raising is expressed by the literal raise followed by an event. *Operation call* is performed by calling the already declared operation with the name and passing the parameters.

4.3.2 Tooling

Palette

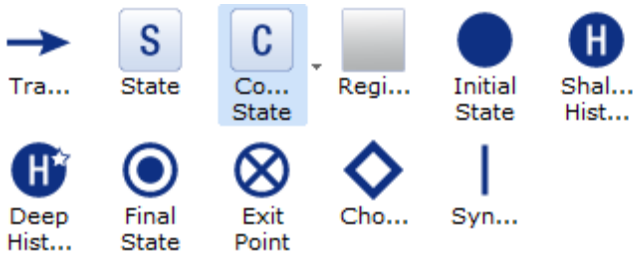


Figure 4.19: YAKINDU SCT palette

The palette provided by YAKINDU SCT offers all the graphical elements for the modeling of Statecharts: Transition, state, composite state (orthogonal state and submachine state), region, initial state, shallow history, deep history, final state, exit point, choice, and synchronization. The palette is shown in Figure 4.19.

An element may also be added to a Statechart using the *assistant provider*.

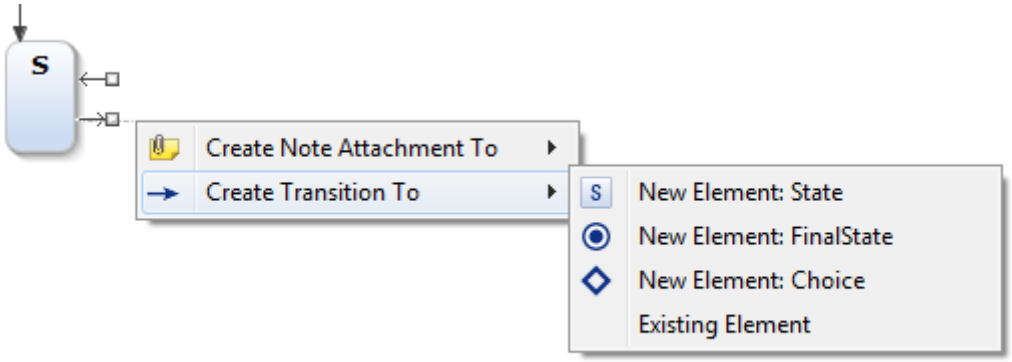
Assistant provider

The *assistant provider* used by YAKINDU SCT extends the GMF modeling assistant providers. It serves to contribute to the modeling of Statecharts by offering the possibility of adding several elements into the diagram.

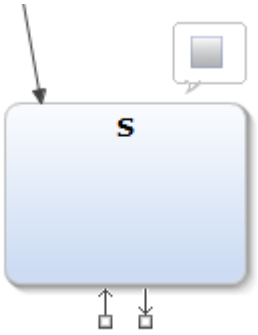
YAKINDU SCT offers two types of assistant providers:

Connection creation assistant The connection creation assistant allows to create an incoming or outgoing transition between two vertices. Vertices were introduced in Section 3.2.1 (A vertex is an element, which may have incoming and outgoing transitions like a state or a choice).

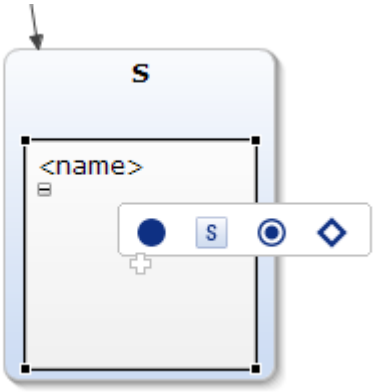
The connection creation assistant appears when hovering over a vertex. It can be used by choosing the incoming or outgoing connection handle and the connection to another vertex. By draging the connection to a region, the connection creation assistant offers additionally the possibility to create a new vertex or to use an existing one. Figure 4.20(a) represents the connection creation assistant that creates an outgoing transition with the option to create a new vertex.



(a) Connection creation assistant



(b) Pop-up bar on states



(c) Pop-up bar on regions

Figure 4.20: YAKINDU SCT assistant provider

Pop-up bar assistant provider YAKINDU SCT permits add elements using a pop-up bar. Figure 4.20(b) shows the pop-up bar that appears when hovering over a state. It allows the creation of regions inside. Figure 4.20(c) represents the pop-up bar that appears when hovering over a region, which allows the creation of vertices.

Properties view

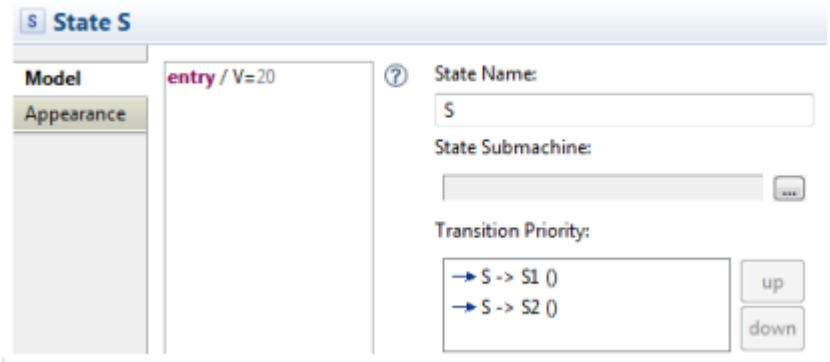


Figure 4.21: YAKINDU SCT state properties view

The YAKINDU SCT Editor allows the modification of elements properties through a properties view. The user can manipulate names, transition reactions, states local reactions, and more other properties. Figure 4.21 shows the properties view of a state. The text box on the left side serves to enter the local reaction. Besides the possibility to manipulate the name of a state, the properties view enables to specify a submachine (in the case of a submachine state) and to define the outgoing transitions priorities by ordering them.

4.3.3 Validator

The YAKINDU SCT Editor offers several validity rules to avoid inconsistent and/or erroneous models. There are two kinds of validity rules: Validity rules for the graphical representation and validity rules for the textual language. Here are the validity rules for the graphical representation:

- A state should have a name.
- A node must be reachable.
- A final state should have no outgoing transitions.
- A state should have at least one outgoing transition.
- An initial entry should have no incoming transitions.

- An initial entry should have a single outgoing transition.
- An entry must not have more than one outgoing transition.
- Outgoing transitions from entries cannot have a trigger or guard.
- A choice must have at least one outgoing transition.
- A choice should have one outgoing default transition.

The validity rules for the textual language are:

- An action should have an effect.
- Entry and exit events are allowed as local reactions only.
- Only one default/unnamed interface is allowed.
- In/Out declarations are not allowed in internal scope.
- Local declarations are not allowed in interface scope.
- The evaluation result of a time expression must be of type integer.
- The evaluation result of a guard expression must be of type boolean.
- The nested assignment of the same variable is not allowed.
- The 'void' type is invalid for variables.

4.4 Comparison

In the previous section, the features offered by the ThinKCharts Editor and the YAKINDU SCT Editor were presented. In order to implement a SyncCharts Editor based on YAKINDU SCT, several elements and features must be added or extended. This section aims to summarize the necessary changes to be made. Only changes that are relevant for the modeling of SyncCharts are considered in this work. The challenge is to make the fewest possible changes so that an update of the original editor does not require a lot of adjustments on the extended editor or better none at all.

The SyncCharts syntax must be fully supported by the new editor. The following subsection gives an overview of the required changes on YAKINDU SCT syntax.

4.4.1 Syntax

This subsection is divided into two parts. The first part compares the graphical representation of KIELER SyncCharts and YAKINDU SCT. The second part compares the textual description language.

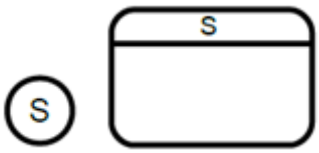
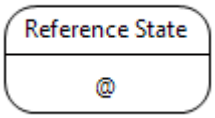
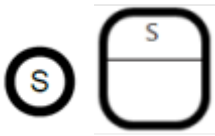
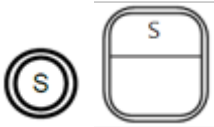
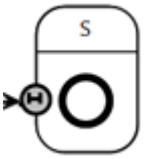
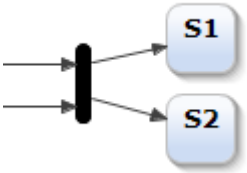
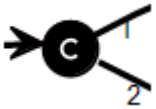
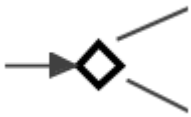
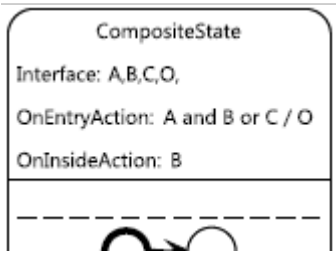
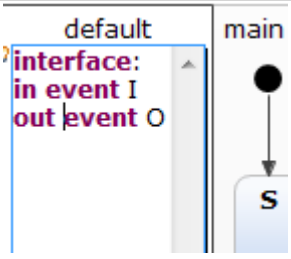
	ThinKCharts	YAKINDU SCT
Transition		
State		
Reference states		
Initial state		
Final state		
History		
Junction		
Choice		
Text compartment		

Table 4.3: Comparison of the graphical representation

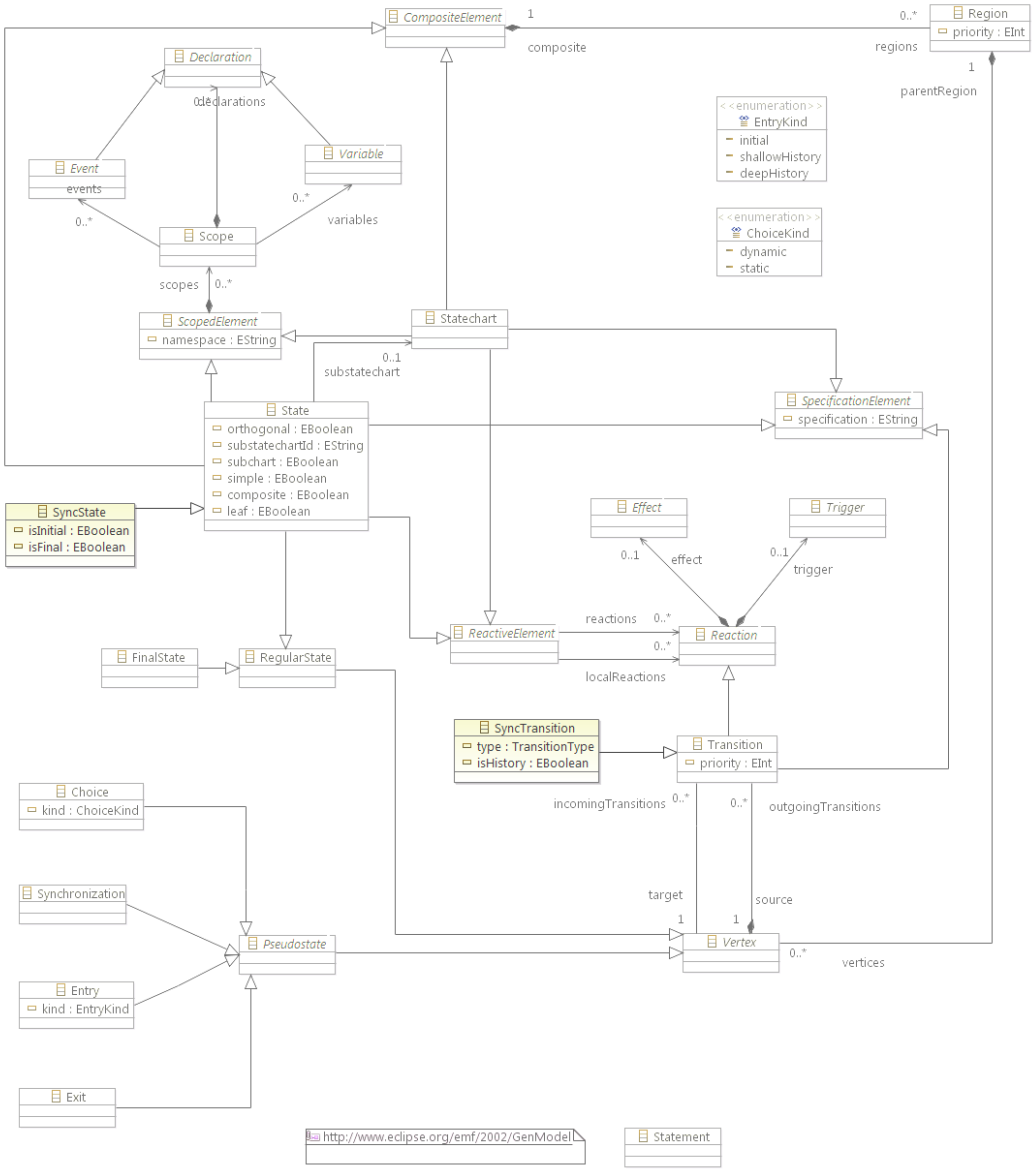


Figure 4.22: Extending the state and transition elements in the YAKINDU SCT Sgraph metamodel

Graphical representation

Table 4.3 summarizes the different graphical elements of both editors. It permits to deduce that the transition element must be extended in this work. YAKINDU SCT provides only one transition type. However, SyncCharts require three transition types (weak abortion, strong abortion, and normal termination). A transition may also be of type history. This option is not supported by YAKINDU SCT. Histories in the latter are pseudo states. The state element should also be adapted. A state should have the possibility to be set to an initial or final state (YAKINDU SCT initial and final states are no longer needed). Junctions have no meaning in SyncCharts, therefore they are ignored. The choice element looks different in YAKINDU SCT, which is here not deemed a sufficient reason for modifying it. YAKINDU SCT provides a text compartment for interface declarations. It is no longer required since interfaces must be declared into a state scope.

In order to adapt the state and transition elements, the Sgraph metamodel is extended (see Figure 4.22). This is explained in more details in Section 5.2.1.

These were the required modifications on the graphical representation. In the following, the textual description language is analysed.

Textual description language

The textual description language represents the elements that are defined using text compartments. Types, declarations, triggers, effects, expressions, and operations are the main elements of the textual description language. Table 4.4 gives a comparison of these elements. The KIELER SyncCharts features directly supported by YAKINDU SCT are marked in green and the features that are not supported are marked in red.

Events do not differ much from Signals. Besides the difference in the notation, a signal may have a double direction, e.g. input output signal S. The editor presented in this thesis uses events to emulate signals. Events are extended to adapt their notation and to be allowed to have a double direction. YAKINDU SCT provides a satisfactory type system. It is fully preserved in the new editor. Declarations, triggers, and effects should be adapted to SyncCharts needs. Expressions and operations should not be changed. However, the pre operator must be added.

	ThinKCharts	YAKINDU SCT
Types	• int • bool • float • double • pure (for pure signals) • unsigned • host	• integer • boolean • real • void • string
Declarations	• pure signals • valued signals • variables	• events • variables
Triggers	• immediate • delay • expression	• regular event • time event • built-in event • expression
Effects	• signal emitting • variable assignment	• assignment • event raising • operation call
Expressions	• logical expressions • arithmetic expressions	• logical expressions • arithmetic expressions • bitwise arithmetic • bit shifting

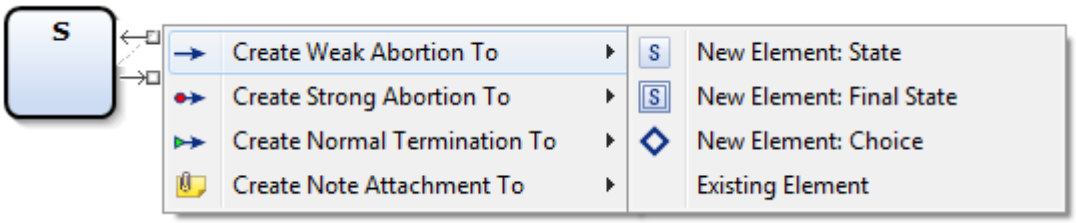
Operations	• compare operators • logical operators • arithmetic operators • pre operator (pre(I)) • value operator (?I)	• compare operators • logical operators • arithmetic operators • bitwise operators • bit shifting operators • value operator

Table 4.4: Comparison of the textual description language. The KIELER features directly supported by YAKINDU are marked in green and the features that are not supported are marked in red

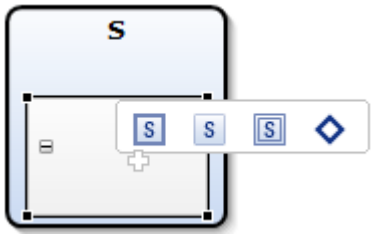
4.4.2 Tooling

Palette

The palette offered by the SyncCharts Editor based on YAKINDU SCT contains only elements that permit the modeling of SyncCharts. The elements are: weak abortion transition, strong abortion transition, normal termination transition, state, initial state, final state, region, and choice. Note that initial and final states are simply states with a predefined initial or final option.



(a) Connection creation assistant



(b) Pop-up bar on regions

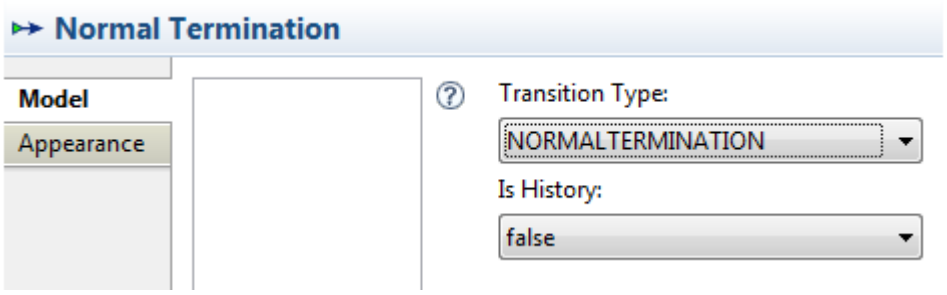
Figure 4.23: The extended assistant provider

Assistant provider

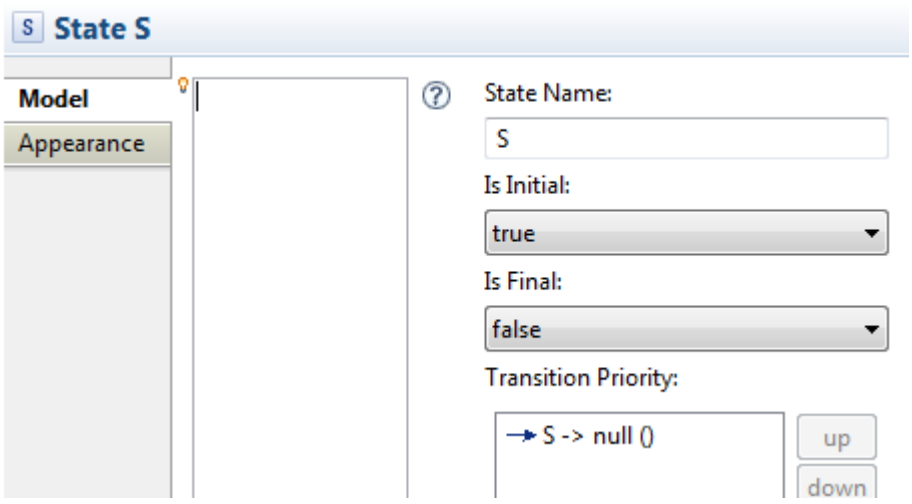
The YAKINDU SCT Editor offers his own mechanism for the Structure-Based Editing. The modeling assistant provider as the pop-up balloons provided by the KIELER framework allows to add several elements into the diagram. It is used by the editor implemented by this work and adapted for the SyncCharts syntax.

Connection creation assistant The extended connection creation assistant allows to create the three types of transitions. The vertices that this feature may create are states, initial states, final states, or choices. Figure 4.23(a) depicts the extended connection creation assistant.

Pop-up bar assistant provider It permits to add a region to a state. It also allows to add a state, an initial state, a final state, or a choice to a region. Figure 4.23(b) represents the pop-up bar implemented by this work.



(a) Transition properties view



(b) State properties view

Figure 4.24: Extended properties views

Properties view

As mentioned in Section 4.4.1, transition and state elements are derived by this work. Therefore, the properties views of these elements should be adapted:

- Adapting the transition properties view to allow the user to change the transition type and to set the transition as history.
- Adapting the state properties view permits to define a state as initial, final, or both.

Figure 4.24 shows the properties views of transitions and states implemented by this work.

4.4.3 Validator

YAKINDU SCT offers several validity rules, which are used by the editor implemented by this work. Since there are elements that were added or modified, new validity rules should be defined and several rules should be overridden. The ThinKCharts Editor validity rules are taken into account in the new editor.

The validity rules that are added by this work are presented below:

- A diagram should have exactly one region.
- The root region should have exactly one state.
- The root state may not be an initial state.
- The root state may not be a final state.
- The root state may not have incoming/outgoing transitions.
- Simple states may not have a normal termination transition.
- A state can only have one outgoing normal termination.
- A macro state with an outgoing normal termination has to contain at least one final state in every parallel region.
- Every region should have exactly one initial state.
- A normal termination may not have a trigger.
- Simple states cannot have strong aborts
- Inter-level transitions are forbidden.
- Every choice needs an incoming transition.
- A trigger should be a signal or a variable of type boolean.
- Cannot assign a value to a pre operator.

4.4.4 Automatic layout

The automatic layout provided by KIELER is used by the SyncCharts Editor as well as the SCCharts Editor based on YAKINDU SCT to arrange models. Figure 4.26 shows the ABRO example modeled using the SyncCharts Editor after and before applying the automatic layout.

4.4.5 White spaces

The diagrams modeled using YAKINDU SCT are not compact enough. They contain white spaces, which are reduced by this work as shown in Figure 4.26. White spaces may be reduced as bellow:

- Eliminating the white space under a state name.
- Reducing the font size of labels and text compartments.
- Minimizing the vertical spacing inside a state.

4.5 Project design

This chapter provided the requirements for the implementation of the new editor. In order to implement a generic editor based on YAKINDU SCT, a global project extends the Sgraph and Stext metamodels to implement the Syncgraph and Synctext metamodels, as depicted in Figure 4.25. Since, SyncCharts and SCCharts syntactically only differ by variables, the SyncCharts Editor and SCCharts Editor use together the Syncgraph metamodel for the graphical representation of their elements. Each editor extends the Synctext metamodel to implement the variable definition according to their syntax.

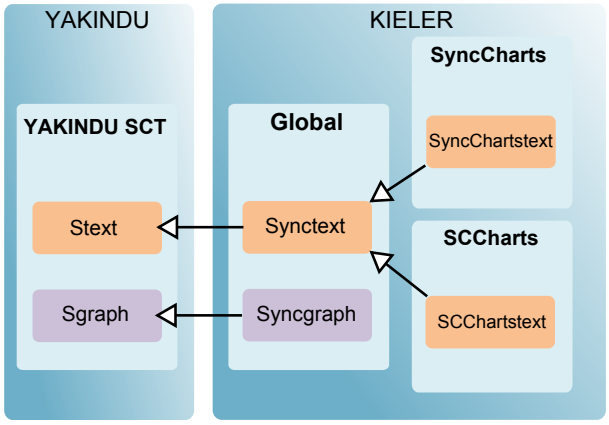
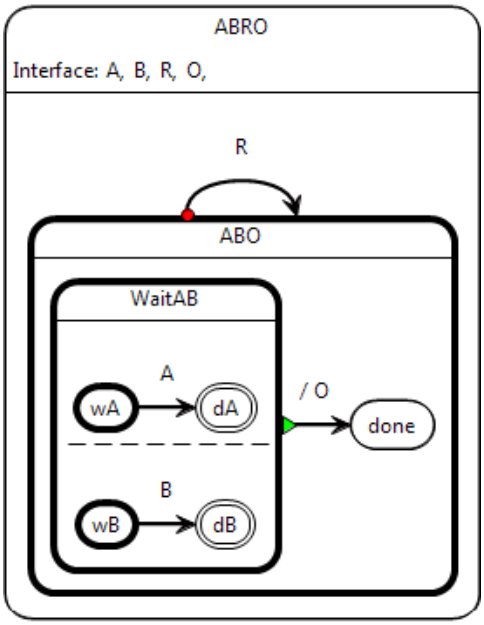
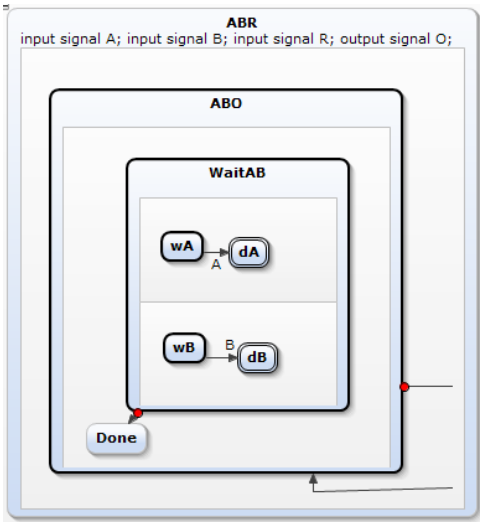


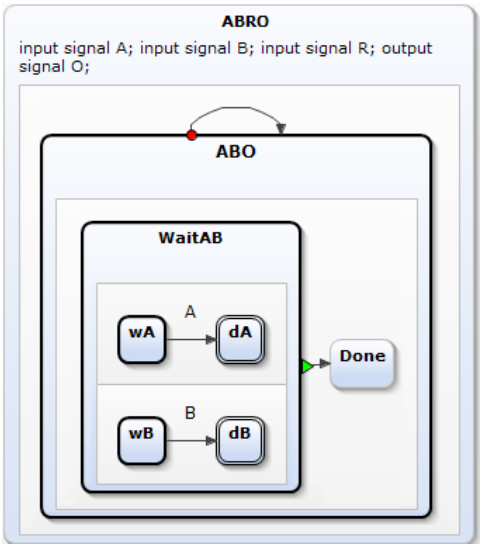
Figure 4.25: An abstract overview of the project design and the metamodels dependencies



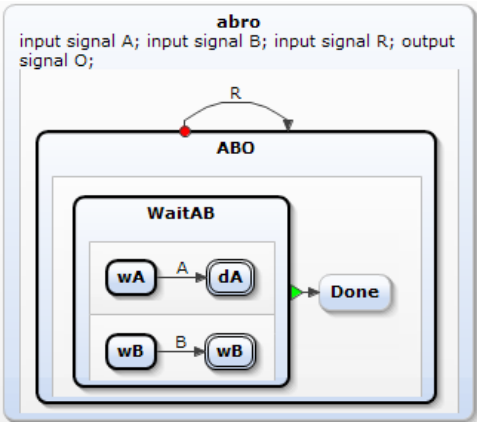
(a) using the GMF based ThinKCharts Editor



(b) before automatic layout



(c) after automatic layout and before reducing white spaces



(d) after automatic layout and reducing white spaces

Figure 4.26: The ABRO example using the ThinKCharts Editor and the SyncCharts Editor based on YAKINDU SCT before and after applying the automatic layout and reducing white spaces

5 Implementation

This chapter consists of two parts. The first part presents the contribution of Itemis AG in this work. The second part introduces the implementation of the SyncCharts Editor as well as the SCCharts Editor.

5.1 The contribution of Itemis AG

Itemis AG provided technical support for the creation of a new extended YAKINDU SCT Editor. They also adapted the YAKINDU SCT Editor to improve its extensibility.

5.1.1 Create a new extended YAKINDU SCT Editor

Creating a derived YAKINDU SCT Editor can be divided into three parts. The first part creates a new editor plug-in. The second part creates a new Xtext project for the definition of the SyncCharts grammar. In the third part, a new plug-in is created for the integration of the SyncCharts grammar into the editor implemented at the first part. The different parts are presented in the following.

Editor

This part creates the editor plug-in, which extends the editor plug-in of the YAKINDU SCT project. Extension points are also extended in order to register the editor.

- Create a new Eclipse plug-in: The first step is to create a new plug-in named de.cau.cs.kieler.yakindu.syncharts.ui.editor for the case of the SyncCharts Editor.
- Create a folder obj16 for icons: The folder contains additional icons (e.g., editor logo and palette icons like strong abortion icon).
- Add the require-Bundle org.yakindu.sct.ui.editor to the project: This plug-in implements the YAKINDU SCT Editor. It is required in order to be extended.
- Create the de.cau.cs.kieler.yakindu.syncharts.ui.editor.editor package, which contains the following classes: SyncChartsDiagramEditor, and SyncChartsDiagramActionBarContributor.

– The SyncChartsDiagramEditor class extends StatechartDiagramEditor. It defines the editor identifier:

```
public static final String ID =
"de.cau.cs.kieler.yakindu.ui.editor.editor.SyncChartsDiagramEditor";
```

It also overrides the method getContributorId() in order to return the new editor identifier:

```
@Override
public String getContributorId() {
   return ID;
}
```

– The SyncChartsDiagramActionBarContributor class extends DiagramActionBarContributor. It overrides the getEditorId() as well as the getEditorClass() method to return the extended editor identifier and the SyncChartsDiagramEditor class.

```
@Override
protected String getEditorId() {
   return SyncChartsDiagramEditor.ID;
}

@Override
protected Class<SyncChartsDiagramEditor> getEditorClass() {
   return SyncChartsDiagramEditor.class;
}
```

It also overrides the init() method to fix a saving problem in Eclipse Indigo.

```
@Override
public void init(IActionBars bars) {
   super.init(bars);
   //workaround for
   //     https://bugs.eclipse.org/bugs/show_bug.cgi?id=346648
   bars.setGlobalActionHandler(GlobalActionId.SAVE, null);
}
```

- Register the editor: In order to register the editor, the following lines extend four extension points. The Editors extension point registers the editor information (e.g., name, icon, and extensions). The propertyContributor[1] enables the tabbed properties view. The propertyTabs describes the tabs for a contributor. The propertySections describes the sections for a contributor. It specifies for each Statecharts element whose class implements its *editpart*. Editparts are implemented by GEF. They are responsible for applying changes to the model in the editor. The extensions are defined in the plugin.xml presented in Listing 5.1.

[1] http://www.eclipse.org/articles/Article-Tabbed-Properties/tabbed_properties_view.html

Listing 5.1: Extending org.yakindu.sct.ui.editor Extension Points

```
<!-- Editor -->
<extension point="org.eclipse.ui.editors">
<editor
   class="de.cau.cs.kieler.yakindu.syncchartsui.
         editor.editor.SyncChartsDiagramEditor"
   contributorClass="de.cau.cs.kieler.yakindu.syncchartsui.
         editor.editor.SyncChartsDiagramActionBarContributor"
   default="true"
   extensions="ysc"
   icon="icons/obj16/logo-16.png"
   id="de.cau.cs.kieler.yakindu.syncchartsui.
         editor.editor.SyncChartsDiagramEditor"
   matchingStrategy="org.eclipse.gmf.runtime.diagram.ui.resources.
         editor.parts.DiagramDocumentEditorMatchingStrategy"
   name="YAKINDU SyncCharts Editor">
</editor>
</extension>

<extension point="org.eclipse.ui.views.properties.tabbed.propertyContributor"
   id="prop-contrib">
   <propertyContributor
      contributorId="de.cau.cs.kieler.yakindu.syncchartsui.
            editor.editor.SyncChartsDiagramEditor"
      labelProvider="org.yakindu.sct.ui.editor.
            propertysheets.SheetLabelProvider">
      <propertyCategory category="domain"/>
      <propertyCategory category="visual"/>
      <propertyCategory category="extra"/>
   </propertyContributor>
</extension>

<extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs"
   id="proptabs">
   <propertyTabs
      contributorId="de.cau.cs.kieler.yakindu.syncchartsui.
            editor.editor.SyncChartsDiagramEditor">
      <propertyTab
         category="domain"
         id="property.tab.domain"
         label="Model"/>
      <propertyTab
         category="visual"
         id="property.tab.AppearancePropertySection"
         label="Appearance"/>
      <propertyTab
         category="visual"
         id="property.tab.DiagramPropertySection"
         label="Diagram"/>
   </propertyTabs>
</extension>

<extension point="org.eclipse.ui.views.properties.tabbed.propertySections"
   id="propsections">
   <propertySections
      contributorId="de.cau.cs.kieler.yakindu.syncchartsui.
            editor.editor.SyncChartsDiagramEditor">
   <!-- State model section -->
      <propertySection
         id="property.section.domain.state"
         tab="property.tab.domain"
         class="org.yakindu.sct.ui.editor.propertysheets.
```

```
62                StatePropertySection">
63          <input type="org.yakindu.sct.ui.editor.editparts.
64                StateEditPart"/>
65          <input type="org.yakindu.sct.ui.editor.editparts.
66                StateNameEditPart"/>
67          <input type="org.yakindu.sct.ui.editor.editparts.
68                StateTextCompartmentEditPart"/>
69          <input type="org.yakindu.sct.ui.editor.editparts.
70                StateTextCompartmentExpressionEditPart"/>
71       </propertySection>
72    <!-- Statechart model section -->
73       <propertySection
74          id="property.section.domain.statechart"
75          tab="property.tab.domain"
76          class="org.yakindu.sct.ui.editor.propertysheets.
77                StatechartPropertySection">
78          <input type="org.yakindu.sct.ui.editor.editparts.
79                StatechartNameEditPart"/>
80          <input type="org.yakindu.sct.ui.editor.editparts.
81                StatechartTextEditPart"/>
82          <input type="org.yakindu.sct.ui.editor.editparts.
83                StatechartTextExpressionEditPart"/>
84       </propertySection>
85    <!-- Transition model section -->
86       <propertySection
87          id="property.section.domain.transition"
88          tab="property.tab.domain"
89          class="org.yakindu.sct.ui.editor.propertysheets.
90                TransitionPropertySection">
91          <input type="org.yakindu.sct.ui.editor.editparts.
92                TransitionEditPart"/>
93          <input type="org.yakindu.sct.ui.editor.editparts.
94                TransitionExpressionEditPart"/>
95       </propertySection>
96    <!-- Entry model section -->
97       <propertySection
98          id="property.section.domain.entry"
99          tab="property.tab.domain"
100         class="org.yakindu.sct.ui.editor.propertysheets.
101               EntryPropertySection">
102         <input type="org.yakindu.sct.ui.editor.editparts
103               .EntryEditPart"/>
104      </propertySection>
105   <!-- Exit model section -->
106      <propertySection
107         id="property.section.domain.exit"
108         tab="property.tab.domain"
109         class="org.yakindu.sct.ui.editor.propertysheets.
110               ExitPropertySection">
111         <input type="org.yakindu.sct.ui.editor.editparts.
112               ExitEditPart"/>
113      </propertySection>
114   <!-- Region model section -->
115      <propertySection
116         id="property.section.domain.region"
117         tab="property.tab.domain"
118         class="org.yakindu.sct.ui.editor.propertysheets.
119               RegionPropertySection">
120         <input type="org.yakindu.sct.ui.editor.editparts.
121               RegionEditPart"/>
122      </propertySection>
123   <!-- Choice model section -->
```

```
124       <propertySection
125          id="property.section.domain.choice"
126          tab="property.tab.domain"
127          class="org.yakindu.sct.ui.editor.propertysheets.
128                ChoicePropertySection">
129          <input type="org.yakindu.sct.ui.editor.editparts.ChoiceEditPart"/>
130       </propertySection>
131    </propertySections>
132 </extension>
```

- Create the de.cau.cs.kieler.yakindu.syncchartsui.editor.wizards package: This package implements the wizard that opens when creating a new diagram. It contains the SyncChartsCreationWizard class, which extends CreationWizard. This class defines an identifier for the wizard.

```
1 public static final String ID =
2 "de.cau.cs.kieler.yakindu.syncchartsui.editor.wizards." +
3 "SyncChartsCreationWizard";
```

  The class also overrides the addPages() method in order to implement the wizard page for the creation of a SyncCharts diagram. The method allows to set the wizard title and its description text.

```
1 @Override
2 public void addPages() {
3    modelFilePage = new CreationWizardPage("DiagramModelFile", getSelection(),
4    "ysc");
5    modelFilePage.setTitle("KIELER YAKINDU SyncCharts Diagram");
6    modelFilePage.setDescription("Create a new KIELER YAKINDU SyncCharts
7    Diagram File");
8    addPage(modelFilePage);
9 }
```

- Register the wizard: The wizard must be registered. Therefore, the following lines should be added to the plugin.xml presented in Listing 5.1:

```
1  <!-- New Diagram Wizard -->
2  <extension point="org.eclipse.ui.newWizards">
3     <wizard
4     category="KIELER"
5     class="de.cau.cs.kieler.yakindu.syncchartsui
6              .editor.wizards.SyncChartsCreationWizard"
7     icon="icons/obj16/logo-16.png"
8     id="de.cau.cs.kieler.yakindu.syncchartsui
9              .editor.wizards.SyncChartsCreationWizard"
10    name="KIELER YAKINDU SyncCharts Diagram"
11    project="false">
12 </wizard>
13 </extension>
```

Xtext

This part creates the Xtext project, which implements the SyncCharts grammar by extending the Stext grammar and adapting the elements already mentioned in Section 4.4.1.

- Create the Xtext project de.cau.cs.kieler.yakindu.syncharts.model.stext having the name de.cau.cs.kieler.yakindu.syncharts.model.stext.SyncExp and the extensions syncexp. Then, add the require-Bundle org.yakindu.sct.model.stext to allow the overriding of the YAKINDU SCT Stext grammar.

- Extend the Stext grammar by adding the following code to the syncexp.xtext file:

Listing 5.2: This code is a subset of the SyncCharts grammar. The latter is introduced in Section 5.2.

```
1  grammar de.cau.cs.kieler.yakindu.model.stext.Synctext
2  with org.yakindu.sct.model.stext.SText
3
4  generate synctext "http://kieler.cs.cau.de/yakindu/stext/"
5
6  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7  import "http://www.yakindu.org/sct/statechart/SText" as stext
8  import "http://www.yakindu.org/sct/sgraph/2.0.0" as sgraph
9  import "http://www.yakindu.org/base/types/2.0.0" as types
10
11 /******** INTERFACE DECLARATION ************** */
12 ////defines the possible scopes for a state
13 StateScope:
14    {SimpleScope} declarations+=(VariableDeclaration
15                          | SignalDeclaration
16                          | OperationDeclaration
17                          | LocalReaction)*;
```

- Modify the generation workflow to register the Stext grammar by adding the lines 4 to 7 to the StandaloneSetup in GenerateSyncexp.mwe2:

```
1  bean = StandaloneSetup {
2     scanClassPath = true
3     platformUri = "${runtimeProject}/.."
4     registerGeneratedEPackage = "org.yakindu.sct.model.stext.stext.
5                                  StextPackage"
6     registerGenModelFile = "platform:/resource/org.yakindu.sct.model.stext/
7                           model/SText.genmodel"
8  }
```

Integration

As mentioned in Chapter 3, a transition, a state, or a Statechart can contain further specifications. A specification specifies a textual definition in an element. This part allows the integration of the textual definition into the specification of an element. It gives the implementation for the case of a transition. The following classes should be created in the same way for the state and the Statechart specifications.

- Create an Eclipse plug-in de.cau.cs.kieler.yakindu.syncharts.ui.integration.stext and add the require-Bundles de.cau.cs.kieler.yakindu.syncharts.model.stext and de.cau.cs.kieler.yakindu.syncharts.model.stext.ui.

- Create the following package: de.cau.cs.kieler.yakindu.syncchart s.ui.integration.stext.parsers. This package integrates the parsers that parse transition, state, and Statechart specifications to check for the correct syntax and to create the internal representation form.

  – Create the TransitionAntlrParser that extends SyncExpParser:

```
1  package de.cau.cs.kieler.yakindu.syncchart s.ui.integration.stext.parsers;
2
3  import org.yakindu.sct.model.stext.stext.TransitionSpecification;
4  import de.cau.cs.kieler.yakindu.syncchart s.model.stext.
5        parser.antlr.SyncExpParser;
6
7  public class TransitionAntlrParser extends SyncExpParser {
8     @Override
9     protected String getDefaultRuleName() {
10       return TransitionSpecification.class.getSimpleName();
11    }
12 }
```

  – Create the TransitionContentAssistParser that extends SyncExpParser:

```
1  package de.cau.cs.kieler.yakindu.syncchart s.ui.integration.stext.parsers;
2
3  import java.util.Collection;
4  import org.antlr.runtime.RecognitionException;
5  import org.eclipse.xtext.ui.editor.contentassist.antlr.FollowElement;
6  import org.eclipse.xtext.ui.editor.contentassist.antlr.
7           internal.AbstractInternalContentAssistParser;
8  import de.cau.cs.kieler.yakindu.syncchart s.model.stext.ui.
9           contentassist.antlr.SyncExpParser;
10 import de.cau.cs.kieler.yakindu.syncchart s.model.stext.ui.
11          contentassist.antlr.internal.InternalSyncExpParser;
12
13 public class TransitionContentAssistParser extends SyncExpParser {
14    @Override
15    protected Collection<FollowElement> getFollowElements(
16          AbstractInternalContentAssistParser parser) {
17       try {
18          InternalSyncExpParser typedParser
19             = (InternalSyncExpParser) parser;
20          typedParser.entryRuleTransitionSpecification();
21          return typedParser.getFollowElements();
22       } catch (RecognitionException ex) {
23          throw new RuntimeException(ex);
24       }
25    }
26 }
```

- Create the following package: de.cau.cs.kieler.yakindu.syncchart s.ui.integration.stext.modules.

– Create the TransitionRuntimeModule that extends SyncExpRuntimeModule:

```
1  package de.cau.cs.kieler.yakindu.syncharts.ui.integration.stext.modules;
2
3  import org.eclipse.xtext.parser.IParser;
4  import de.cau.cs.kieler.yakindu.syncharts.model.
5        stext.SyncExpRuntimeModule;
6  import de.cau.cs.kieler.yakindu.syncharts.ui.integration.
7        stext.parsers.TransitionAntlrParser;
8
9  public class TransitionRuntimeModule extends SyncExpRuntimeModule {
10    @Override
11    public Class<? extends IParser> bindIParser() {
12       return TransitionAntlrParser.class;
13    }
14 }
```

– Create the package TransitionUIModule that extends SyncExpUiModule:

```
1  package de.cau.cs.kieler.yakindu.syncharts.ui.integration.stext.modules;
2
3  import org.eclipse.ui.plugin.AbstractUIPlugin;
4  import org.eclipse.xtext.ui.editor.contentassist.
5        antlr.IContentAssistParser;
6  import de.cau.cs.kieler.yakindu.syncharts.model.
7        stext.ui.SyncExpUiModule;
8  import de.cau.cs.kieler.yakindu.syncharts.ui.integration.
9        stext.parsers.TransitionContentAssistParser;
10
11 public class TransitionUIModule extends SyncExpUiModule {
12    public TransitionUIModule(AbstractUIPlugin plugin) {
13       super(plugin);
14    }
15
16    @Override
17    public Class<? extends IContentAssistParser>
18          bindIContentAssistParser() {
19       return TransitionContentAssistParser.class;
20    }
21 }
```

- Create the following class in the default package:

```
1  package de.cau.cs.kieler.yakindu.syncharts.ui.integration.stext;
2
3  import org.yakindu.sct.ui.editor.extensions.AbstractExpressionsProvider;
4  import com.google.inject.Module;
5  import de.cau.cs.kieler.yakindu.syncharts.ui.integration.
6       stext.modules.TransitionRuntimeModule;
7  import de.cau.cs.kieler.yakindu.syncharts.ui.integration.
8       stext.modules.TransitionUIModule;
9
10 public class TransitionExpressionProvider
11         extends AbstractExpressionsProvider {
12
13   @Override
14   protected Module getRuntimeModule() {
15      return new TransitionRuntimeModule();
16   }
17
```

```
18      @Override
19      protected Module getUIModule() {
20          return new TransitionUIModule(Activator.getDefault());
21      }
22  }
```

The YAKINDU SCT Editor was extended to create a new editor. The derived editor extends the Stext metamodel presented in Section 3.2.2. It allows to modify the editparts, the palette, the properties view, and the assistant provider using extension points. However, it is not possible to extend the Sgraph metamodel in order to implement a derived SyncCharts transition and state. During this work, Itemis extended their framework to allow for deriving specific types in the Sgraph model. The modifications are discussed in the following.

5.1.2 Yakindu extensions, by Itemis AG

Itemis AG allowed the adaption of the Sgraph using extension points. The following lines may be added to the plugin.xml to integrate an extended metamodel into the editor. The extended Sgraph is named Syncgraph in the SyncCharts Editor based on YAKINDU.

```
1  <extension point="org.eclipse.emf.ecore.generated_package">
2      <package
3          uri="http://kieler.cs.cau.de/yakindu/sccharts/"
4          class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncgraphPackage"
5          genModel="model/syncgraph.genmodel"/>
6  </extension>
```

The extended elements should be bound to the editor. In order to do that, Itemis AG improved the editor API by providing a *Module* class org.yakindu.sct.ui.editor.module.-SCTModul. The Module class provides default bindings for the metamodel, the palette, the Diagram Initializer, and the editparts. This is done with Google Guice, which was introduced in Section 3.1.5. The diagram initializer specifies the default Statecharts elements by creating a new diagram. The code in Listing 5.3 depicts the relevant subset of the SCTModul class to implement the SyncCharts Editor. The methods may be overridden to return the appropriate classes, e.g., create a DefaultSyncPaletteFactory class, which defines the palette elements, and override the getPaletteFactory() method to return the new class. The getFileExtension() method in line 33 can be overridden to return the extended editor file extension. The getContributorId() method in line 44 can be overridden to change the property page. The getTranisitionEditPart() (line 48) may be overridden to extend the editpart of a transition.

Listing 5.3: A subset of the SCTModul class

```
1   /**
2    * returns an implementation if {@link IMetaModelTypeFactory} that registers
3    * the default statechart {@link IElementType}s. Override if you want to
4    * contribute custom element types
5    *
6    */
7   protected Class<? extends IMetaModelTypeFactory> getMetaModelTypeFactory() {
8      return StatechartMetaModelTypeFactory.class;
9   }
10
11  /**
12   * returns an implementation of {@link ISCTPaletteFactory} that registeres
13   * the default palette entries. Override if you want to add or remove
14   * palette entries.
15   */
16  protected Class<? extends ISCTPaletteFactory> getPaletteFactory() {
17     return DefaultSCTPaletteFactory.class;
18  }
19
20  /**
21   * returns an implementation of {@link IDiagramInitializer} that initializes
22   * new created diagrams.
23   *
24   * @return
25   */
26  protected Class<? extends IDiagramInitializer> getDiagramInitializer() {
27     return DefaultDiagramInitializer.class;
28  }
29
30  /**
31   * Returns the default file extension for diagrams.
32   */
33  protected String getFileExtension() {
34     return "sct";
35  }
36
37  /**
38   * Override the property sheet id if you want to contribute your own
39   * property sheets via
40   * org.eclipse.ui.views.properties.tabbed.propertyContributor extension
41   * point
42   *
43   */
44  protected String getContributorId() {
45     return "org.yakindu.sct.ui.editor.editor.StatechartDiagramEditor";
46  }
47
48  protected Class<? extends IGraphicalEditPart> getTransitionEditPart() {
49     return TransitionEditPart.class;
50  }
```

The support provided by Itemis AG made it possible to implement an extended SyncCharts Editor. The following section presents the implementation details.

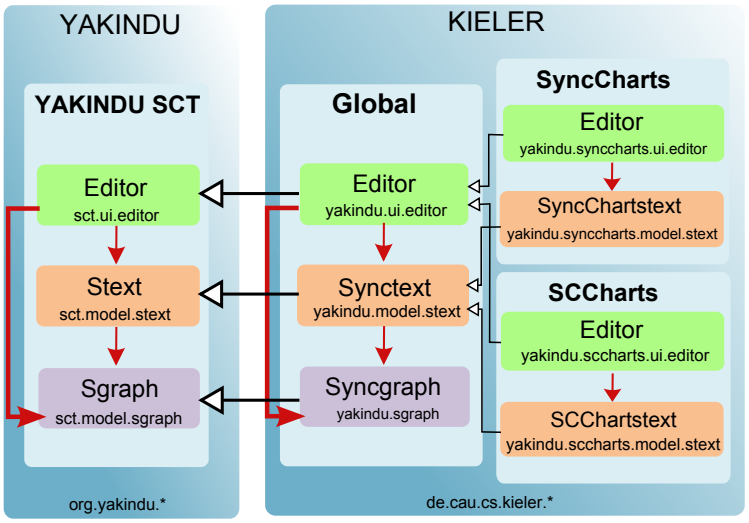


Figure 5.1: This figure gives a detailed overview of the project design shown in Figure 4.25. It also shows the projects names. The red arrows show the dependencies between the sub-projects

5.2 Using YAKINDU SCT extension mechanisms for implementing a SyncCharts Editor

As mentioned in Section 1.4, the aim of this thesis is to implement a SyncCharts Editor and a SCCharts Editor based on YAKINDU SCT. Figure 5.1 shows the design of the project and gives an overview of the dependencies between its sub-projects. This work proposes a generic approach for the implementation of both editors. The Global project contains the elements used in common. It extends the Sgraph metamodel to implement the Syncgraph metamodel. Since SyncCharts and SCCharts have the same graphical representation, Syncgraph is used together by both editors. The Global project also extends the Stext metamodel to implement the Synctext metamodel. The latter is extended by the SyncCharts Editor and the SCCharts Editor to add special requirements. Syntactically, SyncCharts and SCCharts differ by variables. The editor implemented by the Global project extends the palette, the diagram initializer, the editparts, the properties view, and the modeling assistant provider.

This section is divided into five parts. The first part presents the Sgraph project, which contains the Syncgraph metamodel. The second part gives an overview of the Stext projects, which implement the Synctext metamodel as well as the SyncChartstext and the SCChartstext. The third part presents the implementation of the Editor projects. The fourth part implements the automatic layout to arrange the modeled diagrams. The last part gives an overview of reducing white spaces.

5.2.1 The Sgraph project

The Sgraph project (de.cau.cs.kieler.yakindu.sgraph) implements the Syncgraph metamodel as well as the validator for the graphical representation.

Metamodel

The Syncgraph metamodel shown in Figure 5.2 extends the Sgraph metamodel to add two elements: *SyncTransition* and *SyncState*. A SyncTransition extends the Sgraph transition. It has additionally the attributes type and isHistory. The type may be one of the following options: WEAKABORT, STRONGABORT, or NORMALTERMINATION. The isHistory attribute is of type boolean. It is set to true if the transition is a history transition. A SyncState extends the state already existing in the Sgraph metamodel. It adds the isInitial and isFinal attributes of type boolean, which reflect respectively if a state is an initial or final state.

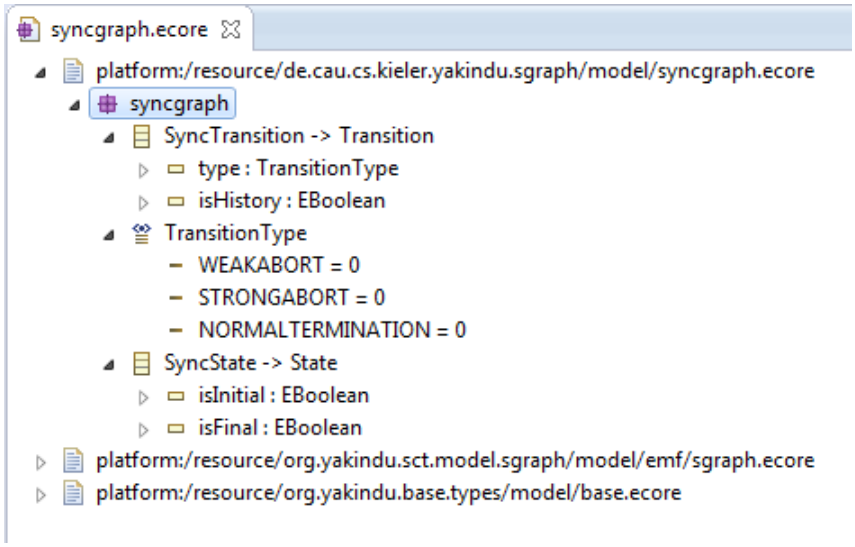


Figure 5.2: The Syncgraph metamodel

Validator for the graphical representation

The validator package de.cau.cs.kieler.yakindu.sgraph.validator implements the validity rules for the graphical representation presented in Section 4.4.3. It contains the SyncGraphJavaValidator class, which extends the SGraphJavaValidator (The YAKINDU SCT Sgraph validator). The following code is a subset of the SyncGraphJavaValidator class. It implements the outgoingTransitionCount() validity rule, which verifies that a final state has no outgoing transitions. The method tests if the state is a final state (line 9). In the case of a final state, it returns a warning if the number of outgoing transitions is greater than zero (lines 10 - 13).

```
1  public static final String ISSUE_FINAL_STATE_OUTGOING_TRANSITION
2     = "A final state should have no outgoing transition.";
3
4  /**
5   * Verify that a final state has no outgoing transitions
6   */
7  @Check(CheckType.FAST)
8  public void outgoingTransitionCount(SyncState finalState) {
9     if (finalState.isIsFinal()) {
10       if ((finalState.getOutgoingTransitions().size() > 0)) {
11          warning(ISSUE_FINAL_STATE_OUTGOING_TRANSITION, finalState,
12                null, -1);
13       }
14    }
15 }
```

The SyncGraphJavaValidator implements only the validity rules for the graphical representation. The validator for the textual language is presented with the Synctext, SyncChartstext, and SCChartstext metamodels in the following.

5.2.2 The Stext project

The Stext project implements the elements of the textual description language that are not supported by YAKINDU SCT, which are introduced in Section 4.4.1. The Stext project is composed of three projects. The Global project (de.cau.cs.kieler.yakindu.model.stext) contains the Synctext metamodel, which extends the Stext metamodel and the global validator for the textual language. The de.cau.cs.kieler.yakindu.syncharts.model.stext and de.cau.cs.kieler.yakindu.scharts.model.stext projects contain respectively the SyncChartstext and SCChartstext metamodels. The Stext sub-projects are presented in the following.

de.cau.cs.kieler.yakindu.model.stext

The de.cau.cs.kieler.yakindu.model.stext, shown in Figure 5.1, is an Xtext project. It implements the Synctext metamodel and the global validator for the textual language. The validator is introduced in Section 5.2.2.

The Synctext extends the Stext grammar, which can be found in the plug-in org.yakindu.sct.model.stext. The code in Listing 5.4 shows the adaption of the YAKINDU SCT grammar to a grammar that can be used in common by the SyncCharts Editor and the SCCharts Editor.

Listing 5.4: The Synctext grammar

```
1  grammar de.cau.cs.kieler.yakindu.model.stext.Synctext
2  with org.yakindu.sct.model.stext.SText
3
4  generate synctext "http://kieler.cs.cau.de/yakindu/stext/"
5
6  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
7  import "http://www.yakindu.org/sct/statechart/SText" as stext
8  import "http://www.yakindu.org/sct/sgraph/2.0.0" as sgraph
9  import "http://www.yakindu.org/base/types/2.0.0" as types
```

```
10
11  /******** INTERFACE DECLARATION ************** */
12  ////defines the possible scopes for a state
13  StateScope:
14     {SimpleScope} declarations+=(VariableDeclaration
15                           | SignalDeclaration
16                           | OperationDeclaration
17                           | LocalReaction)*;
18
19  /*********************************************/
20  /*        Signal Definition            */
21  /*********************************************/
22  SignalDeclaration returns sgraph::Event:
23     SignalDefinition;
24
25  SignalDefinition:
26     {EventDefinition} (isInput?='input')? (isOutput?='output')? 'signal'
27     name=ID (':' type=[types::Type|FQN] ('=' varInitialValue=Expression)?
28                         ('with' varCombineOperator=CombineOperator)?)? ';';
29
30  /*********************************************/
31  /*        Variable Definition          */
32  /*********************************************/
33  // A dummy VariableDefinition declaration,
34  // which will be override by an other grammar
35  VariableDefinition:
36     {VariableDefinition} 'variable' name=ID ';';
37
38  /*********************************************/
39  /*          LocalReactionScope           */
40  /*********************************************/
41  LocalReaction:
42     (trigger=(LocalReactionTrigger | ReactionTrigger))?
43     ('/' effect=(ReactionEffect | SuspendEffect)) ';';
44
45  SuspendEffect returns sgraph::Effect:
46     {SuspendEffect} 'Suspend';
47
48  LocalReactionTrigger returns sgraph::Trigger:
49     {ReactionTrigger} stateReaction=StateReaction
50           ('&&' reactionTrigger=ReactionTrigger?)?;
51
52  StateReaction:
53     Entry | Inside | Exit;
54
55  Entry:
56     {Entry} 'Entry';
57
58  Inside:
59     {Inside} 'During';
60
61  Exit:
62     {Exit} 'Exit';
63
64  /***************** TRANSITION ***************** */
65  // The ReactionTrigger is a Trigger and has an Expression,
66  // an optional delay and isImmediate option represented with a '#'
67  ReactionTrigger returns sgraph::Trigger:
68     {ReactionTrigger} (isImmediate?='#')? (delay=INT)? ((trigger=RegularEventSpec)
69     | ('[' guardExpression=Expression ']'));
70
71     // The ReactionEffect is an Effect.
```

```
72 ReactionEffect returns sgraph::Effect:
73    {ReactionEffect} actions+=Expression (=> ',' actions+=Expression)*; // (';')?;
74
75 /***************** Expressions **************** */
76 // Override the stext::PrimaryExpression
77 // remove the ActiveStateReferenceExpression
78 // and add PreValueExpressionreturns that returns
79 // the value of a variable or the status of a signal in the previous tick.
80 PrimaryExpression returns stext::Expression:
81    PrimitiveValueExpression
82    | FeatureCall
83    | ActiveStateReferenceExpression
84    | PreValueExpression
85    | ParenthesizedExpression
86    | EventValueReferenceExpression
87 ;
88
89 //todo: pre(x) should return the same type of x
90 PreValueExpression returns stext::Expression:
91    {PreValueExpression} 'pre' '(' value=FeatureCall ')';
92
93 /*************** COMBINE OPERATORS ************** */
94 enum CombineOperator returns CombineOperator:
95    NONE="'none'" | ADD="'+'" | MULT="'*'" | MAX="'max'" |
96    MIN="'min'" | OR="'or'" | AND="'and'" | HOST="'host'";
```

The Synctext grammar allows the declaration of signals, variables, operations, and local reactions in a state scope (lines 13 - 17). A SignalDefinition (lines 25 - 28) defines a signal, which may be an input signal, an output signal, or both. It has an identifier name, an optional type, an optional initialValue, and an optional CombineOperator. The VariableDefinition (line 35) may be overridden by the SyncCharts and SCCharts grammar to define the appropriate variables definition notation. The LocalReaction (lines 41 - 43) has a trigger and an effect. A trigger may be either a LocalReactionTrigger or a ReactionTrigger. The LocalReactionTrigger (lines 48 - 52) is composed of a StateReaction (Entry, During, Exit) and an optional ReactionTrigger. The ReactionTrigger (lines 67 - 69) has an optional isImmediate option, an optional delay, and either a RegularEventSpec or an Expression. An effect may be either a ReactionEffect or a SuspendEffect. The ReactionEffect (lines 72 - 73) is a list of Expressions delimited by a comma. The SuspendEffect (lines 45 - 46) may be expressed by the Suspend literal. The Expression is implemented by the Stext grammar. The PrimaryExpression (lines 80- 87) is overridden to remove the ActiveStateReferenceExpression and to add the PreValueExpression (line 84). The latter (lines 90 - 91) implements the pre operator. The CombineOperator (lines 94 - 96) is an enumeration. It represents the combine operators for signals and variables.

The Synctext grammar is extended by the SyncCharts grammar to implement its special definitions. The implementation is included in the de.cau.cs.kieler.yakindu.-syncharts.model.stext project, which is introduced in the following.

de.cau.cs.kieler.yakindu.syncharts.model.stext

The project de.cau.cs.kieler.yakindu.syncharts.model.stext provides the SyncChartstext metamodel that extends the Synctext metamodel, as shown in Figure 4.25. It

is used to define the textual language for the SyncCharts Editor. It uses the Synctext grammar and adapts it to add the variable definition. The following lines are extracted from the SyncChartstext grammar:

```
1  /*********************************************/
2  /*        Variable Definition                */
3  /*********************************************/
4  // Override the Synctext VariableDefinition rule
5  // A variable is a Declaration. It has a Direction, a Name, a Type,
6  // an InitialValue, and a CombineOperator
7  VariableDefinition:
8     {VariableDefinition}  (isStatic?='static')? type=[types::Type|FQN] name=ID
9     ('=' varInitialValue=Expression)? ';';
```

A VariableDefinition (lines 7 - 9) defines a variable. A variable can be static. It has a type, an identifier name, and an optional initialValue, which may be an expression.

The Synctext grammar is also extended by the SCCharts grammar to implement the textual language of the SCCharts Editor. The implementation of the de.cau.cs.kieler.yakindu.sccharts.model.stext project is presented in the following.

de.cau.cs.kieler.yakindu.sccharts.model.stext

As mentioned in Section 4.2, SCCharts allow variables to have a direction. This project provides the SCChartstext metamodel, which defines the variable definition of the SCCharts Editor. It extends the Synctext metamodel as depicted in Figure 5.1. The following code shows its implementation in the Xtext grammar:

```
1  /*********************************************/
2  /*        Variable Definition                */
3  /*********************************************/
4  // Override the Synctext VariableDefinition rule
5  // A variable is a Declaration. It has a Direction, a Name, a Type,
6  // an InitialValue and a CombineOperator
7  VariableDefinition:
8     {VariableDefinition} (isInput?='input')? (isOutput?='output')?
9              (isStatic?='static')? type=[types::Type|FQN] name=ID
10             ('=' initialValue=Expression)?
11             ('with' varCombineOperator=CombineOperator)? ';';
```

A VariableDefinition (lines 7 - 11) defines a variable, which can have a direction (input, output). A variable may be static. It has a type, a name, an optional initialValue, and an optional CombineOperator.

The Stext project provides a validator for the textual language. It is discussed in the following.

The validator

The validator is implemented in the de.cau.cs.kieler.yakindu.model.stext.validation package. The SyncJavaValidator class extends the STextJavaValidator class from the YAKINDU SCT Editor. It implements several methods, which represent the validity rules for the textual language. It also overrides some Stext validity rules to adapt them.

In order to add supplement checks for the type conformance, the SyncTypeInferrer class from the de.cau.cs.kieler.yakindu.sccharts.model.stext.types package extends the STextDefaultTypeInferrer class. STextDefaultTypeInferrer is written in the Xtend language. The SyncTypeInferrer allows the check for the PreValueExpression type conformance (e.g., for A=pre(B), A and B should have the same type).

The validator may be extended by the SyncCharts Editor and the SCCharts Editor to add special requirements. It permits to avoid inconsistent models by displaying the appropriate error or warning in the Editor. The Editor project is presented in the following.

5.2.3 The Editor project

There are three Editor projects, as depicted in Figure 5.1. The Global Editor project de.cau.cs.kieler.yakindu.ui.editor implements the objects used in common by the SyncCharts Editor and the SCCharts Editor.

de.cau.cs.kieler.yakindu.ui.editor

This Global Editor project extends the YAKINDU SCT Editor to add the additional elements defined in the Sgraph project. It implements the editparts, the palette, the properties view, the modeling assistant provider, and the diagram initializer.

Editparts The package de.cau.cs.kieler.yakindu.ui.editor.parts contains the implementation of the editparts for the SyncState and the SyncTransition. The following code is an extract of the SyncTransitionEditPart:

```
1  /**
2  * Update the source decorator (red circle for strong abortion, green
3  * triangle for normal termination, and no decorator for weak abortion)
4  *
5  * @param The
6  *            transition figure
7  */
8  private void updateTransitionSourceDecorator(TransitionFigure transition) {
9     EObject element = resolveSemanticElement();
10    if (element instanceof SyncTransition) {
11       TransitionType type = ((SyncTransition) element).getType();
12       switch (type) {
13       case WEAKABORT:
14          transition.setSourceDecoration(null);
15          break;
16       case NORMALTERMINATION:
17          transition.setSourceDecoration(TransitionDecorator
18                .createNormalTerminationDecoration(getMapMode().DPtoLP(
19                      TransitionDecorator.LINE_WIDTH)));
20          break;
21       case STRONGABORT:
22          transition.setSourceDecoration(TransitionDecorator
23                .createStrongAbortDecoration(getMapMode().DPtoLP(
24                      TransitionDecorator.LINE_WIDTH)));
25          break;
26       }
27    }
28 }
```

It extends the TransitionEditPart class, which implements the editpart of the Statecharts transitions. The SyncTransitionEditPart updates the transition source decorator when the type changes. If the transition is a weak abortion transition (lines 13 - 15), the source decorator is empty. If the transition is a normal termination (lines 16 - 20), the method createNormalTerminationDecoration() creates a green triangle source decorator. If the transition is a strong abortion transition (lines 21 - 25), a red circle source decorator is created.

The YAKINDU SCT Editor displays the <name> literal by default when a state or a region name is left empty. This feature is removed by overriding the getEditString() method from the AttributeParser class, which is extended by the KielerAttributeParser class.

Palette The palette is implemented by the KielerPaletteFactory class, which is included in the de.cau.cs.kieler.yakindu.ui.editor.factory package. It extends the DefaultSCTPaletteFactory class from the YAKINDU SCT Editor to define elements properties (type, name, and icon) and to bind them to the palette.

```
1 protected void createInitialStateEntry(PaletteContainer container) {
2    container.add(new CreationToolEntry("Initial State",
3          "Creates an initial state",
4          getType(KielerMetaModelTypeFactory.SYNC_INITIAL_STATE_ID),
5          findIcon("icons/obj16/Initial-State-16.png"),
6          findIcon("icons/obj32/Initial-State-32.png")));
7 }
```

The lines above defines an initial state element. Once all elements of the palette are defined, they can be bound by the createPaletteEntries() method presented below:

```
1 public void createPaletteEntries(PaletteRoot root) {
2    PaletteContainer container = createToolsCategory(root);
3    createTransitionEntry(container);
4    createStateEntry(container);
5    createInitialStateEntry(container);
6    createFinalStateEntry(container);
7    createRegionEntry(container);
8    createChoiceEntry(container);
9 }
```

Properties view The Properties views are implemented by the de.cau.cs.kieler.yakindu.ui.editor.propertysheets package. The package contains the SyncStatePropertySection and the SyncTransitionPropertySection classes. The SyncTransitionPropertySection extends the TransitionPropertySection class, which implements the properties view of the YAKINDU SCT transitions. The following code is an extract of the SyncTransitionPropertySection class:

```
1  /**
2  * Create the selection Combo. It allows to select the transition type
3  * (WEAKABORT, STRONGABORT or NORMALTERMINATION)
4  *
5  * @param parent
6  *            the parent Composite
7  */
8  private void createTransitionTypeControl(Composite parent) {
9     Label kindLabel = getToolkit().createLabel(parent, "Transition Type: ");
10    GridDataFactory.fillDefaults().applyTo(kindLabel);
11    transitionTypeKindViewer = new ComboViewer(parent, SWT.READ_ONLY
12          | SWT.SINGLE);
13    transitionTypeKindViewer.setContentProvider(new ArrayContentProvider());
14    transitionTypeKindViewer.setLabelProvider(new LabelProvider());
15    transitionTypeKindViewer.setInput(TransitionType.values());
16    GridDataFactory.fillDefaults().grab(true, false)
17          .applyTo(transitionTypeKindViewer.getControl());
18 }
```

It creates a combo-box to allow the user to select the transition type (weak abortion, strong abortion, or normal termination).

Modeling assistant provider The KielerModelingAssistantProvider from the de.cau.cs.kieler.yakindu.ui.editor.assistent package implements the modeling assistant provider. It overrides the getTypesForPopupBar(), getTypesForTarget(), and getTypesForSource() methods from the ModelingAssistantProvider class to bind the graphical elements used by the SyncCharts Editor and the SCCharts Editor. The getTypesForPopupBar() returns the elements displayed by the popup bar assistant provider. The getTypesForTarget() and getTypesForSource() methods return the elements displayed by the connection creation assistant. The following code shows the implementation of the getTypesForPopupBar() method:

```
1  @Override
2  public List<IElementType> getTypesForPopupBar(IAdaptable host) {
3     IGraphicalEditPart editPart = (IGraphicalEditPart) host
4           .getAdapter(IGraphicalEditPart.class);
5     if (editPart instanceof RegionEditPart
6           || editPart instanceof RegionCompartmentEditPart)
7        return Lists.newArrayList(SYNC_INITIAL_STATE, SYNC_STATE,
8              SYNC_FINAL_STATE, CHOICE);
9     if (editPart instanceof StateEditPart
10          || editPart instanceof StateFigureCompartmentEditPart)
11       return Lists.newArrayList(REGION);
12    return Lists.newArrayList();
13    }
```

The method returns an initial state, a simple state, a final state, and a choice (line 7) if the host element is a region (line 5). It returns a region (line 11) if the host element is state (line 9).

Diagram initializer The KielerDiagramInitializer class extends the DefaultDiagramInitializer. It is contained in the de.cau.cs.kieler.yakindu.ui.editor.factory package. The following method initializes the Statechart diagram:

```
1  @Override
2  public void initModel(Statechart statechart, Diagram diagram,
3        PreferencesHint preferencesHint) {
4     // Create an root Region
5     Region region = factory.createRegion();
6     //add the root Region to the Statechart
7     statechart.getRegions().add(region);
8     Node regionView = ViewService.createNode(diagram, region,
9           SemanticHints.REGION, preferencesHint);
10    setRegionViewLayoutConstraint(regionView);
11
12    // Create the root State
13    SyncState state = syncfactory.createSyncState();
14    state.setName(statechart.getName());
15
16    // create the state region
17    Region stateregion = factory.createRegion();
18    state.getRegions().add(stateregion);
19    stateregion.setName(INITIAL_REGION_NAME);
20    region.getVertices().add(state);
21    Node stateNode = ViewService.createNode(
22          getRegionCompartmentView(regionView), state,
23          SemanticHints.STATE, preferencesHint);
24    setStateViewLayoutConstraint(stateNode);
25    }
```

The method initModel() creates the root region (line 5) and adds the latter to the Statechart (line 7). It also creates the root state (line 13) and a region inside (line 17). the root state has the same Statechart name (line 14), which is given in the creation wizard while creating a new diagram.

de.cau.cs.kieler.yakindu.syncharts.ui.editor and de.cau.cs.kieler.yakindu.sccharts.ui.-editor projects extend the de.cau.cs.kieler.yakindu.ui.editor project to register the editors and to set the diagram file extension for each editor. Every project defines its extensions as introduced in Section 5.1.1. This work adds additional extensions for the automatic layout. The following subsection gives an overview of the automatic layout integration in the SyncCharts Editor and the SCCharts Editor.

5.2.4 Automatic layout

As mentioned in Section 4.1.4, the ideal layout for the SyncCharts models is the *dot* layout. In order to register this layout as default for the SyncCharts Editor and the SCCharts Editor, the de.cau.cs.kieler.kiml [15] project offers the layoutInfo extension point, which is extended in the plugin.xml of each editor. The following lines specify the properties of the SyncCharts/SCCharts diagram with regard to automatic layout:

```
1  <extension
2          point="de.cau.cs.kieler.kiml.layoutInfo">
3       <option
4             class="org.yakindu.sct.model.sgraph.Region"
5             option="de.cau.cs.kieler.diagramType"
6             value="de.cau.cs.kieler.layout.diagrams.stateMachine">
7       </option>
```

```
8       <option
9             class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
10            option="de.cau.cs.kieler.diagramType"
11            value="de.cau.cs.kieler.layout.diagrams.boxes">
12      </option>
13      <option
14            class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
15            option="de.cau.cs.kieler.spacing"
16            value="1.0">
17      </option>
18      <option
19            class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
20            option="de.cau.cs.kieler.borderSpacing"
21            value="1.0">
22      </option>
23      <option
24            class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
25            option="de.cau.cs.kieler.sizeConstraint"
26            value="[MINIMUM_SIZE, DEFAULT_MINIMUM_SIZE]">
27      </option>
28      <option
29            class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
30            option="de.cau.cs.kieler.expandNodes"
31            value="true">
32      </option>
33      <option
34            class="de.cau.cs.kieler.yakindu.sgraph.syncgraph.SyncState"
35            option="de.cau.cs.kieler.interactive"
36            value="true">
37      </option>
38      <option
39            class="org.yakindu.sct.model.sgraph.Region"
40            option="de.cau.cs.kieler.borderSpacing"
41            value="5.0">
42      </option>
43      <option
44            class="org.yakindu.sct.model.sgraph.Region"
45            option="de.cau.cs.kieler.direction"
46            value="RIGHT">
47      </option>
48      <semanticOption
49            class="org.yakindu.sct.model.sgraph.Scope"
50            config="de.cau.cs.kieler.synccharts.diagram.
51                    custom.HVLayoutConfig">
52      </semanticOption>
53      <semanticOption
54            class="org.yakindu.sct.model.sgraph.Reaction"
55            config="de.cau.cs.kieler.synccharts.diagram.
56                    custom.ActionLayoutConfig">
57      </semanticOption>
58      <semanticOption
59            class="org.yakindu.sct.model.sgraph.Scope"
60            config="de.cau.cs.kieler.synccharts.diagram.
61                    custom.AnnotationsLayoutConfig">
62      </semanticOption>
63      <semanticOption
64            class="org.yakindu.sct.model.sgraph.Reaction"
65            config="de.cau.cs.kieler.synccharts.diagram.
66                    custom.AnnotationsLayoutConfig">
67      </semanticOption>
68    </extension>
69  </plugin>
```

The extension specifies for a SyncState the following properties:

- spacing='1.0' (lines 13 - 17)
- borderSpacing='1.0' (lines 18 - 22)
- sizeContraint='[MINIMUM_SIZE, DEFAULT_MINIMUM_SIZE]'
- expandNodes=True (lines 28 - 32)
- interactive=True (lines 33 - 37)

The KIELER View Management (KIVI) [24] project offers the option to apply the automatic layout after the model has been changed. The de.cau.cs.kieler.yakindu.-syncchartskivi and the de.cau.cs.kieler.yakindu.sccharts.kivi projects contain both the LayoutAfterModelChangedCombination class, which integrates this feature respectively in the SyncCharts Editor and the SCCharts Editor. The projects use the de.cau.cs.kieler.core.kivi.combinations extension point to register this feature and to allow its use by the KIELER framework.

5.2.5 Reducing white spaces

As mentioned in Section 4.4.5, white spaces can be reduced by eliminating the white space under a state name, reducing the font size, and minimizing the vertical spacing inside a state.

White space under states names

The minimum size of a state in YAKINDU SCT is 40 pixels in height and 40 pixels in width. The size can be configured by manipulating the DEFAULT_NODE_WIDTH and DEFAULT_NODE_HEIGHT attributes of the SyncMapModeUtils class from the de.cau.cs.kieler.yakindu.ui.editor.figures.utils package.

```
1    private static final int DEFAULT_NODE_WIDTH = 30;
2    private static final int DEFAULT_NODE_HEIGHT = 30;
```

Font size of labels and text compartments

The SyncPreferenceInitializer class from the de.cau.cs.kieler.yakindu.ui.editor.preference package allows to set the default font for the editor. The following lines set the verdena font size 8 as default.

```
1 // set default font
2       FontData defaultFont = new FontData("Verdana", 8, SWT.NONE);
3       PreferenceConverter.setDefault(getPreferenceStore(),
4             IPreferenceConstants.PREF_DEFAULT_FONT, defaultFont);
```

Vertical spacing inside states

The default vertical spacing provided by YAKINDU SCT is 5 pixels. The following lines minimize the vertical spacing to 0 pixel. They are extracted from the SyncState class, which is included in the de.cau.cs.kieler.yakindu.ui.editor.figures package.

```
1 public SyncStateFigure() {
2       // reduce white spaces
3       ((GridLayout) this.getLayoutManager()).verticalSpacing=0;
4 }
```

This chapter presented the contribution of Itemis AG and gave details about the implementation of the SyncCharts Editor as well as the SCCharts Editor based on YAKINDU SCT. The next chapter evaluates and summarises this work. It also gives some ideas for future research.

6 Conclusions

This work implements a SyncCharts Editor as well as a SCCharts Editor by extending the YAKINDU SCT Editor instead of completely creating an independent editor. The following section evaluates this approach by means of maintenance effort.

6.1 Evaluation

In order to evaluate this work that aimed in minimizing maintenance effort, the following criteria are measured in the ThinKCharts Editor and the new SyncCharts Editor based on YAKINDU SCT: Number of lines of the code, number of classes, number of plug-ins, and number of dependencies. The results are presented in Table 6.1.

	ThinKCharts Editor		SyncCharts Editor Based on YAKINDU						Reduction % Rate	
			Global		SyncCharts		Total			
	A	M	A	M	A	M	A	M	A	M
Plug-ins	12	6	4	3	4	3	8	6	33	0
Packages	61	20	36	14	30	11	66	25	-	-
Classes	473	220	100	24	62	30	162	54	66	75
Lines of code	125.446	10.146	53.792	1.388	53.603	341	107.395	1729	15	83
Dependencies	78		34		28		62		21	

A=All
M=Manual

Table 6.1: This table compares the source code (the manual code only as well as the manual code and the auto generated) of the ThinKCharts Editor and the SyncCharts Editor based on YAKINDU. It also shows the reduction rate of the SyncCharts Editor based on YAKINDU source code

In the first place, the table permits to compare both editors based on the criteria mentioned above and gives an overview of the reduction rate of the SyncCharts Editor based on YAKINDU SCT source code. Hand coded as well as auto generated, the number of plug-ins and the number of packages do not differ much in both editors. The ThinKCharts Editor contains 473 classes, 220 of them are hand coded. The SyncCharts Editor based on YAKINDU contains less plug-ins. It is composed of 162 classes, only 54 of them are hand coded. The ThinKCharts Editor has 125.446 lines of code. It has 10.146 lines of hand coded code. The SyncCharts Editor based

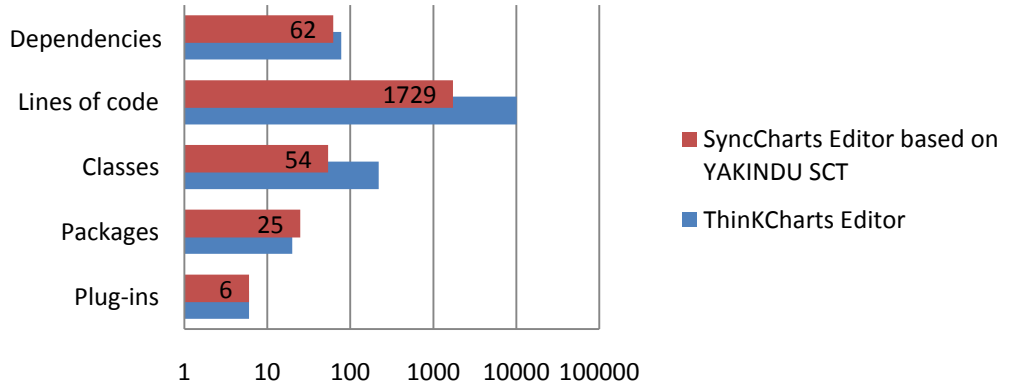


Figure 6.1: Comparison of the manual code of the ThinKCharts Editor and the SyncCharts Editor based on YAKINDU SCT. The horizontal axis has a logarithmic scale. It represents the number of plug-ins, packages, classes, lines of code, and dependencies

on YAKINDU contains 18.051 less lines of code. It has only 1.729 lines of hand coded code (a reduction rate of 83%). This is significantly smaller than the number of lines of hand coded code in the ThinKCharts Editor. The number of dependencies in the SyncCharts Editor based on YAKINDU is also smaller than the number of dependencies of the ThinKCharts Editor. There is a difference of 16 dependencies. Figure 6.1 compares the manual code of the ThinKCharts Editor and the SyncCharts Editor based on YAKINDU SCT. The results show that the approach used in this work provides a SyncCharts Editor which is much more maintainable than the ThinKCharts Editor, which contains more hand coded classes and code.

In the second place, Table 6.1 shows the efficiency of the generic editor implemented by this work. As depicted in Figure 5.1, the Global project is extended by the SyncCharts Editor. The latter contains 3 plug-ins, 11 packages, 30 classes, and 341 lines of code, which are manually written (not auto generated). The SCCharts Editor was extended from the Global project with minimal effort.

6.2 Summary

This master thesis presented an approach on how to implement a SyncCharts Editor and a SCCharts Editor based on YAKINDU SCT in a generic way. The SCCharts Editor allows the modeling based on the concept of Statecharts. Part of this work is a detailed comparison between the ThinKCharts Editor and the YAKINDU SCT Editor. That aimed in providing Itemis AG with necessary information for generalizing the YAKINDU SCT to make an implementation of various Statechart dialects based on YAKINDU SCT possible.

This approach minimized the effort of maintenance of the SyncCharts Editor and the new SCCharts Editor. It also contributed to the improvement of the extensibility of the YAKINDU SCT project.

6.3 Future Work

Some ideas for future extensions and research based on this work are presented in the following.

6.3.1 SyncChart Importer

This master thesis provided a SyncCharts Editor, which will be integrated in the KIELER framework. When replacing the ThinKCharts Editor, it is no longer possible to edit old models.

Furthermore, the KIELER framework provides a model transformations infrastructure, which allows to transform an Esterel program into a SyncChart [28]. However, the resulting SyncChart may only be edited using the ThinKCharts Editor.

A SyncChart Importer may be implemented in order to import old models to the new format.

6.3.2 Simulation and Code Generation

As stated in Section 2.1, the KIELER framework provides a SyncCharts simulator [19][20] based on the KlePto project [21]. The simulator offers the user the opportunity to directly execute and simulate a SyncChart. The KIELER framework also allows to generate C-Code from a SyncChart. These two features could be interesting for the SyncCharts Editor and the SCCharts Editor, which were implemented in this work.

6.3.3 An Actor Oriented Editor based on YAKINDU

As mentioned in Section 1.2, the KAOM project permits rendering and simulating actor oriented modeling languages like Ptolemy. An idea for a future work is to implement a KAOM Editor based on YAKINDU Damos[1].

6.3.4 A SCXML converter

State Chart eXtensible Markup Language (SCXML)[2] is an XML notation that allows to describe Statecharts [10]. It enables to represent hierarchy, history, concurrency, and synchronization [11]. The objective of SCXML is to generify state diagrams notations.

[1] http://code.google.com/a/eclipselabs.org/p/yakindu/
[2] http://www.w3.org/TR/2005/WD-scxml-20050705/

KIELER SyncCharts as well as SCCharts models are represented in the XMI notation, which is developed for representing UML diagrams. A valuable future work is to represent them in the SCXML notation. This can be done by adapting the editor serializer or by converting the generated XMI notation using an Extensible Stylesheet Language Transformations (XSLT) script.

Bibliography

[1] SWT/JFace in Action: GUI Design with Eclipse 3.0 (1st ed.). page 496. Manning Publications, 2004.

[2] Daniel Amyot, Hanna Farah, and Jean-Francois Roy. Evaluation of Development Tools for Domain-Specific Modeling Languages. In *In System Analysis and Modeling: Language Profiles*, volume 4320, pages 183 – 197. Springer, 2006.

[3] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical Report RR 95–52, rev. RR 96–56, I3S, Sophia-Antipolis, France, Rev. April 1996.

[4] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. `http://www.esterel-technologies.com`.

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

[6] Charles André. Computing SyncCharts reactions. *Electronic Notes in Theoretical Computer Science*, 88(33):3–19, 2004.

[7] Gérard Berry. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.

[8] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] Steven D. Bragg, Carl G. Driskill, Ground Systems Group, Inc Frontier Engineering, Stillwater, and OK. Diagrammatic-graphical programming languages and DoD-STD-2167A. In *Cost Effective Support Into the Next Century*, pages 211–220. AUTOTESTCON '94. IEEE Systems Readiness Technology Conference. ', AUTOTESTCON '94., Sep 1994.

[10] Jenny Brusk and Torbjörn Lager. Developing Natural Language Enabled Games in (Extended) SCXML. In *International Symposium on Intelligence Techniques in Computer Games and Simulations.*

[11] Jenny Brusk, Torbjörn Lager, Anna Hjalmarsson, and Preben Wik. DEAL – Dialogue Management in SCXML for Believable Game Characters. In *Future Play '07 Proceedings of the 2007 conference on Future Play*.

[12] Björn Duderstadt. Sccharts: A sequentially constructive statecharts dialect. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, November 2012.

[13] Stephen A. Edwards Dumitru Potop-Butucaru. *Compiling Esterel*. Springer, 2007.

[14] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

[15] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. Automatic layout and structure-based editing of UML diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)*, Dresden, March 2010.

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

[17] Michael Matzen. A generic framework for structure-based editing of graphical models in Eclipse. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf`.

[18] Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse Rich Client Platform : Designing, Coding, and Packaging Java™ Applications, Second Edition*. Addison-Wesley Professional, 2010.

[19] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf`.

[20] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010*, GI-Edition – Lecture Notes in Informatics (LNI), pages 891–896, Leipzig, Germany, September 2010. Bonner Köllen Verlag.

[21] Christian Motika, Hauke Fuhrmann, Reinhard von Hanxleden, and Edward A. Lee. Executing domain-specific models in Eclipse. Technical Report 1214, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2012. ISSN 2192-6247.

[22] Christian Motika, Miro Spönemann, Hauke Fuhrmann, Christoph Krüger, John Julian Carstens, and Reinhard von Hanxleden. KIELER Actor Oriented Modeling (KAOM). Poster presented at 9th Biennial Ptolemy Miniconference (PTCONF'11), Berkeley, CA, USA, February 2011.

[23] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. Synchronous Java: Light-weight, deterministic concurrency and preemption in Java. Technical Report 1213, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2012. ISSN 2192-6247.

[24] Martin Müller. View management for graphical models. Master thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2010. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mmu-mt.pdf.

[25] Steve Northover and Mike Wilson. *SWT: The Standard Widget Toolkit, Volume 1.* Addison-Wesley Professional, 2004.

[26] André Ohlhoff. Simulating the Behavior of SyncCharts. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/aoh-st.pdf.

[27] Mark Rogalski. The eclipse rich client platform. Technical report, IBM, 2005. http://www.eclipse.org/ercp/RCP-TechPaper.pdf.

[28] Ulf Rüegg, Christian Motika, and Reinhard von Hanxleden. Interactive transformations for visual models. In *3rd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2011) at conference INFORMATIK 2011*, GI-Edition – Lecture Notes in Informatics (LNI), Berlin, Germany, October 2011. Bonner Köllen Verlag.

[29] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2008. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf.

[30] Matthias Schmeling. ThinKCharts—the thin KIELER SyncCharts editor. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf.

[31] Christian Schneider. On integrating graphical and textual modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2011. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/chsch-dt.pdf.

[32] Michael von der Beeck. A comparison of STATECHARTS Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128 – 148. Springer-Verlag, 1994.

[33] Reinhard von Hanxleden. Synccharts in c—a proposal for light-weight, deterministic concurrency. In Albert Benveniste, Stephen A. Edwards, Edward Lee, Klaus Schneider, and Reinhard von Hanxleden, editors, *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*, number 09481 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 November 2009.

[34] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. SCCharts—Sequentially Constructive Statecharts for safety-critical applications. submitted, 2013.

[35] Reinhard von Hanxleden, Michael Mendler, Joaquin Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Partha Roop, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A Conservative Extension of the Synchronous Model of Computation. Technical report, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, to appear.

[36] Hong Yul Yang, Ewan Tempero, and Hayden Melton. An Empirical Study into Use of Dependency Injection in Java. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on.*