Christian-Albrechts-Universität zu Kiel

Student Research Project

ThinKCharts

The Thin KIELER SyncCharts Editor

Matthias Schmeling

September 2, 2009

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl. Inf. Hauke Fuhrmann

Abstract

Model-Driven Development (MDD) nowadays is a common means to create software on a high level of abstraction. It makes use of graphical depictions of software systems to abstract from data and algorithms and thus streamlines the development process and increases productivity. *ThinKCharts* is a graphical *SyncCharts* editor that uses MDD techniques to improve the process of developing synchronous *StateCharts*. It was created by employing several code generation frameworks. Afterwards, the generated code was modified in some aspects to take peculiarities of SyncCharts into account and improve usability. While the possibility to automatically generate large portions of code was a major asset, it was a difficult task to integrate all desired modifications. Many of the functionalities a good SyncCharts editor has to provide could not simply be plugged in but had to be implemented by thoroughly studying and altering the generated code.

Contents

List of Figures

List of Tables

List of Acronyms

COM	Component Object Model
EMF	Eclipse Modeling Framework
EMOF	Essential MetaObject Facility
GEF	Graphical Editing Framework
GEMS	Generic Eclipse Modeling System
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework
IDE	Integrated Development Environment
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
MDD	Model-Driven Development
MOF	MetaObject Facility
OMG	Object Management Group
RCP	Rich Client Platform
SSM	Safe State Machine
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

SyncCharts, a dialect of David Harel's *StateCharts* [5] and invented by Charles André [1], is a widely used technique to represent software with hierarchical and parallel components. SyncCharts have the same semantics as *Mealy machines* [7], but enhance those by adding hierarchy and parallelism. This means that the states known from Mealy machines may contain other states. The states contained in another state can be seperated in groups of states that are connected with each other and are executed concurrently. In order for this to work, every one of those groups has to have an initial state. Furthermore, there is a new special kind of transition, a so-called *normal termination transition*, which represents the transition from a state that has to be taken as soon as a final state is reached in every one of its inner state groups. For further information on SyncCharts and a detailed discussion of their semantics see André [1]. Figure 1.1 shows an example of a SyncChart.

As SyncCharts can become quite complex, their creation and readability can sometimes be demanding even despite all those convenient features mentioned above. As explained by Fuhrmann and von Hanxleden [4], automatic layouts and structure based editing mechanisms would be useful to reduce the difficulties of their creation and modification and improve their readability. In addition, the editor that was automatically generated in the project described in this thesis did not allow model elements to have several different appearances. However, SyncCharts need such a feature since the states in such a diagram may be depicted as rounded rectangles or ellipses, have a thin or a thick or even a doubled border and so on. Also, the editor did not allow to define the triggers and effects of transitions by simply writing them next to it. These and other features have to be implemented to improve the usability of such an editor.

1.1 Tool Introduction

The *ThinKCharts* editor is built upon several other projects. The next sections introduce them one by one.

1.1.1 Eclipse

Eclipse[1] is an Integrated Development Environment (IDE). It is available for a wide variety of programming languages, such as Java and C++, and several different purposes like plug-in development, model-driven development, and others. The

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

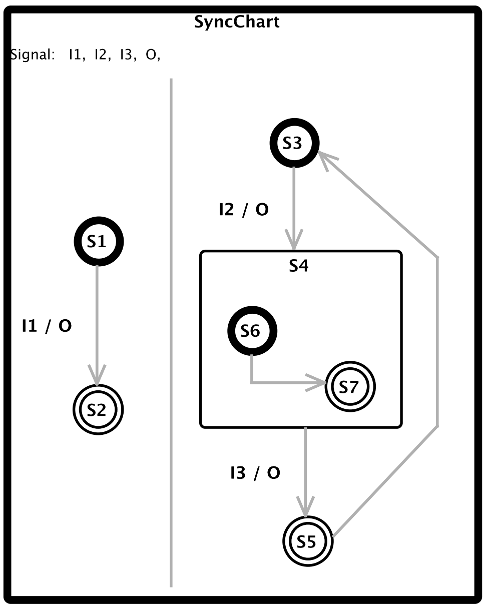


Figure 1.1: A SyncChart.

advantage of Eclipse is its extensibility. The core of Eclipse consists only of a small kernel that is responsible for loading and starting plug-ins. All additional functionality, such as loading and saving files, editing text files, drawing model diagrams, and so on are provided by those plug-ins, which are loaded and executed when needed. In addition, every plug-in may define its own extension points that can be extended by other plug-ins as well. By connecting plug-ins with the Eclipse kernel and with each other, whole rich development environments can be created. Although Eclipse usually comes with a variety of pre-installed plug-ins, it is possible to remove those and only add plug-ins that are needed for a specific purpose to create a Rich Client Platform (RCP). Figure 1.2 illustrates the plug-in mechanism.

1.1.2 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[2] is a framework that is used to create abstract representations in form of models. For this purpose, users may define custom meta-models that determine how any entity that is to be represented in a model is constructed. The meta-models adhere to the Essential MetaObject Facility (EMOF) Specification by the Object Management Group (OMG)[3]. They can be represented in annotated Java code, *XMLSchema*, or so-called *Ecore models*. The latter are similar to class diagrams of the Unified Modeling Language (UML). They may contain packages, which in turn contain classes. These classes have attributes as well as

[2]http://www.eclipse.org/emf/
[3]http://www.omg.org/mof/

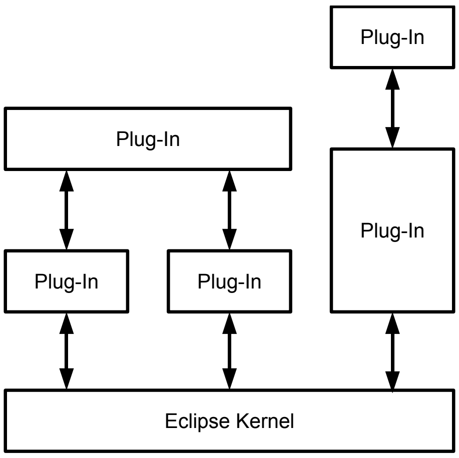


Figure 1.2: The structure of Eclipse.

operations. Classes may be linked by *aggregations*, *associations*, and *generalizations*. Aggregations depict a relationship that may be called 'belongs to'. A human for example aggregates a head, two arms, and two legs. The limbs belong to the person and if a magician makes that person disappear, its limbs disappear, too. Associations represent references to other classes. An example for an association is a list of participants. The list contains references to students, one reference per student. These students exist outside of the list, so when someone throws the list into the litter bin, the referenced students do not suddenly die or disappear. Generalizations represent inheritance and can be viewed as an 'is a' relationship. This means that if a vehicle generalizes a car, then the car 'is a' vehicle. Furthermore, Ecore models allow to define custom data types and enumeration types.

EMF is also responsible for the persistent storage of models. Every model as well as every meta-model is automatically translated to Extensible Markup Language (XML) and stored as a plain text file. This makes it possible to modify models on a textual level rather than with a diagram editor. However, modifying complex models with just a text editor is quite demanding. Figure 1.3 depicts a simple example of an Ecore diagram, while Figure 1.4 shows the same model in plain text.

1.1.3 Graphical Editing Framework and Draw2D

Draw2D is on the one hand a toolkit that allows to create graphical representations of objects and to draw them on the screen. The Graphical Editing Framework (GEF)[4] is on the other hand a means to generate graphical applications from models. It provides an additional layer around *Draw2D* to reach a higher abstraction level.

[4]http://www.eclipse.org/gef/

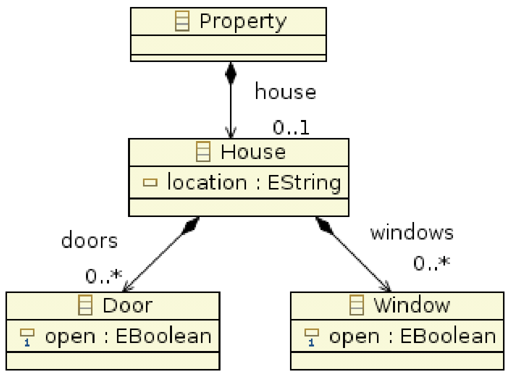


Figure 1.3: An Ecore diagram.

```
<?xml version="1.8" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="defaultname"
    nsURI="http://defaultname/1.0" nsPrefix="defaultname">
  <eClassifiers xsi:type="ecore::EClass" name="House">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
          name="location"
          eType="ecore:EDataType http://www.eclipse.org
            /emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
          name="doors"
          upperBound="-1"
          eType="#//Door"
          containment="true"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
          name="windows"
          upperBound="-1"
          eType="#//Window"
          containment="true"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="EEnum0"/>
  <eClassifiers xsi:type="ecore:EClass" name="Door">
    <eStructuralFeatures xsi:type="ecore:EAttribute"
          name="open" lowerBound="1"
          eType="ecore:EDataType http://www.eclipse.org
            /emf/2002/Ecore#//EBoolean"/>
  </eClassifiers>
</ecore:EPackage>
```

Figure 1.4: Textual representation of an Ecore model.

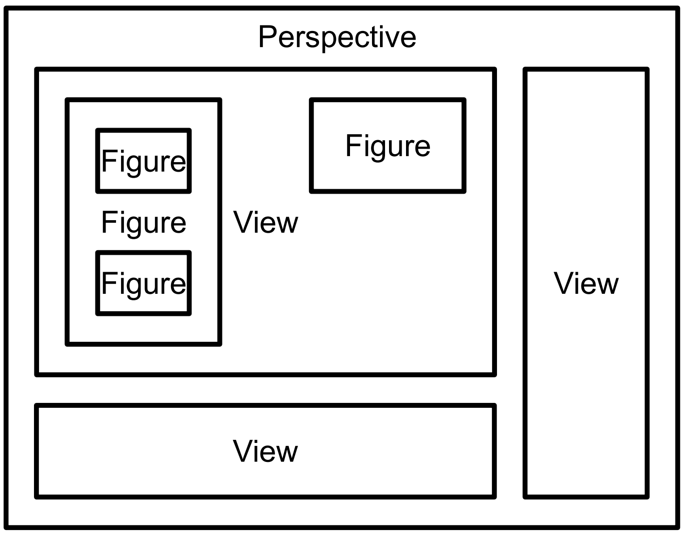


Figure 1.5: The perspective-view-figure hierarchy.

The entity that provides this abstraction is the *EditPart*. It acts as a controller that receives commands and forwards those to the underlying *Draw2D* objects. The basic unit of *Draw2D* that is most important for this paper, is the Figure. This class represents anything that is visible on the screen. A Figure may contain an image, a rectangle, a list of vectors, and many other things. Additionally, it may contain other Figures as well, so that graphical depictions on the screen may be comprised of a whole hierarchy of Figures. Every Figure also has a layout, which determines how its inner Figures are to be arranged. Figures may be contained within a view, which can be thought of as a window. Examples for views are the properties view or the outline in Eclipse. Several views in turn form a perspective. One example for a perspective is the debug perspective in Eclipse. Figure 1.5 illustrates this hierarchy.

1.1.4 Graphical Modeling Framework

The *Graphical Modeling Framework*[5] forms a bridge between the two aforementioned technologies EMF and GEF. It uses meta-models that have been created with EMF and thus are described in XML to generate GEF code, which when executed runs a graphical editor for the entities depicted in the meta-model. As easy as it sounds here, this process is quite complex and involves several steps, which are described in this section.

[5]http://www.eclipse.org/gmf/

The meta-model Figure 1.6 summarizes the Graphical Modeling Framework (GMF) code generation process. The first step is to create a meta-model of the entities that are to be created with the generated diagram editor. As explained above, this model is an Ecore model and is described in a way similar to UML class diagrams. After that, a so-called *genmodel* is derived from the meta-model automatically. It contains the information of the Ecore model, but also a little more and resembles a class diagram even more closely. The genmodel is used to generate the model as well as the edit code. The model code consists of Java classes that represent the entities depicted in the meta-model along with their attributes and operations, such as getters and setters. The edit code builds upon the model code and provides means to access certain parts of the model elements. It is used later by the diagram code that is generated in this process.

The graphical definition model In addition to the Ecore model, three more models need to be defined: The *graphical definition model*, the *tooling definition model*, and the *mapping definition model*. The graphical definition model determines how the entities of the meta-model are to be displayed in the editor. It consists of a canvas, which represents the highest level element of the editor. The canvas contains a figure gallery, which holds all the figures that might be displayed in the diagram. In the case of SyncCharts these might be arrows for transitions, circles and rectangles for states, and others. It is also possible to put so-called custom figures into the figure gallery. These figures point to a Java class that implements the figure's functionality. This way it is possible to modify and enhance the behaviour of the displayed figures to a great amount. A graphical definition model may also contain nodes, connections, compartments, and labels. The nodes are representations of elements that can be dragged and dropped onto the diagram surface, while connections represent elements that may connect two different nodes with each other. Nodes and connections also reference figures from the figure gallery, which are supposed to be their graphical representations. Compartments are containers for an arbitrary amount of other elements and are assigned to the figure they are supposed to be contained within. Labels finally are used to show textual information. They may also reference figures from the figure gallery, so that they are displayed within these figures. A graphical definition model for a SyncCharts editor might for example contain a node for the states and assign those nodes the circle figures from the figure gallery, while also containing a connection for the transitions which reference an arrow from the figure gallery. Also, a compartment might be defined and assigned to the state rectangle figure to later contain other states, and a label might be added, which is assigned to the circle figure to later depict the state's name. Figure 1.7 shows an example of a simple graphical definition model.

The tooling definition model The tooling definition model determines which tools are available to create and modify model entities. Its top level element is the tool registry. It may contain a tool palette among several other things such as a main

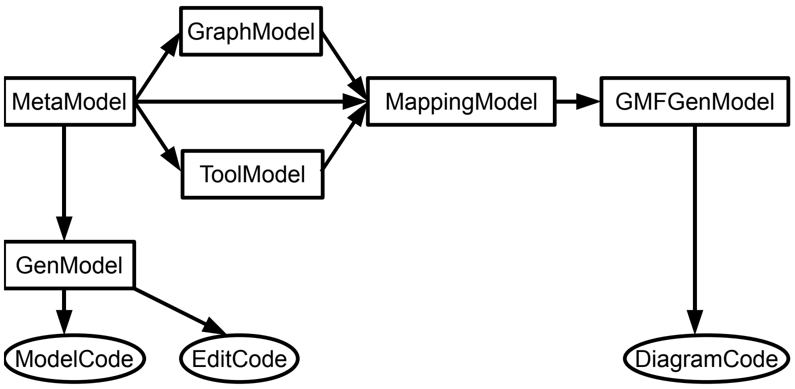
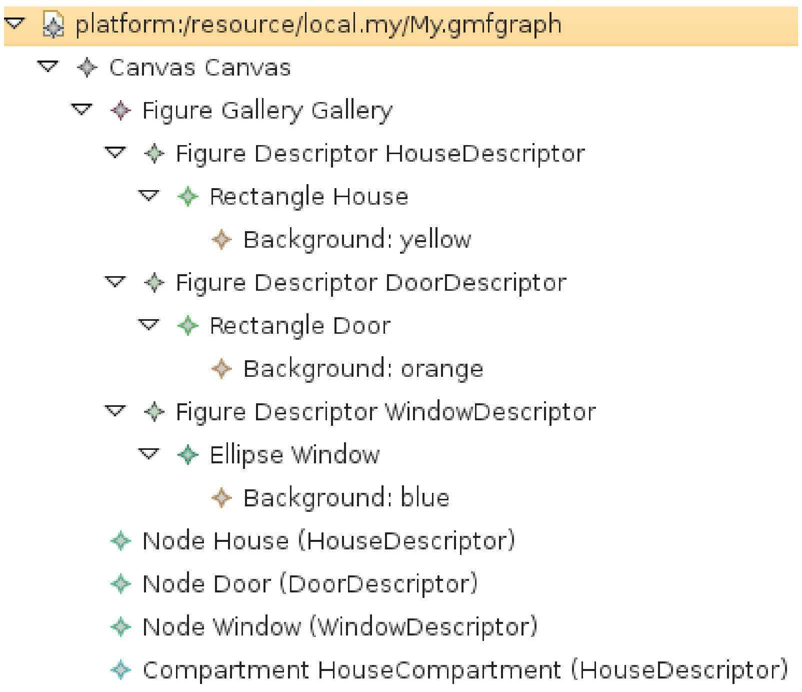
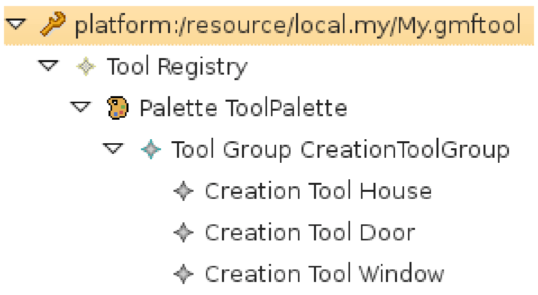


Figure 1.6: The GMF code generation process.



```
platform:/resource/local.my/My.gmfgraph
  Canvas Canvas
    Figure Gallery Gallery
      Figure Descriptor HouseDescriptor
        Rectangle House
          Background: yellow
      Figure Descriptor DoorDescriptor
        Rectangle Door
          Background: orange
      Figure Descriptor WindowDescriptor
        Ellipse Window
          Background: blue
    Node House (HouseDescriptor)
    Node Door (DoorDescriptor)
    Node Window (WindowDescriptor)
    Compartment HouseCompartment (HouseDescriptor)
```

Figure 1.7: The outline of a graphical definition model.



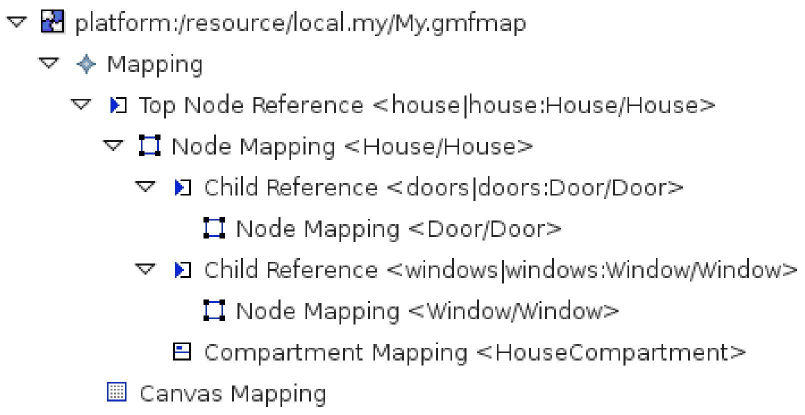
platform:/resource/local.my/My.gmftool
Tool Registry
Palette ToolPalette
Tool Group CreationToolGroup
Creation Tool House
Creation Tool Door
Creation Tool Window

Figure 1.8: The outline of a tooling definition model.

menu, a context menu, and so on. The tool palette in turn contains the available tools. These may also be seperated in groups to improve the graphical structure. A tooling definition model for SyncCharts might for example contain two creation tools: one for states and one for transitions. Figure 1.8 depicts a simple example of a tooling definition model.

The mapping definition model The mapping definition model is responsible for linking the graphical definition model and the tooling definition model with the Ecore model. Its top level element is the *mapping*. The most important contents of the mapping are the canvas mapping, the top level references, and the link mapping. The canvas mapping determines which model element the diagram surface represents. Top level references contain node mappings, which link the elements of the Ecore model that may be put directly onto the diagram surface with their corresponding graphical representation and the tool that is used to create them. Link mappings do the same with model elements that are used to connect two nodes in the diagram. Node mappings may contain child references and compartments. The child references are defined in the same way as top level references but are also assigned a compartment in which they are supposed to show up. Compartment mappings in turn are linked to their corresponding compartments in the graphical definition model. In the case of SyncCharts, a simple mapping definition model could for example link states with the circle figure from the graphical definition model and the state creation tool from the tooling definition model and additionally add a compartment and a child reference to the state mapping, where the child reference points to the state mapping itself, so that states can contain other states. Also, a link mapping could link the transition class with the arrows from the graphical definition model and the transition creation tool from the tooling definition model. Figure 1.9 shows a small sample mapping definition model.

Generating the code When all of the above models are defined, the Ecore model, the graphical definition model, the tooling definition model, and the mapping definition model are merged into a *diagram editor generation model*. This model contains all the information needed to generate the editor code. It also provides some more



```
platform:/resource/local.my/My.gmfmap
  Mapping
    Top Node Reference <house|house:House/House>
      Node Mapping <House/House>
        Child Reference <doors|doors:Door/Door>
          Node Mapping <Door/Door>
        Child Reference <windows|windows:Window/Window>
          Node Mapping <Window/Window>
        Compartment Mapping <HouseCompartment>
    Canvas Mapping
```

Figure 1.9: The outline of a mapping definition model.

possibilities to further modify code generation that are not available in the aforementioned models. Finally, the diagram editor generation model is used to generate the code of the diagram editor.

The structure of GMF To fully understand this paper, it is important to know about the underlying structure of GMF diagram editors. It is shown in Figure 1.10. Basically, a GMF editor can be divided into two parts: The static, permanent part and the dynamic, non-permanent part. The permanent part consists of the domain model and the notation model. The domain model is the semantic model of the entity that is to be modified with the editor. It is an instance of the meta-model and also written in XML containing all semantical information of the represented entity. The notation model contains notational information such as the position and size of nodes etc. It is also written in XML and forms the textual representation of the diagram file. There exists an element in the notation model for every element in the domain model.

The dynamic part of a GMF editor consists of all elements that are dynamically created during runtime. On the one hand, there are the edit parts and on the other hand there are the figures. The figures are simply the graphical representations of the model elements. The edit parts are some kind of controller. Every time a diagram is modified, the modification that is to be applied passes through the edit part that corresponds to the element that is to be changed. The edit part then checks if the modification is valid and if, this is the case, forwards it to the notation model, which adapts to the changes and in turn passes the modification on to the domain model, which also adapts to it. Then, the edit part notifies its corresponding figure of the change, so that it can react to it and change its size or position for example. Note that for every edit part there exists one corresponding figure as well as one element in the notation and in the domain model. Each of the four entities provides just a different view on the element.

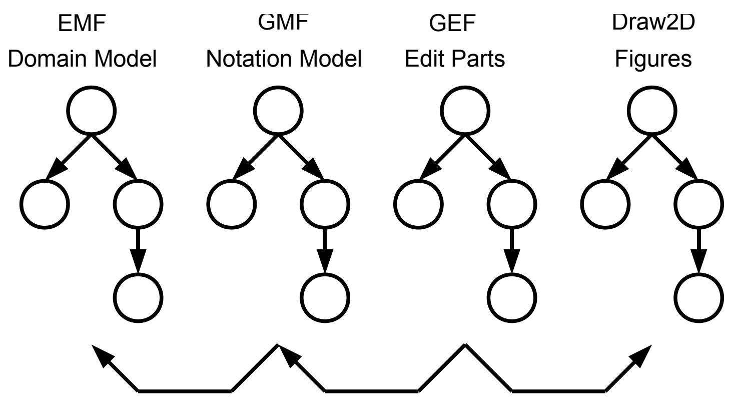


Figure 1.10: The relations between domain model, notation model, edit parts, and figures.

```
State :
  name = ID "/" (innerStates+=State ("," innerStates+=State)*)?
;
```

Figure 1.11: An XText grammar rule.

1.1.5 XText

The XText[6] project by openArchitectureWare provides functionality to create textual model editors. This is done by defining a grammar in the XText grammar language and then letting the editor be automatically created with the click of a button. The XText grammar language resembles an extended Backus-Naur-Form. A particular grammar consists of a list of rules that not only determine which characters an expression may be comprised of, but also to which attributes of a model element the expression is assigned. A small example will clarify this:

Figure 1.11 shows a rule that determines how a state may be defined in a textual way. It consists of an ID, which is basically a string but with a more limited character selection, followed by a slash and a list of inner states, each of them seperated by a comma. The ID is assigned to the attribute 'name' of the state, while the inner states are added to a list of states within the state class that is called 'innerStates'. As this is supposed to be a simple example, transitions, signals, and other things are omitted.

The XText syntax The syntax of XText is easy to understand. The expressions required to understand the paper are shown in Table 1.1.

[6] http://www.eclipse.org/Xtext/

'(' and ')'	Round brackets are used to group expressions.
'|'	The vertical bar denotes an 'or', which means that it is possible to write either the expression to the left or the one to the right of the '|'.
'?'	The question mark is used to denote that the expression to the left might optionally be written.
'*'	The asterisk is the sign for an arbitrary amount, which means that the expression to its left side may be written an arbitrary number of times in sequence. It is also allowed to omit the expression.
'='	This is used to express an assignment. The expression to the right is evaluated and its value is assigned to the feature on the left. It also implies that the element for which this rule is defined contains the denoted feature .
'+='	The '+=' represents the addition of the element to the right to the feature on the left. It implies that this rule's element contains a reference to a list of the appropriate type, which is named like the feature on the left side of the '+='.

Table 1.1: The XText syntax.

The advantage of the generated XText editor is that its parser automatically creates instances of the parsed elements. This is used by the ThinKCharts editor to parse strings contained in labels and integrate the created model elements into the current model.

1.1.6 The Kiel Integrated Environment for Layout Eclipse Rich Client

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[7] project provides mechanisms to improve model-driven development. It integrates into Eclipse and makes use of the aforementioned technologies EMF, GEF, and GMF. This ensures that every functionality offered by the KIELER project can be applied to any model editor that was generated with GMF. As described by Spönemann et al. [9], the basic idea of KIELER is to provide automatic layout mechanisms to make model-based design of software systems more comfortable and thus increase productivity. However, there are many other fields of model-driven development tackled by KIELER that are currently under development. In the future it will provide means of structure-based editing as well as model simulation and a so-called view management, which among other things supports automatic highlighting and zooming during simulations. It will also offer editors for several modeling languages, graphical as well as textual. The ThinKCharts editor is one of those graphical editors created with GMF. The code generated by GMF was modified in several aspects to make editing of SyncCharts with ThinKCharts as natural as possible.

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

1.2 Problem Description

An important feature that needs to be implemented is *attribute awareness*. Automatically generated GMF editors provide tools to create nodes and connections between those. Every node as well as every connection has a designated appearance, a figure in GMF terms, which is the only one it can ever adopt. However, in SyncCharts diagrams there are simple states and complex states. The former are depicted by an ellipsis containing the state's name, while the latter are represented by a rounded rectangle which contains not only the state's name but also its inner states, local signal declarations and so on. For automatically generated GMF editors there is only one possibility to support the two different appearances, namely to provide two different tools, one to create simple states and one to create complex states. While this is a valid approach, a more convenient way would be to provide only one tool to create states and an attribute to determine whether the state is a simple or a complex one. When this attribute is changed by the user, the state should automatically adapt its appearance according to the new value. The ThinKCharts editor uses this approach to improve the process of creating SyncCharts and increase productivity.

Definition of transitions Another feature is the convenient definition of transition labels, which contain a trigger, emissions, and assignments. The usual way to define those is to write down the expression that makes the trigger, followed by a slash, a list of signals that are to be emitted, and a list of variables along with the expressions that are to be asigned to them. Automatically generated GMF editors do not allow such a method. Since transition labels contain elements that have to be created dynamically, the only way to define them is to create triggers, emissions, assignments, and every expression with its own node creation tool and assign them to each other and the transition. The ThinKCharts editor provides a custom parser that is linked to transition labels and allows users to define transition triggers, emissions, and assignments in the usual way.

Keeping SyncCharts consistent The ThinKCharts editor is the successor of Arne Schipper's Safe State Machine (SSM) editor that was also developed within the KIELER project [8]. The SSM editor offers a means of creating and editing SyncCharts that is completely in line with GMF. This does not only mean that it refrains from attribute awareness and a custom parser, but also that its underlying meta-model has to be very complex in order to make sure that no invalid SyncCharts can be created. This lead to a huge number of similar classes, which are distinguished by the way they are used. The meta-model of the SSM contained for example the classes *Transition*, *TransitionTriggerable*, and *TransitionImmediateable* to distinguish between transitions that may have triggers, those that may be immediate, and all other transitions. Then, all other kinds of transitions like *Strong- and WeakAbortion-Transitions* extend those three classes. The same problem occurs with the different kinds of states.

The approach of the ThinKCharts editor is different: The underlying meta-model is kept as small as possible, thus there is only one class *State* that represents all kinds of states and also only one class *Transition* that is the representative of all possible transitions. In order to keep the meta-model small, it even allows some configurations that would lead to invalid SyncCharts. For example, it is possible to declare a pseudo state final, although this is not allowed for pseudo states. To ensure the correctness of every created syncChart, the Synccharts editor makes use of a validation feature that can be triggered by the user and checks the syncChart for inconsistencies.

There were also some smaller problems that occurred during the development of the ThinKCharts editor, such as the need to provide a custom layout for states or integrating the modifications made to the editor into the code generation process to enable the editor to be regenerated automatically with a few mouse-clicks. These are discussed in appropriate sections.

1.3 Related Work

This section introduces some projects that are related to ThinKCharts. These are the Generic Eclipse Modeling System (GEMS)[8], and the Generic Modeling Environment (GME)[9].

1.3.1 Generic Modeling Environment

The Generic Modeling Environment (GME) has been developed at the Institute for Software Integrated Systems at Vanderbilt University [6]. It is a tool for developing modeling and code generation environments and as such enables users to define new modeling languages. Like Eclipse it has a modular architecture and can be extended by attaching new software components to the existing ones. However, GME makes use of the Component Object Model (COM), a technology developed by *Microsoft* that allows interprocess communication and dynamic creation of objects. Despite the goal of the COM technology to provide platform independence, it relies on hardware that is supported by the family of *Windows* operating systems. As a consequence, GME is not platform independent. This a major disadvantage, since in the area of model driven development, it is desired that models can be exchanged between different contexts. Some efforts were made to combine GME with the flexibility of EMF [2]. These finally lead to the development of the Generic Eclipse Modeling System.

1.3.2 Generic Eclipse Modeling System

The Generic Eclipse Modeling System (GEMS) is an open source project developed by the Distributed Object Computing Group at Vanderbilt University's Institute for

[8]http://www.eclipse.org/gmt/gems/
[9]http://w3.isis.vanderbilt.edu/Projects/gme/

Software Integrated Systems [10]. Unlike GME, GEMS is a subproject of Eclipse written in Java and makes use of EMF. This way it provides the exchangability of models that its predecessor GME was missing. In addition, it is able to read models developed in GME. A drawback of GEMS compared with GMF is its lack of customizability. The code generation process of GEMS is much simpler than that of GMF and involves only one model, namely the meta-model. There is no place for other models, which further specify the functionality and visuals of the generated editor like the graphical, tooling, and mapping definition model in GMF. It is possible to modify the visuals of the generated editor by employing stylesheets. However, those are a little harder to use than the models of GMF, at least in the opinion of the author.

The remainder of this paper deals with the development of ThinKCharts with GMF. Section 2 provides a deeper insight into the problems discussed in Section 1.2 and introduces the strategies that were used to solve those problems, while Section 3 explains how the solutions were implemented and illustrates their structure and behavior. Finally, Section 4 concludes the discussion with a summary and an outlook on future work.

2 Concept

This section discusses the solutions to the problems encountered during the development of the ThinKCharts editor on a conceptual level. Section 2.1 covers the attribute awareness of diagram figures, Section 2.2 deals with the parsing of transition triggers, emissions, and assignments, Section 2.3 discusses the validation mechanism included in the editor, and Section 2.4 finally covers some small challenges that had to be overcome.

2.1 Attribute Awareness

In standard GMF editors, the figures that represent model elements have only one fixed appearance. This appearance is defined in the graphical definition model. For example, this might be a rectangle with a black outline and white background that represents a state. Or it might be a circle in the same colors, but it is not possible to assign both representations to states. This has some negative implications on SyncCharts editors generated with GMF. In SyncCharts, there are elements that have more than one appearance, while their current appearance depends on their current status. States for example have an elliptic form, if they are simple, which means that they do not contain other states, signals, variables, actions, or other things. However, states that contain such things are represented as rounded rectangles with their inner states, signals, and others depicted within themselves. Also, initial states are supposed to have a thicker border than normal states and final states are depicted with a double border. Pseudo states have a completely different appearance because they do not display a name, but are only small grey circles with a 'P' inside. Transitions also have different appearances. Weak abortion transitions are simple black arrows, strong abortion transitions feature a little red circle at their start point, and normal termination transitions have a little green triangle in the same place. There are also history transitions, which have a grey cirle with an 'H' inside at their end point. Figure 2.1 shows the possible appearances of states and transitions mentioned above.

The default Behaviour of GMF Editors If a standard GMF editor is supposed to support all the aforementioned different kinds of states and transitions, every single one of them needs to get its own class in the meta-model as well as its own figure in the graphical definition model, and its own creation tool in the tooling definition model. As a result, such a GMF SyncCharts editor would be littered with lots of different tools and all of them would either create a state or a transition.

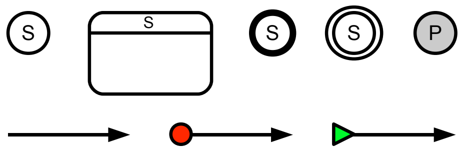


Figure 2.1: Different appearances of states and transitions.

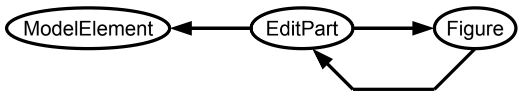


Figure 2.2: Relation between an attribute-aware figure, its edit part, and model element.

Furthermore, since all those different states and transitions are completely different model elements, if a simple state in a syncChart is supposed to be substituted for a final state, the simple state has to be deleted before the final state is newly added. As a consequence, all incoming and outgoing transitions of the simple state will be deleted as well and the user has to reenter them by hand. This issue is even more problematic in the case of complex states with lots of inner states, signals, variables, etc. Just to change such a state from simple to final, all its contents have to be deleted along with the state itself and then inserted again.

Making Figures aware of their Model Elements To avoid so much trouble, the ThinKCharts editor uses another approach, which is called 'attribute awareness'. Its goal is to make the figures depicted in the editor sensitive to changes that are made to their corresponding model elements. As it has already been explained in Section 1.1.4, the edit parts form the controlling layer, so every change that is to be made to a model element first passes through its corresponding edit part. After the model element has been changed, it notifies every adapter that is registered in its adapters list of that change. The key is to let the model element's figure be notified of every change done to the model element by adding it to the model element's adapters list. However, this is not that easy, since edit parts contain references to the model elements they control, but figures do not know their controlling edit parts. To remedy this, every edit part that creates a figure to represent itself has to pass itself as a parameter to the created figure, so the figure can retrieve its model element by asking the edit part and then register itself as an adapter of that model element. Figure 2.2 illustrates the relation between an attribute-aware figure and its edit part and model element.

Reacting on Changes in the Model The next step is to make the figure react on changes done to the model element. In order to implement this, the figure inspects the model element every time it has been notified of a change. Then it evaluates the current status of the model element and adapts its appearance accordingly. The evaluation takes place in the following way: Every attribute aware figure contains a list of possible appearances. Every one of these appearances is accompanied by a list of conditions that have to be true in order for the appearance to be valid. The figure searches through all of its possible appearances and evaluates their conditions. If all of them equal true for one of the appearances, that appearance becomes the current one and the figure updates itself. If several appearances are valid, the first one in the list is taken. If none of the specified appearances is valid, the figure falls back to a predefined default appearance. Note that with this approach it is not necessary to save all possible combinations of features along with its corresponding appearances. Only the cases in which a figure has to take on a special appearance are to be defined and in every of those cases only the conditions that have to evaluate to true have to be specified. This way a lot of memory space might be saved.

2.2 Parsing

The usual way to define triggers, emissions, and assignments in SyncCharts is to write them next to the corresponding transition. If an editor is supposed to support such input methods, it has to dynamically create the denoted model elements and assign them to the transition. However, automatically generated GMF editors do not support the creation of complex model elements through text fields. It is only possible to specify the values of an element's attributes in such text fields. Those fields may even contain several attributes at once where their representation can be specified with patterns that provide a format for the contained text. However, whole model elements always have to be created as nodes or connections in the diagram, through drag-and-drop, context menus, or popup buttons. As a consequence, every trigger, assignment, emission, and even expression has to be represented by a node. Additionally, to link expressions, triggers, transitions and the like to each other, connections are needed. So if a transition is to be created along with the label 'I1 AND I2 / O', the user has to create a node for the trigger 'I1 AND I2', and then one for the signal emission 'O'. Three more nodes also have to be created: One for the signal 'I1', one for the signal 'I2' and one for the operator 'AND'. Then, 'I1' and 'I2' have to be connected to the operator 'AND' to denote that they are its operands and the operator 'AND' has to be connected to the trigger 'I1 AND I2'. Even in this small example the structure that is to be created is quite big and its creation demanding. Figures 2.3(a) and 2.3(b) compare both notation variants visually. Of course, in real SyncCharts transitions are often much more complex, so the standard GMF method of using nodes to represent every model element is insufficient as it is too tedious to repeatedly specify transitions in such a manner.

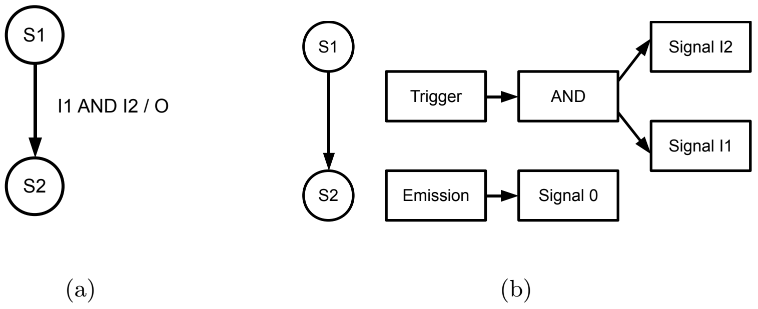


Figure 2.3: A transition with its trigger and emission defined in a label (left) and in separate nodes (right).

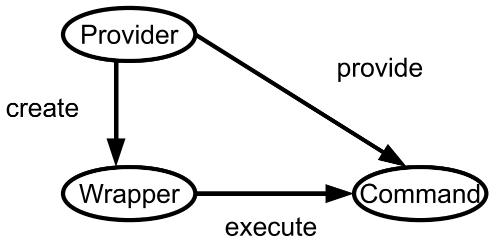


Figure 2.4: Relations between provider, wrapper, and command.

Extending the Functionality of Labels The ThinKCharts editor extends the behavior of transition labels to allow the usual way of definition. This is also true for actions, since they also have triggers, emissions, and assignments. ThinKCharts provides a custom parser, which parses the contents of appropriate labels, creates the elements specified therein, and returns them. These elements are then checked for inconsistencies and if they are valid are integrated into the already existing model. To accomplish this, ThinKCharts makes use of openArchitectureWare's XText project [3]. It allows to specify a grammar and generates a textual editor from it. For this project not the whole editor is of interest, but only the generated parser. Unfortunately, GMF cannot deal with XText parsers because the parsers it uses itself are different from the generated ones. To solve this problem, a wrapper was created that encapsulates the XTextParser and provides all the functionality GMF expects from a parser. In addition, a custom parser provider was made that takes care of the texts that are to be displayed within the label, determines which events cause the parser to be executed, and also creates a new Wrapper and provides the custom XText command, which handles the creation of the elements specified in the label text. Figure 2.4 illustrates the relations between the wrapper, the provider and the command.

2.3 Validation

As has been explained in Section 1.1.4, the meta-model is the core of any GMF editor. The meta-model the ThinKCharts editor uses to model SyncCharts has been kept as small and easy to understand as possible. However, this approach brought some disadvantages with it: With the current model, it is possible to create SyncCharts in the ThinKCharts editor, which are not valid in the sense of the usual SyncCharts semantics. Usually, the meta-model itself is supposed to prevent inconsistencies. Unfortunately, in the case of SyncCharts this leads to a very big, complicated meta-model. One example are the different kinds of states that exist: States can be initial, final, pseudo states, textual states, reference states, or simply normal states. Also combinations of those are allowed, such as initial reference states or final textual states. However, not every possible combination is valid. Pseudo states, for example, can neither be initial nor final. To prevent invalid combinations to occur, it would be necessary to give states an attribute with an enumeration type that lists every possible combination of the above values, so that only one of those could be chosen. Indeed, it is better from a logical point of view to divide the different kinds into two groups, one group consisting of 'normal', 'initial', and 'final' and the other consisting of 'normal', 'pseudo', 'reference', and 'textual'. The disadvantage of this approach is that there is no possibility to express constraints in the meta-model like 'if a state is a pseudo state it is not allowed to be final'. Another way of preventing inconsistencies to occur would be to create different classes for different kinds of states, which means there would be one class for normal states, one for pseudo states etc. Apparently, this approach would impair the readability of the meta-model to a great amount.

The Built-In Validation Mechanism To keep the meta-model as small as possible and prevent inconsistencies from occuring, the ThinKCharts editor makes use of a validation mechanism that comes with GMF. This mechanism allows to specify so-called check files in openArchitectureWare's *Check* language [3]. These files consist of a number of rules, which define constraints that have to hold and a warning or error message that is to display in case the constraint does not hold. Figure 2.5 shows a simple example of such a check rule. The depicted rule determines that states, which are of the kind 'pseudo', have to either have no name at all or have a name with the length 0. If this is not the case, the message "pseudo states must not have a name!" is displayed. Although the sample rule is very simple, the *Check* language allows to specify very complex expressions, since it supports a *Collection* type together with many operators such as the universal quantifier, means to filter objects from collections and so on. The validation mechanism can be activated in the diagram editor generation model and then accessed by the user through the 'Validate' entry in the menu bar.

```
context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not have a name!" :
   (name == null) || (name.length == 0);
```

Figure 2.5: A check rule.

2.4 Further challenges

This sections covers some smaller challenges that had to be overcome. Section 2.4.1 deals with the integration of modified code into the code generation process and Section 2.4.2 discusses a custom layout that had to be implemented in order to represent SyncCharts in the usual manner.

2.4.1 Code templates

When developing the ThinKCharts editor, great importance was attached to integrating all modifications that were made after the GMF generation process into the generation process itself. This way, no Java code has to be produced, but only the models that are needed to generate it. Most changes are made outside of the editor code itself, since the GMF models already provide means to change existing elements like figures or layouts by pointing to an external class that implements the custom behaviour. However, sometimes it was required to change the editor code itself. In those cases so-called code templates were deployed. These templates affect the code generation process of GMF by overwriting only certain parts of the code. There are templates for every possible kind of code file: Templates for nodes, connections, compartments and everything else that is to be generated. Fortunately, not the whole set of code templates has to be provided, but only those that differ from the usual templates.

2.4.2 Layout

Attribute aware state figures require a special layout since their contents are supposed to change position and size depending on the state's status: Simple states show their name in the center, while complex states arrange their name at the top and signal, variable, and other declarations underneath. If the state contains inner states, those are depicted at the bottom. Figure 2.6 illustrates this.

The Difficulties with Compartments In addition, the mechanism of compartments further complicates the layout, since on the one hand only the topmost compartment in a compartment stack is available to the user and on the other hand a set of compartments, which is distributed over the whole area of a state, makes the addition of further elements very tedious. Figure 2.8 illustrates this. A complex state has several compartments that contain either signals or variables or states etc. If these compartments share the free area within the state, then it is only possible to add a new signal to the state by clicking on the compartment that is supposed to contain signals. Since the borders of the compartments are not visible in order for states to appear in the usual manner, it is required to guess where the needed compartment is located. Figure 2.7 shows a simple and a complex state together with a sample distribution of their compartments. Of course, it would be possible to simply add borders to the compartments, but there is another way, which preserves the usual appearance of states and makes adding elements to states easy enough. Unfortunately, simply making all compartments the size of the state and laying them on top of one another makes the compartments at the bottom unavailable to the user. So it was decided to make compartments, which do not have any contents, disappear by giving them a size of 0 and let the compartments with contents share the available space. This has the advantage that empty compartments do not waste space. In order to make even empty compartments available to the user, a context menu was added that allows to add signals, variables, other states, and so on to a selected state by right-clicking it and selecting the appropriate option. This way, the usual appearance of states is preserved and the process of adding new elements to it is not too tedious.

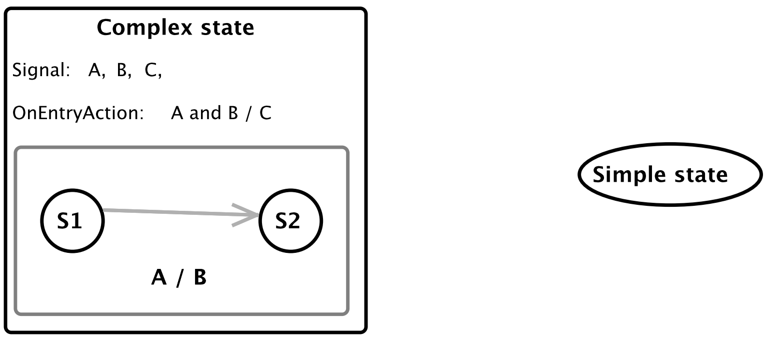


Figure 2.6: A complex state (left) and a simple state (right).

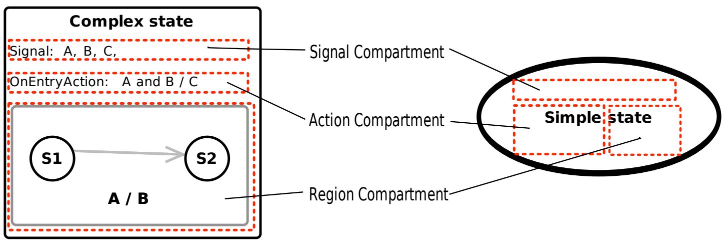


Figure 2.7: A sample distribution of a state's compartments.

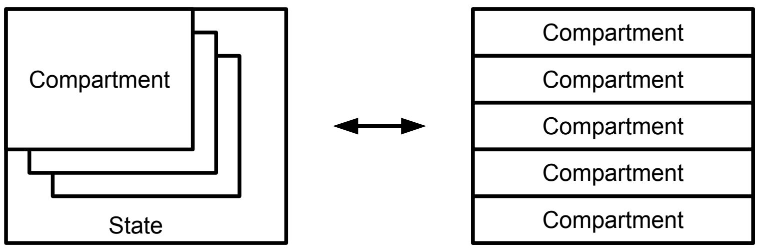


Figure 2.8: Different ways to layout compartments.

3 Implementation

This section discusses the implementation of the concepts that were introduced in the prior section in detail. Section 3.1 covers the process of generating a standard GMF editor. The next three sections deal with the three aforementioned topics attribute awareness, parsing, and validation, and Section 3.5 discusses the smaller challenges of code templates and layout.

3.1 GMF Code Generation

The following sections each discuss one of the models used during the code generation process.

3.1.1 The meta-model

The starting point of the whole GMF generation process is the meta-model, which is also called domain model and is contained in a file with the ending *.ecore*. Figure 3.1 shows the complete meta-model. The best place to start explaining the diagram is the class 'State'. A state contains several attributes: A name, which is of the type string, a stateFlag, and a stateKind. The stateFlag is of the enumeration type StateFlag and can be either 'normal, 'initial', or 'final'. The StateKind is also an enumeration type and contains the values 'normal', 'pseudo', 'reference', and 'textual'. Besides those attributes, a state also contains several references to other classes, the most important of them being 'outgoing transitions', which references an instance of the class transition. There is also the reference 'regions', which points to a list of regions. These regions in turn contain references to a list of states they contain. This way it is possible to realize hierarchy and parallelism in ThinKCharts, as regions represent the parallel components of a state and contain further states within themselves. States also contain references to a list of signals, variables, and three different references to lists of actions. These three kinds of actions are onEntry actions, onInside actions, and onExit actions, which are executed upon the appropriate events. Additionally, states may contain signal renamings, if they are of the type 'reference', and a suspension trigger, which determines a condition on which the state has to pause execution. A suspension trigger has a boolean attribute to specify whether it is immediate and a string to define the expression that causes suspension. Signal renamings contain references to two signals, the old one and the new one that replaces it. Note that all of the aforementioned references are containment references, except those signal references of signal renamings. This is because it is required that, except for the root element, every model element is contained in

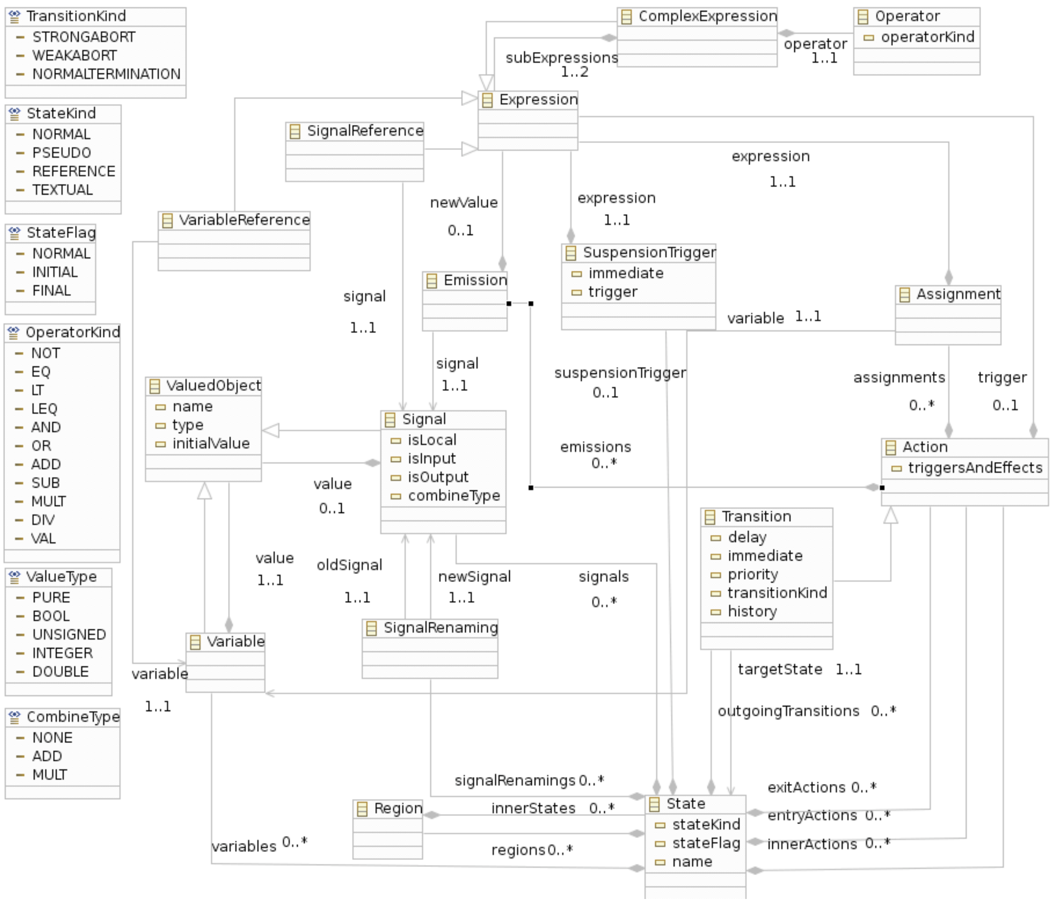


Figure 3.1: The ThinKCharts meta-model.

exactly one other element. Since signals are already contained in their parent states, the signal renamings have to simply reference them.

Valued Objects The classes 'Signal' and 'Variable' are both derived from 'ValuedObject', since they have some things in common: Both have a name, a type, and an initial value. Note that the meta-model does not distinguish pure signals from valued signals. The type of a valued object may either be 'pure', 'boolean', 'unsigned', 'integer', or 'double', which are the most used datatypes. In addition, signals contain also the three flags 'isInput', 'isLocal', and 'isOutput' to specify the kind of the signal and a combineType, which determines how the values of multiple emissions of the same signal are combined. Here, addition and multiplication are available.

Actions Actions contain a string which holds information about the action's trigger, emissions, and assignments. Note that this string has no semantical function as well as the 'trigger' string of the suspension trigger. They are needed in the meta-model because the ThinKCharts editor is supposed to show their information in the diagram. In standard GMF it is only possible to do this when the label in the diagram has access to some printable attributes of the model, for example a string or an integer. However, in the future the editor might be extended to construct the label text automatically from the 'trigger', 'assignments', and 'emissions' attribute. Actions may also contain an expression that represents the trigger and an arbitrary number of emissions and assignments. Emissions reference the signal that are supposed to be emitted and may contain an expression, which specifies the value that is to be assigned to the emitted signal. Assignments reference a variable that is to be changed and also contain one expression, which determines the new value to be assigned to the variable.

Transitions Transitions, which are derived from actions, may have a delay that determines how often its trigger has to become true in order for the transition to be taken. They may also have a priority of the type integer to specify which transition has to be considered first if there are some with overlapping triggers. The boolean attributes 'immediate' and 'history' determine if the transitions trigger is immediate and whether the transition should in case it leads to a complex state jump directly into the state that was last active within the complex state. Additionally, transitions feature a kind attribute, which can be 'weak abortion', 'strong abortion', or 'normal termination', and a reference to their target state.

Expressions Finally, there is the expression. An expression can either be a reference to a signal or a variable or a complex expression. Note that expressions cannot directly be a signal or a variable, because that would conflict with the requirement that every element has to be contained in exactly one other element. Since expressions are contained within actions, an expression that was a signal would be contained in its parent state and also in its parent action. Complex expressions feature one operator, which may be 'not', 'eq' (=), 'lt' (<), 'leq' (<=), 'and', 'or', 'add' (+), 'sub' (-), 'mult' (*), 'div' (/), or 'val' (?), and one or two subexpressions. In the case of unary operators such as 'not' and 'val', there would be only one subexpression, while in the other cases there would be two.

3.1.2 The graphical definition model

The graphical definition model seems complex at first because of its size, but it is straightforward to understand. The figure gallery contains four figures represented by figure descriptors: One figure for states, one for transitions, one for regions, and one invisible figure. The invisible figure is needed to be laid underneath labels that represent signals, variables and other text based elements. The reason for this is that those elements have to be layouted in a special way to depict SyncCharts in the usual manner. However, if a compartment contains only text labels it has to be a list compartment, which does not provide means to define a custom layout. So the text labels have to be children of real node figures, so that a shape node compartment can

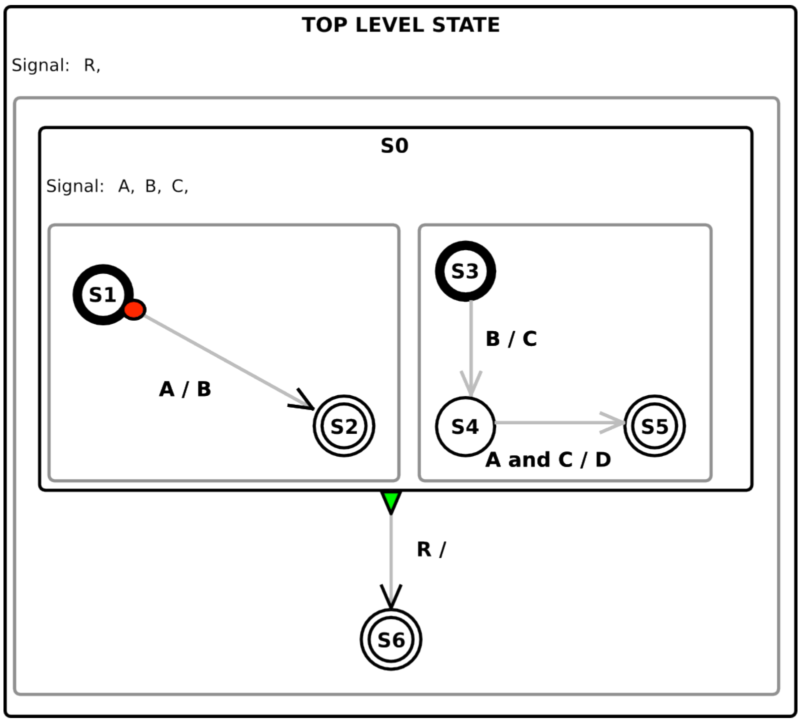


Figure 3.2: A syncChart that was created with ThinKCharts.

be used to hold them, which features a custom layout. Figure 3.2 shows a sample SyncChart created with ThinKCharts that features the most important model elements: Macro states are rounded rectangles, simple states ellipses or circles, regions are grey rounded rectangles. The invisible figures of the declared signals appear like simple text. In Figure 3.3, a complete syncChart is depicted with all of its graphical components.

All of the figures mentioned above are custom figures, which means they point to external classes that implement their behaviour. Except for the region figure they also contain each one label to represent textual attributes. These labels can be referenced from external elements through the child access contained in the figure. The state figure also features a custom layout, which is discussed later.

Other than the figure gallery, the canvas holds three different nodes, one for the states, one for the regions, and one for invisible figures, which represent all the textual nodes like signals, variables, etc. There is also one connection to represent the transitions and a lot of compartments and labels to hold the different elements and depict textual information such as state names, transition triggers, and so on. Figure 3.4 shows the outline of the graphical definition model.

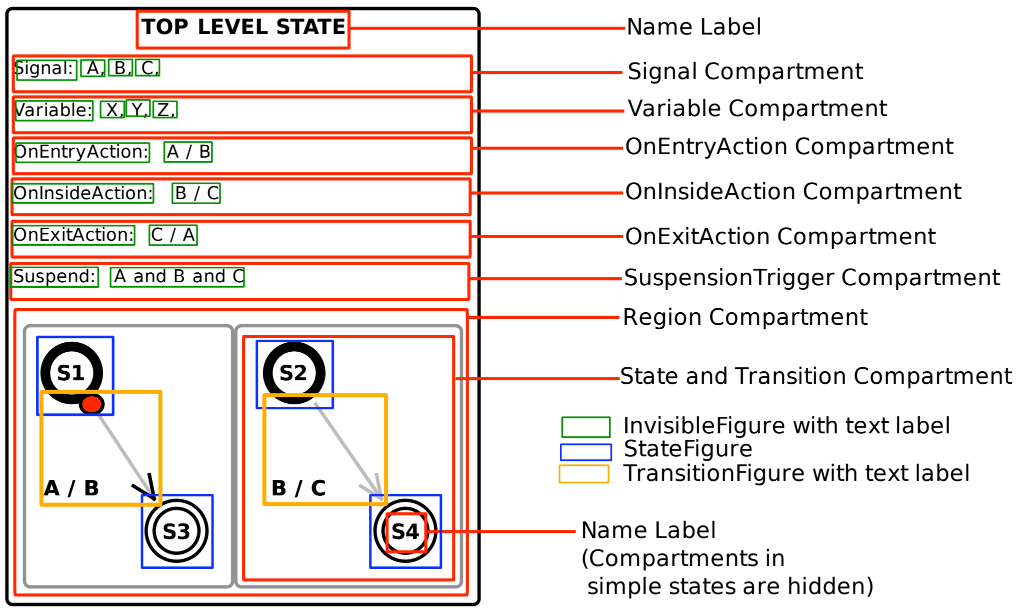


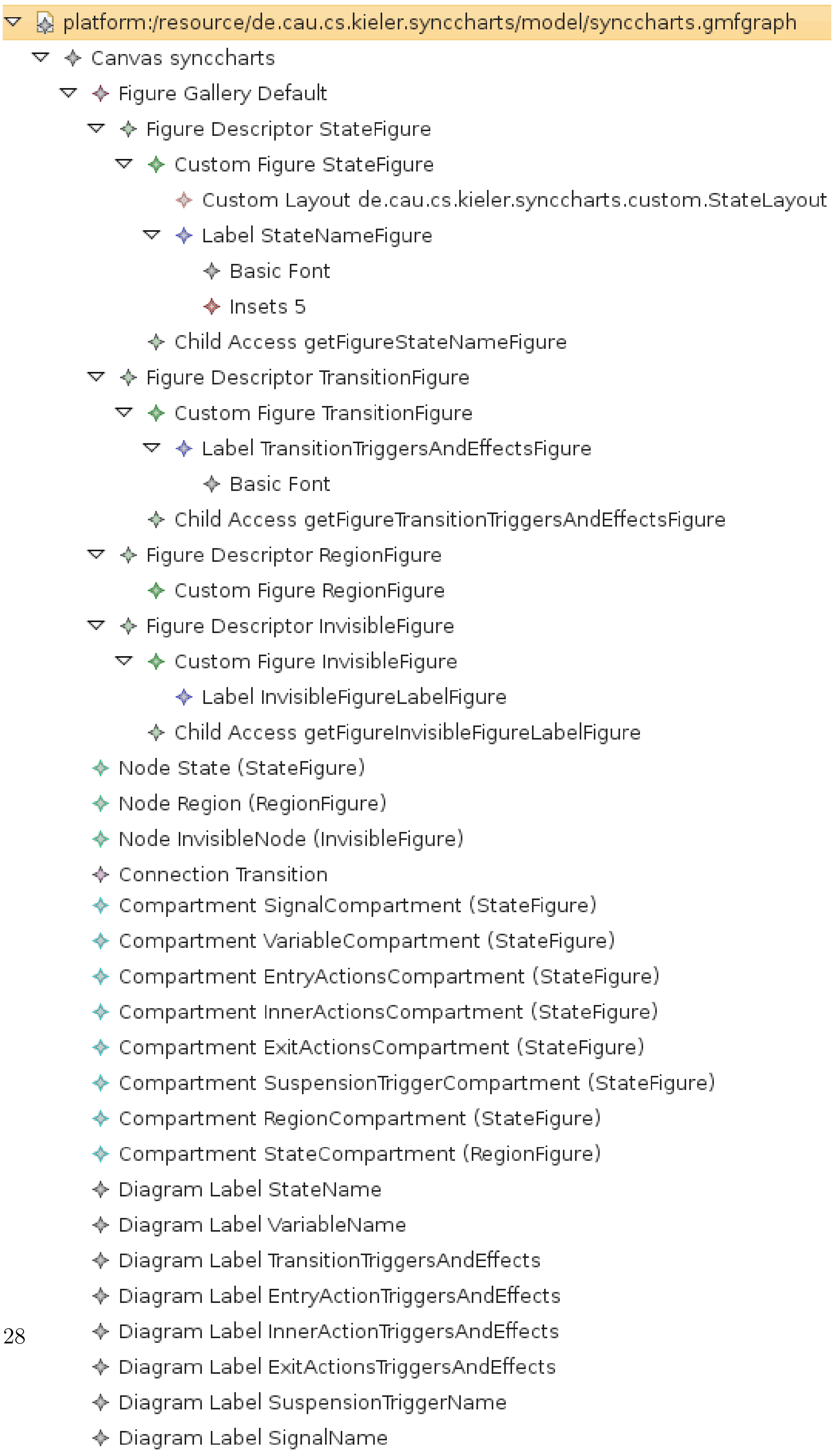
Figure 3.3: A complete SyncChart.

3.1.3 The tooling definition model

The tooling definition model is very compact and easy to understand. The tool registry holds only one palette with one tool group inside. This tool group contains two creation tools, one for the creation of states and one for the creation of transitions. All other elements like signals and variables are added to states through a context menu. The outline of the tooling definition model is depicted in Figure 3.5.

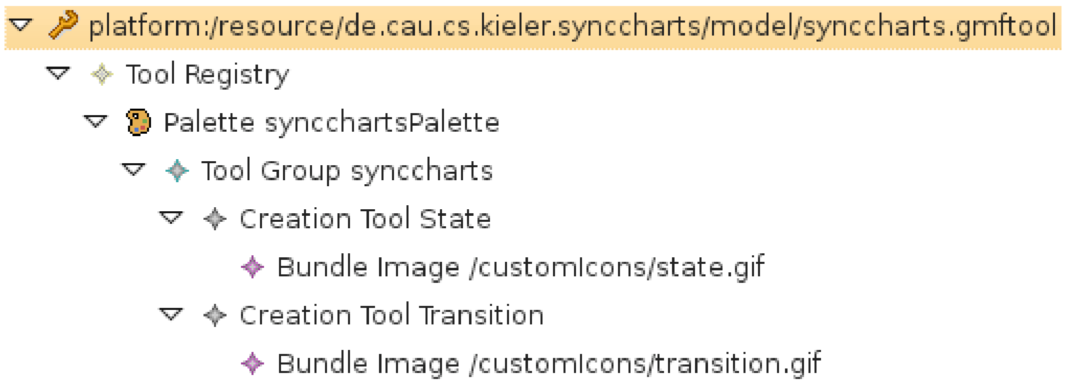
3.1.4 The mapping definition model

The mapping definition model links the meta-model, the graphical definition model, and the tooling definition model together. It contains a canvas mapping that maps the canvas to a region, a top node reference, and a link mapping, which assigns transitions their corresponding figure and tool. The link mapping also contains a feature label mapping to link the 'triggersAndEffects' string to the diagram label that is supposed to depict it. The top node reference contains a node mapping that maps states to their appropriate figure and tool as well. It has also a feature label mapping for the state's name. Then there are several child references and compartments, one of each for every model element that is supposed to show up in a state. Except for the child reference that points to regions every one of them contains a node mapping with a feature label mapping, just as the link mapping, since every one of those elements (variables, signals, etc.) need a label to depict one or more of their attributes. The compartment mappings link the child references



```
platform:/resource/de.cau.cs.kieler.synccharts/model/synccharts.gmfgraph
  Canvas synccharts
    Figure Gallery Default
      Figure Descriptor StateFigure
        Custom Figure StateFigure
          Custom Layout de.cau.cs.kieler.synccharts.custom.StateLayout
          Label StateNameFigure
            Basic Font
            Insets 5
        Child Access getFigureStateNameFigure
      Figure Descriptor TransitionFigure
        Custom Figure TransitionFigure
          Label TransitionTriggersAndEffectsFigure
            Basic Font
        Child Access getFigureTransitionTriggersAndEffectsFigure
      Figure Descriptor RegionFigure
        Custom Figure RegionFigure
      Figure Descriptor InvisibleFigure
        Custom Figure InvisibleFigure
          Label InvisibleFigureLabelFigure
        Child Access getFigureInvisibleFigureLabelFigure
    Node State (StateFigure)
    Node Region (RegionFigure)
    Node InvisibleNode (InvisibleFigure)
    Connection Transition
    Compartment SignalCompartment (StateFigure)
    Compartment VariableCompartment (StateFigure)
    Compartment EntryActionsCompartment (StateFigure)
    Compartment InnerActionsCompartment (StateFigure)
    Compartment ExitActionsCompartment (StateFigure)
    Compartment SuspensionTriggerCompartment (StateFigure)
    Compartment RegionCompartment (StateFigure)
    Compartment StateCompartment (RegionFigure)
    Diagram Label StateName
    Diagram Label VariableName
    Diagram Label TransitionTriggersAndEffects
    Diagram Label EntryActionTriggersAndEffects
    Diagram Label InnerActionTriggersAndEffects
    Diagram Label ExitActionsTriggersAndEffects
    Diagram Label SuspensionTriggerName
    Diagram Label SignalName
```

Figure 3.4: The outline of the ThinKCharts graphical definition model.



```
platform:/resource/de.cau.cs.kieler.syncharts/model/syncharts.gmftool
  Tool Registry
    Palette synchartsPalette
      Tool Group syncharts
        Creation Tool State
          Bundle Image /customIcons/state.gif
        Creation Tool Transition
          Bundle Image /customIcons/transition.gif
```

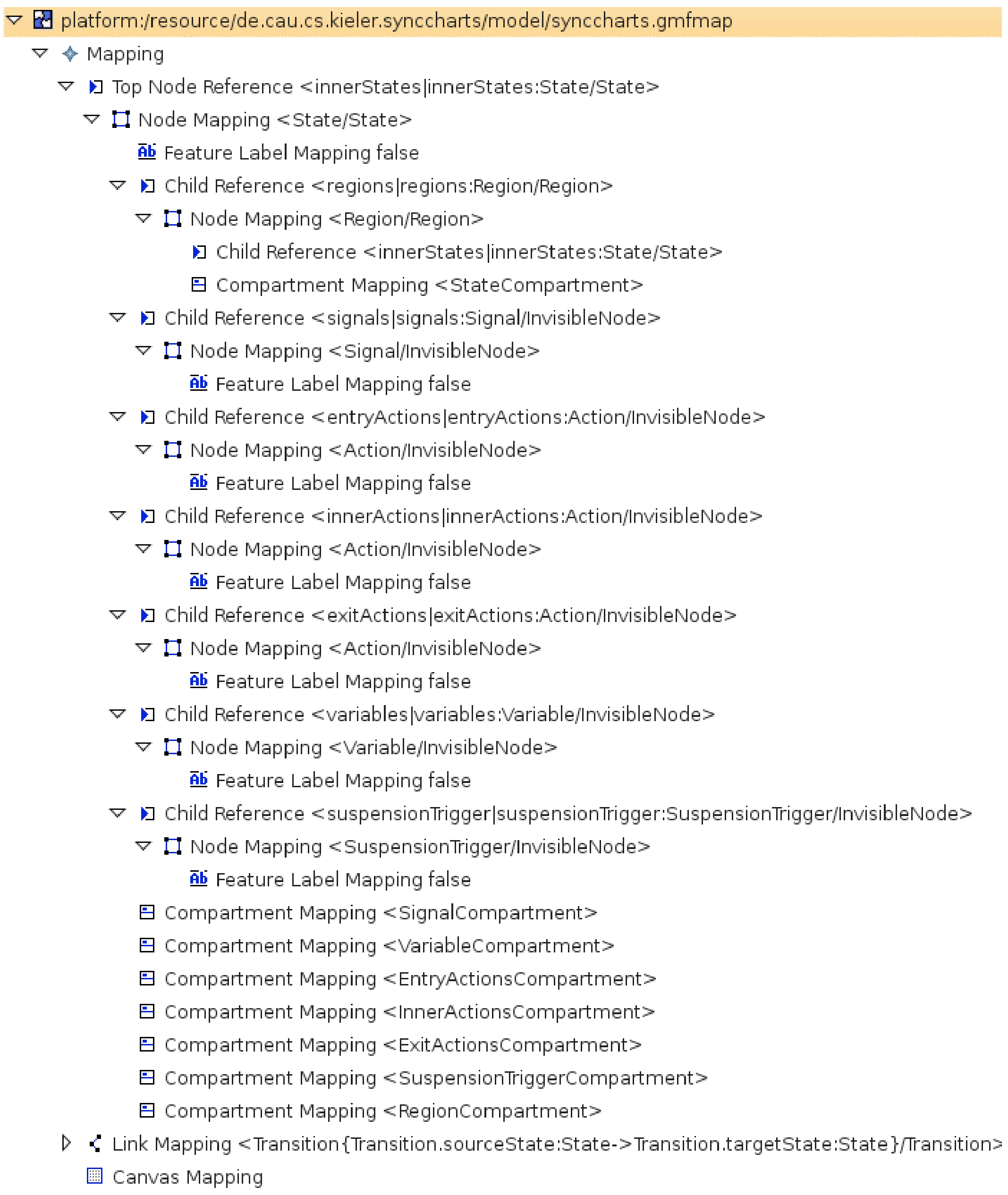
Figure 3.5: The outline of the ThinKCharts tooling definition model.

and the compartment figures in which they are supposed to show up. The region child mapping is a special case, since it contains a child reference pointing to states, so that regions may contain states. They also have a compartment mapping to link the child reference and the state compartment figure together. Figure 3.6 shows the outline of the mapping definition model.

3.2 Attribute Awareness

The code for attribute aware figures is located in a seperate project, which is called 'de.cau.cs.kieler.syncharts.custom'. To enable the generated diagram editor project to find the project, it has to be added to its dependencies first. It is also possible to enter this information in the diagram editor generation model, so that the editor code is already generated with this modification. Although almost all custom code is contained in the 'custom' project, some generated code has to be modified, in order for the attribute awareness to work: When the edit parts of attribute aware figures create those, they have to pass themselves as a parameter. Figure 3.7 shows this. Additionally, the constructor of the generated figure class within the edit part has to be modified. Since it extends the class 'AttributeAwareFigure', it calls the constructor of its parent class and at the same time forwards the edit part so it can be used to retrieve the corresponding model element. This modification to the code is entered in the diagram editor generator model. The body of the generated figure class is located in the inner class viewmap of each edit part. Figure 3.8 shows the modified code.

The AttributeAwareFigure The class 'AttributeAwareFigure' itself extends the Draw2D class 'Figure' and implements the adapter interface. In addition to the attributes of a Draw2D figure, it provides a list of so-called 'conditionalFigures'. This list contains all the information under which conditions which figure is the right one to be shown. It also adds a variable to hold its currently valid figure, its



```
platform:/resource/de.cau.cs.kieler.synccharts/model/synccharts.gmfmap
  Mapping
    Top Node Reference <innerStates|innerStates:State/State>
      Node Mapping <State/State>
        Feature Label Mapping false
        Child Reference <regions|regions:Region/Region>
          Node Mapping <Region/Region>
            Child Reference <innerStates|innerStates:State/State>
            Compartment Mapping <StateCompartment>
        Child Reference <signals|signals:Signal/InvisibleNode>
          Node Mapping <Signal/InvisibleNode>
            Feature Label Mapping false
        Child Reference <entryActions|entryActions:Action/InvisibleNode>
          Node Mapping <Action/InvisibleNode>
            Feature Label Mapping false
        Child Reference <innerActions|innerActions:Action/InvisibleNode>
          Node Mapping <Action/InvisibleNode>
            Feature Label Mapping false
        Child Reference <exitActions|exitActions:Action/InvisibleNode>
          Node Mapping <Action/InvisibleNode>
            Feature Label Mapping false
        Child Reference <variables|variables:Variable/InvisibleNode>
          Node Mapping <Variable/InvisibleNode>
            Feature Label Mapping false
        Child Reference <suspensionTrigger|suspensionTrigger:SuspensionTrigger/InvisibleNode>
          Node Mapping <SuspensionTrigger/InvisibleNode>
            Feature Label Mapping false
        Compartment Mapping <SignalCompartment>
        Compartment Mapping <VariableCompartment>
        Compartment Mapping <EntryActionsCompartment>
        Compartment Mapping <InnerActionsCompartment>
        Compartment Mapping <ExitActionsCompartment>
        Compartment Mapping <SuspensionTriggerCompartment>
        Compartment Mapping <RegionCompartment>
    Link Mapping <Transition{Transition.sourceState:State->Transition.targetState:State}/Transition>
    Canvas Mapping
```

Figure 3.6: The outline of the ThinKCharts mapping definition model.

```java
protected IFigure createNodeShape() {

   StateFigure figure = new StateFigure(this);
   return primaryShape = figure;
}
```

Figure 3.7: The modified constructor call.

```java
public StateFigure(EditPart e) {

   super(e);
   StateLayout layoutThis = new StateLayout();

   this.setLayoutManager(layoutThis);

   createContents();
   }
```

Figure 3.8: The modified constructor.

default figure, and its corresponding model element. The new method 'setModelElementAndRegisterFromEditPart' is responsible for using the edit part to look up the model element and register the figure in the model element's list of adapters. The most important adapter method to be implemented is the method 'notifyChanged'. It is called every time an attribute of the model element changes. This method searches through the list of 'ConditionalFigures' and checks if one of them is valid. If this is the case, it updates the current figure and calls the paint method. This one has been slightly changed, so that it does not draw the attribute aware figure itself, but the figure that is stored in the 'currentFigure' variable. Also a little fix had to be added, because the figures were always cut off at the left and top side. With the fix they are moved and resized a little so that they are shown correctly.

Attribute Aware Nodes The custom figures pointed to in the graphical definition model are those that extend 'AttributeAwareFigure'. Basically, they have only a modified constructor, which creates all the figures that are needed for the different appearances of the figure, create all the needed conditions and group all those in conditional figures. Then the default figure, current figure, and list of conditional figures are defined and the figure calls its own 'notifyChanged' method to update its appearance. Some figures also feature additional methods or override some of them. The state figure adds methods to decide whether its corresponding model element is a simple state, to check which of its references are not empty, and to extract its own name out of its label. The latter could also have been achieved by using the model element, but it is better to stay in the figure's domain whenever possible. These methods are used in the modified methods that calculate the figure's minimum and preferred size. Simple states have a fixed minimum height and base their minimum width on the length of their name. Complex states consider also the compartments that are not empty. For states the preferred size is the same as the minimum size. For the so-called invisible figures the minimum and preferred size is also the same. It is based on the size of the label contained in the figure.

Attribute Aware Connections Unfortunately, attribute aware connections need some special treatment, since polylines are considerably different from normal figures. They do not feature conditional figures, but conditional looks, which means that they adjust their start and end decorations depending on the status of the transition. This mechanism can easily be extended to also adjust things like line color, line width, and so on, but for the ThinKCharts editor it is sufficient to consider only the decorations. Although these changes are minimal, it is not possible for attribute aware connections to simply extend attribute aware figure, since they already extend the class 'PolylineConnectionEx' and Java allows only single inheritance. This causes some redundant code.

Conditional connection looks are different from conditional figures in that they do not feature a figure along with the list of conditions, but a start and an end decoration. The frequently mentioned conditions consist of a feature and value, which the feature is supposed to have. They also provide a method to check if they are fulfilled. Size conditions extend normal conditions and do not check for equality, but for the number of elements the specified feature contains. This is used for many referencing features that point to lists of elements. The name of every feature, such as the kind of a state or the target state of a transition, is specified in the generated SyncchartsPackage.

Figure 3.9 shows the relations between all the aforementioned classes.

Another possible approach To sum it up, the approach presented in this paper to make figures aware of changes in the model does fulfill its purpose. However, it has some drawbacks: The fact that attribute aware figures hold references to their model elements interferes with the design principle of GMF that visual elements should not need to know anything about the semantics. Further, it was necessary to modify the code generation by changing some templates. As has been found out later, this could have been avoided, because every edit part knows its parent and the topmost edit part holds a reference to the topmost model element. This means it is possible for any figure to get information about its corresponding model element by taking a circuit over its edit part, the parent edit parts, then the topmost model element, and from there search for its own model element. This is in fact a cleaner way to achieve the same goal and is likely to be used in future projects that rely on attribute awareness.

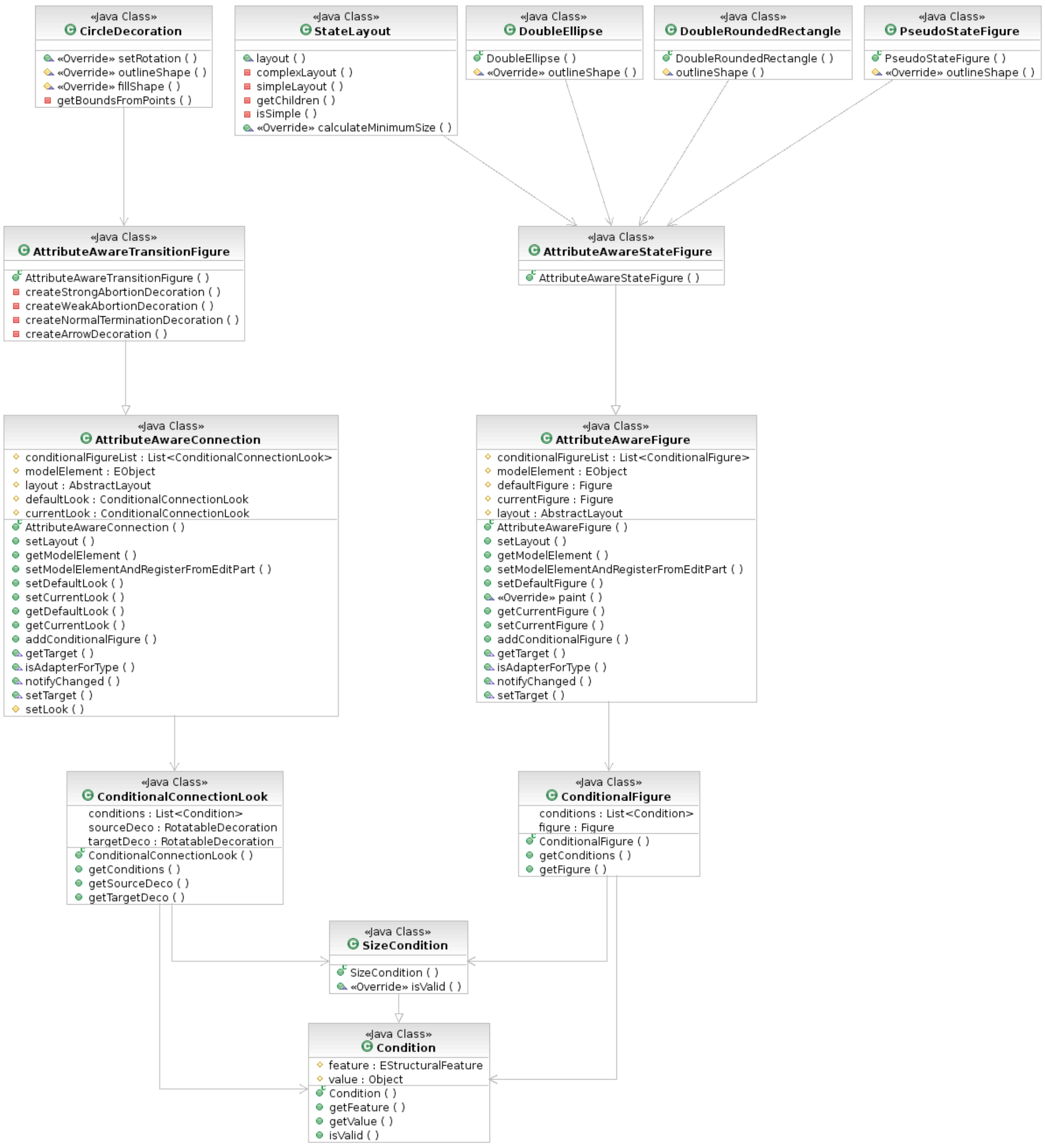

«Java Class» CircleDecoration
«Override» setRotation ( )
«Override» outlineShape ( )
«Override» fillShape ( )
getBoundsFromPoints ( )
«Java Class» StateLayout
layout ( )
complexLayout ( )
simpleLayout ( )
getChildren ( )
isSimple ( )
«Override» calculateMinimumSize ( )
«Java Class» DoubleEllipse
DoubleEllipse ( )
«Override» outlineShape ( )
«Java Class» DoubleRoundedRectangle
DoubleRoundedRectangle ( )
outlineShape ( )
«Java Class» PseudoStateFigure
PseudoStateFigure ( )
«Override» outlineShape ( )
«Java Class» AttributeAwareTransitionFigure
AttributeAwareTransitionFigure ( )
createStrongAbortionDecoration ( )
createWeakAbortionDecoration ( )
createNormalTerminationDecoration ( )
createArrowDecoration ( )
«Java Class» AttributeAwareStateFigure
AttributeAwareStateFigure ( )
«Java Class» AttributeAwareConnection
conditionalFigureList : List<ConditionalConnectionLook>
modelElement : EObject
layout : AbstractLayout
defaultLook : ConditionalConnectionLook
currentLook : ConditionalConnectionLook
AttributeAwareConnection ( )
setLayout ( )
getModelElement ( )
setModelElementAndRegisterFromEditPart ( )
setDefaultLook ( )
setCurrentLook ( )
getDefaultLook ( )
getCurrentLook ( )
addConditionalFigure ( )
getTarget ( )
isAdapterForType ( )
notifyChanged ( )
setTarget ( )
setLook ( )
«Java Class» AttributeAwareFigure
conditionalFigureList : List<ConditionalFigure>
modelElement : EObject
defaultFigure : Figure
currentFigure : Figure
layout : AbstractLayout
AttributeAwareFigure ( )
setLayout ( )
getModelElement ( )
setModelElementAndRegisterFromEditPart ( )
setDefaultFigure ( )
«Override» paint ( )
getCurrentFigure ( )
setCurrentFigure ( )
addConditionalFigure ( )
getTarget ( )
isAdapterForType ( )
notifyChanged ( )
setTarget ( )
«Java Class» ConditionalConnectionLook
conditions : List<Condition>
sourceDeco : RotatableDecoration
targetDeco : RotatableDecoration
ConditionalConnectionLook ( )
getConditions ( )
getSourceDeco ( )
getTargetDeco ( )
«Java Class» ConditionalFigure
conditions : List<Condition>
figure : Figure
ConditionalFigure ( )
getConditions ( )
getFigure ( )
«Java Class» SizeCondition
SizeCondition ( )
«Override» isValid ( )
«Java Class» Condition
feature : EStructuralFeature
value : Object
Condition ( )
getFeature ( )
getValue ( )
isValid ( )

Figure 3.9: Class diagram of the custom classes.

3.3 Parsing

Three objects are required to realize the parsing of labels by a custom XText parser: A provider, a wrapper, and a command. The provider is responsible for creating and returning the wrapped XText parser on demand. The wrapper encapsulates the functionality of the XText parser and provides the functionality required by GMF. The command creates the elements specified in the label text and integrates them into the existing syncChart. Figure 3.10 depicts the class diagram for those three classes.

Provider, Wrapper, and Command The provider is quite simple. It extends the class 'AbstractProvider'. The only methods needed to change are the 'provides' method, which determines if it provides a parser for a given operation, and the method 'getParser'. The latter is used by the former one and creates a wrapped XText parser to return it.

The wrapper implements the 'IParser' interface. It provides the five methods 'getParseCommand', 'getPrintString', 'getEditString', 'isAffectingEvent', and 'isValidEditstring'. These names of the methods explain their purpose: 'getParseCommand' simply returns the custom XText command. The method 'getPrintString' returns the string that is to be printed within the label when it is not currently edited. The returned string is the text of the action's or suspension trigger's label accompanied by a hash mark if the action or suspension trigger is immediate. The string that is returned by the 'getEditString' method, which means when the label is currently edited, is the same, except that the rhombus is omitted. The method 'isAffectingEvent' determines if a given event should activate the parser. This is the case if the event states that either the label of an action or the label of a suspension trigger has been changed. Finally, 'isValidEditString' determines if an entered string is valid. This is done by checking if the signals contained in the string are defined in one of the surrounding states.

The new command extends the class 'AbstractTransactionalCommand'. The only method to be modified is 'doExecuteWithResult'. First it uses the XText parser to parse the string and stores the returned action or expression. If the returned element is valid and all its signals are defined in the surrounding state the element is to be integrated in, it is integrated by simply rerouting some pointers.

The Communication To clarify the roles of the objects mentioned above, Figure 3.11 depicts a sequence diagram containing the communication that takes place upon the creation of a new action. As soon as the user has chosen the option to create a new action from the context menu, the provider creates a new wrapped XText parser. The wrapper checks if it has to react to the event. Since this is true, it waits for the new string to be entered, checks if the string is valid, and (supposed this is the case) retrieves the corresponding command that is to be executed. The command is the custom XText command. So the wrapper creates the command, the command is executed, and control is finally returned to the user.

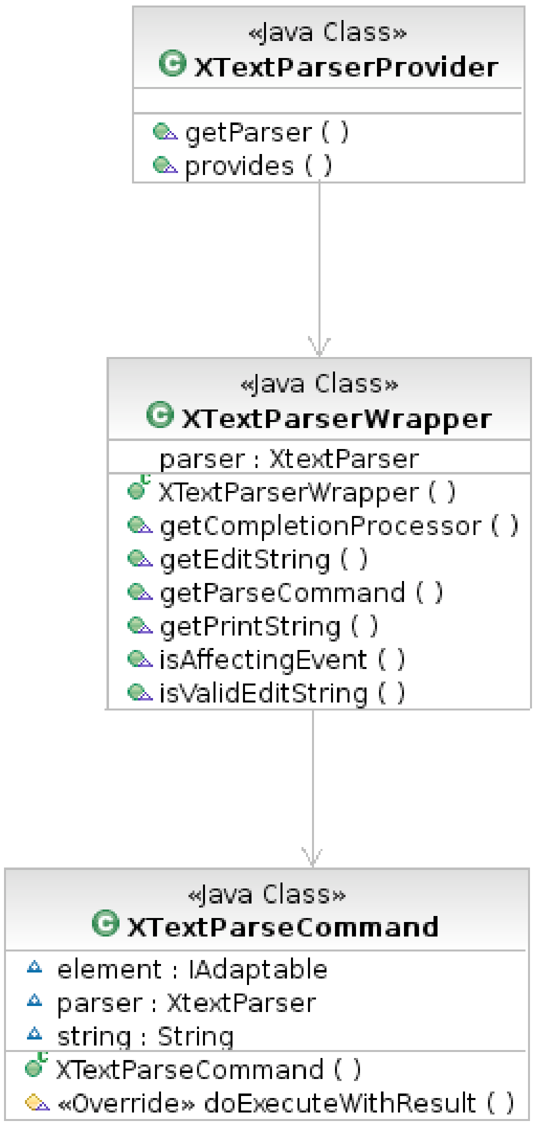


Figure 3.10: Class diagram of the parser classes.

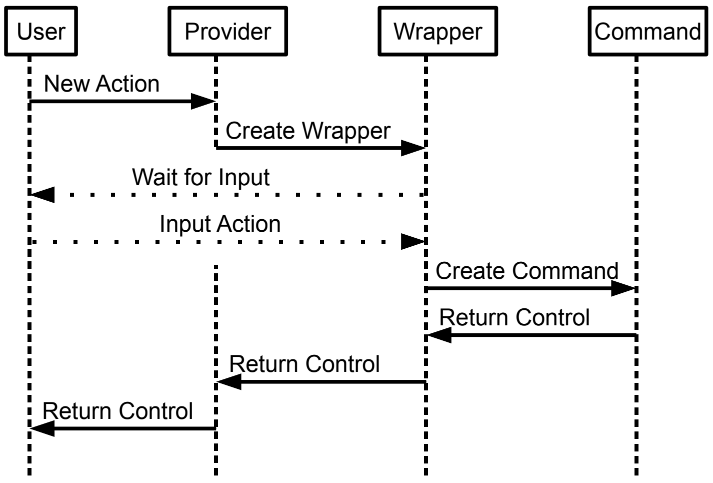


Figure 3.11: Sequence diagram of action creation.

The Parser Since the surroundings of the XTextParser have been covered above, the parser itself is discussed next. It has been generated from a grammar, which is depicted in Figure 3.12. First the ThinKCharts meta-model is imported. In order to publish the ThinKCharts meta-model, it is necessary to register its package in the workflow that generates the Java code for the parser. To accomplish this, the file named 'generate.oaw' has to be modified like shown in Figure 3.13. After the import of the meta-model, the grammar rules follow. The first rule defines what an action has to look like. It may contain an expression, which is assigned to the trigger reference of the action and may optionally be followed by a slash, which in turn is followed by an arbitrary number of emissions or assignments, which are added to the appropriate references. An expression may either be a signal reference, a variable reference, or a complex expression. An expression is always surrounded by brackets. This is necessary, because an expression like *A and B or C* would not be unambiguous since it is not clear if the *and* or the *or* is to be evaluated first. Within the expression are either an operator followed by another expression or two expressions around one operator. This allows to use unary and binary operators. An emission consists of a signal, optionally followed by an expression, which is surrounded in brackets. This enables the notation of pure and valued signals. An assignment contains a variable followed by a '=' and an expression. Signal and variable references simply contain a signal or a variable. Signals are represented by an ID and variables either by an ID or an INT (integer value). This makes it possible to define constants by using variables. An operator is defined by its operator kind, which may be one of the known enumaration literals. Note that the operators are written in the usual way, since characters like "+" and "?" are mapped to their textual equivalents.

```
importMetamodel "http://www.informatik.uni-kiel.de/rtsys/syncharts"
    as synccharts;

Action [synccharts::Action] :

   (trigger=Expression)? ("/" (effects+=Emission |
       effects+=Assignment)*)?;

Expression [synccharts::Expression] :

   SignalReference | VariableReference | ComplexExpression;

ComplexExpression [synccharts::ComplexExpression] :

   "(" ( subExpressions+=Expression)? operator=Operator
       subExpressions+=Expression ")";

Emission [synccharts::Emission] :

   signal=Signal ("(" newValue=Expression ")")?;

Assignment [synccharts::Assignment] :

   variable=Variable "=" expression=Expression;

SignalReference [synccharts::SignalReference] :

   signal=Signal;

VariableReference [synccharts::VariableReference] :

   variable=Variable;

Signal [synccharts::Signal] :

   name=ID;

Variable [synccharts::Variable] :

   name=ID | value=INT;

Operator [synccharts::Operator] :

   operatorKind=OperatorKind;

Enum OperatorKind [synccharts::OperatorKind] :

   NOT="not" | EQ="=" | LT="<" | LEQ="<=" | AND="and" |
       OR="or" | ADD="+" | SUB="-" | MULT="*" | DIV="/" | VAL="?";
```

Figure 3.12: The ThinKCharts grammar.

```
<workflow>
  <property file='generate.properties'/>
  <bean class="org.eclipse.mwe.emf.StandaloneSetup">
       <registerGeneratedEPackage value="de.cau.cs.kieler
        .syncCharts.SyncchartsPackage"/>
  </bean>
  <component file='org/openarchitectureware/xtext/
  Generator.oaw' inheritAll='true'/>
</workflow>
```

Figure 3.13: The modified generate.oaw.

3.4 Validation

The validation mechanism can be enabled by setting the options 'Validation Decorators' and 'Validation Enabled' in the topmost edit part of the diagram editor generation model to true. Additionally, a dependency to org.openarchitectureware.emf.check has to be added and the extension org.openarchitectureware.adapter.emf.check.checks. Then the meta-model with its nsURI and the check file have to be added to that extension. This is done in the manifest as seen in Figure 3.14. The only thing that is left to do is to fill the check file with rules. The following is a list of all inconsistencies that can be uncovered by the ThinKCharts editor followed by the check file in Figures 3.15, 3.16, and 3.17.

- Give pseudo states a flag other than 'normal', regions, signals, actions, or a suspension trigger.
- Give states, which are not reference states, signal renamings.
- Give states, which are not reference or textual states, a body text.
- Give states, which are not initial, no incoming transition.
- Give states neither a label nor an ID.
- Give a region more or less than one initial state.
- Make local signals input or output.
- Make a signal neither input, nor output, or local.
- Assign pure signals a combine type.
- Assign a delay to immediate transitions.
- Assign a delay less than 1 to a transition.
- Assign operators a wrong number of operands.

```
<plugin>

  <extension point="org.eclipse.emf.ecore.generated_package">
    <package
        uri="http://kieler.cs.cau.de/syncharts"
        class="de.cau.cs.kieler.syncharts.SyncchartsPackage"
        genModel="model/syncharts.genmodel"/>

  <extension
        point="org.openarchitectureware.adapter.emf.check.checks">
     <metaModel
          nsURI="http://kieler.cs.cau.de/syncharts">
        <checkFile
             path="model/SyncchartsChecks.chk">
        </checkFile>
     </metaModel>
  </extension>

</plugin>
```

Figure 3.14: The modified plugin.xml.

- Assign pure signals a value.
- Assign valued signals no value.
- Report parser errors.

3.5 Further challenges

This section covers some smaller challenges that had to be overcome during the development of the ThinKCharts editor. Section 3.5.1 discusses the integration of the modified diagram code into the generation process, Section 3.5.2 deals with the custom layouts, and Section 3.5.3 covers the modification of the context menu.

3.5.1 Code templates

Great value was placed on integrating all modifications, which were made after the GMF generation process, into the generation process itself. On some occasions, it was necessary to modify the generated code directly. To integrate these changes, code templates were employed. These are written in openArchitectureWare's Expand language. It is not necessary to understand the whole language to follow this paper. However, the most important key words are explained in Table 3.1.

There were only four templates that had to be adjusted. The modified templates are stored in a pre-defined folder hierarchy. The 'Use dynamic templates' option in

```
import syncharts;

// ===============================================================
// Pseudo States
context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not be initial or final!" :
   isInitial == false && isFinal == false;

context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not contain regions!" :
   (regions == null) || (regions.size == 0);

context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not contain signals!" :
   (signals == null) || (signals.size == 0);

context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not contain actions!" :
   ((entryActions == null) || (entryActions.size == 0))
   && ((innerActions == null) || (innerActions.size == 0))
   && ((exitActions == null) || (exitActions.size == 0));

context State if (type == StateType::PSEUDO) ERROR "pseudo
    states must not contain a suspension trigger!" :
   suspensionTrigger == null;

// ===============================================================
// Reference States
context State if (type != StateType::REFERENCE) ERROR "only
    reference states may contain signal renamings!" :
   (signalRenamings == null) || (signalRenamings.size == 0);

context State if (type != StateType::REFERENCE ||
    type!= StateType::TEXTUAL) ERROR "Only textual or
    reference states may contain body text!" :
   (bodyText == null) || (bodyText.matches("\\s*"));

// ===============================================================
// General States
context State if ((parentRegion != null || parentRegion
    .parentState != null) && isInitial == false) // not checked
    for root state ERROR "Not reachable state! Every state needs
    at least one incoming transition!" :
   (parentRegion.innerStates.exists(e|e.outgoingTransitions
      .exists(t|t.targetState == this)));

context State if ( label == null || label.matches("\\s*"))
    ERROR "Anonymous states (= no label) need an ID" :
   (id != null && id.length>0);
```

Figure 3.15: The ThinKCharts check file.

```
// ============================================================
// Regions
context Region if (parentState != null) // not checked for
    root state ERROR "Every region should have exactly one
    initial state!" :
   (innerStates.select(s|s.isInitial).size == 1);
// ============================================================
// Signals
context Signal if (isLocal) ERROR "local signals must not
    be input or output!" :
   ((!isInput) && (!isOutput));

context Signal ERROR "signals have to be input, local or
    output!" :
   isInput || isLocal || isOutput;

context Signal if ((initialValue == null)
    || (initialValue == ""))
    ERROR "only valued signals may have a combine type
      other than NONE!" :
   combineOperator == CombineOperator::NONE;

// ============================================================
// Transitions
context Transition if (isImmediate) ERROR "Immediate
    transitions must not have a delay!" :
   (delay == null) || (delay == 1);

context Transition ERROR "Delays have to be at least 1!" :
   (delay == null) || (delay >= 1);

// ============================================================
// Expressions
context ComplexExpression if ((operator == OperatorType::NOT)
    || (operator == OperatorType::VAL)) ERROR "Unary operators
    may not have more or less than one operand!" :
   subExpressions.size == 1;

context ComplexExpression if (!((operator == OperatorType::NOT)
    || (operator == OperatorType::VAL))) ERROR "Binary operators
    may not have more or less than two operands!" :
   subExpressions.size == 2;
```

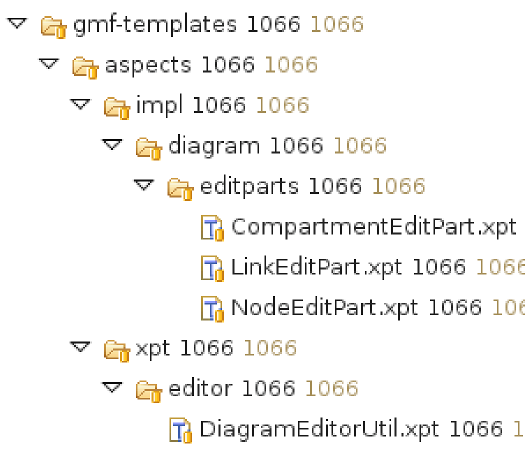
Figure 3.16: The ThinKCharts check file.

```
// ===========================================================
// Emissions

context Emission if (signal.type == ValueType::PURE) ERROR
    "Pure signals cannot be assigned a value!" :
   newValue == null;

context Emission if (!(signal.type == ValueType::PURE)) ERROR
    "Valued signals must be assigned a value!" :
   newValue != null;

// ===========================================================
// Actions

context Action if (triggersAndEffects != null &&
    triggersAndEffects.trim().compareTo("") != 0 )
    ERROR "Triggers and Effects String has not correctly been
   parsed yet!" :
   (trigger != null || (effects!=null && effects.size >0) );
```

Figure 3.17: The ThinKCharts check file.

IMPORT	The import instruction is used to load rules that have been defined in an external file.
DEFINE	The define keyword initiates a new code generation rule. Every rule defines one method. It is followed by the rule's name. This is also the name of the method that is generated. Then follow the keyword 'for' and the name of the class the method belongs to.
EXPAND	The expand statement is used to reference other rules, which are inserted in that place.
IF	The if statement is used to branch generation rules. In an if statement is is possible to use attributes and methods of a visible class.
FOREACH	This statement is used to iterate through references that point to lists. An example for such a reference is the 'outgoingTransitions' reference of the class 'State'.

Table 3.1: The most important key words of the Expand language.



```
gmf-templates 1066 1066
  aspects 1066 1066
    impl 1066 1066
      diagram 1066 1066
        editparts 1066 1066
          CompartmentEditPart.xpt
          LinkEditPart.xpt 1066 106
          NodeEditPart.xpt 1066 10
    xpt 1066 1066
      editor 1066 1066
        DiagramEditorUtil.xpt 1066 1
```

Figure 3.18: The hierarchy of the ThinKCharts template folder.

the diagram editor generation model has to be enabled and the template path set to the folder the hierarchy of templates is contained in. In the case of the ThinKCharts editor it looks like shown in Figure 3.18.

The file 'NodeEditPart.xpt' is responsible for the generation of node edit parts. In this file only an 'this' had to be inserted into the constructor call of the method 'createNodeShape' to let the edit part pass itself to the created figure. Figure 3.19 shows the important part.

The same had to be done in the file 'LinkEditPart.xpt', so that connection edit parts pass themselves to the their corresponding connection figures. It is shown in Figure 3.20.

A few more changes were made to compartment edit parts. At first the compartment's borders are made invisible if they are not simple line borders by removing them. Then all but the state compartment is assigned a custom layout and finally if the compartment is not a region or a state compartment the compartment's title is added as a figure into the compartment itself to preserve the usual look of states. Figure 3.21 illustrates this. Note that the 'CompartmentEditPart.xpt' is only responsible for shape compartments, which are used in the ThinKCharts editor exclusively. It does not affect list compartments.

Finally, the file 'DiagramEditorUtil.xpt' is adjusted to make the editor automatically create a top level state if a new empty diagram is created. The method that is responsible for this is 'createInitialModel'. To accomplish this, a new state with a region inside and the name 'TOP LEVEL STATE' is added to the root region, which is then returned. Figure 3.22 depicts the modified method.

3.5.2 Layout

Two custom compartments had to be implemented in order to make SyncCharts in the ThinKCharts editor appear as used to. The first affects the layout of states.

```
«DEFINE createNodeShape(node : gmfgen::GenNode)
    FOR gmfgen::FigureViewmap-»
   «LET (if figureQualifiedClassName = null then ’org.eclipse.draw2d.RectangleFigure’
     else figureQualifiedClassName endif) AS fqn-»
      «EXPAND xpt::Common::generatedMemberComment»
      protected org.eclipse.draw2d.IFigure createNodeShape() {
         «REM» Added "this" «ENDREM»
         return primaryShape = new «fqn»(this)«IF node.getLayoutType()
             = gmfgen::ViewmapLayoutType::XY_LAYOUT»{
            protected boolean useLocalCoordinates() {
               return true;
            }
         }«ENDIF»;
      }

      «EXPAND getPrimaryShapeMethod FOR fqn-»
   «ENDLET-»
«ENDDEFINE»

«DEFINE createNodeShape(node : gmfgen::GenNode)
    FOR gmfgen::InnerClassViewmap-»
   «EXPAND xpt::Common::generatedMemberComment»
   protected org.eclipse.draw2d.IFigure createNodeShape() {
      «REM» Added "this" to let edit part pass itself to the figure «ENDREM»
      «className» figure = new «className»(this);
      «IF node.childNodes->size() > 0 and node.getLayoutType() = gmfgen::ViewmapLayoutType::XY_LAYOUT-»
      figure.setUseLocalCoordinates(true);
      «ENDIF-»
      return primaryShape = figure;
   }

   «EXPAND getPrimaryShapeMethod FOR className-»
«ENDDEFINE»

«DEFINE getPrimaryShapeMethod FOR String-»
   «EXPAND xpt::Common::generatedMemberComment»
   public «self» getPrimaryShape() {
      return («self») primaryShape;
   }
«ENDDEFINE»
```

Figure 3.19: The modified NodeEditPart.xpt.

```
«DEFINE createLinkFigure(link : gmfgen::GenLink) FOR gmfgen
    ::InnerClassViewmap»
  protected org.eclipse.draw2d.Connection createConnectionFigure() {
    «REM» Added "this" to let edit part pass itself to
        the figure <ENDREM»
    return new «className»(this);
  }

  «EXPAND xpt::Common::generatedMemberComment»
  public «className» getPrimaryShape() {
    return (<className») getFigure();
  }

  «classBody»
«ENDDEFINE»
```

Figure 3.20: The modified LinkEditPart.xpt.

It has been assigned to the state figure in the graphical definition model. The second modifies the layout of the text figures within the signal, variable, action, and suspension trigger compartments and also the region compartment. As stated above, it is assigned to the compartment through the modification of the compartment edit part template.

The custom State Layout The state layout does the following: In simple states the compartments are hidden, because they are all empty, and the name label is put in the center of the figure. If the state is a complex one, the name label is centerd on top. Then the required width and height is calculated. Since the elements are laid out in a single column, the required width is the width of the widest element and the required height is the sum of all heights. Empty compartments are not considered in this calculation. Also, the title labels of empty compartments are made invisible, because otherwise they would affect the state's minimum size. After those calculations are done, the non-empty compartments share the available space.

The custom Compartment Layout The compartment layout first sorts the children of the figure to layout in order to make wrapping labels appear first in the list. This ensures that if the compartment contains its title, the title appears to the left. Then the new bounds of every child are calculated. For invisible figures this is their minimum size, since this equals to the preferred size of the contained label. For all other figures it is their preferred size. The position of every child is calculated, so that they are laid out in rows and wrapped around if they reach the state's border.

```
«DEFINE createFigure FOR gmfgen::GenCompartment-»
   «IF not needsTitle-»
     «REM»By default titles are shown even if there are no TitleStyle, we need to switch it off«ENDREM»«-»
     «EXPAND xpt::Common::generatedMemberComment»
     public org.eclipse.draw2d.IFigure createFigure() {
        org.eclipse.gmf.runtime.diagram.ui.figures.ResizableCompartmentFigure
            result = (org.eclipse.gmf.runtime.diagram.ui.figures.ResizableCompartmentFigure)
           super.createFigure();
        result.setTitleVisibility(false);

        «REM» Make compartment borders invisible «ENDREM»
        Object border = result.getBorder();
        if (border instanceof org.eclipse.draw2d.LineBorder) {
           org.eclipse.draw2d.LineBorder lineBorder = ((org.eclipse.draw2d.LineBorder) border);
           lineBorder.setWidth(2);
           lineBorder.setColor(org.eclipse.draw2d.ColorConstants.black);
        }
        result.setBorder(null);
        «REM» Give all but state compartments a custom layout «ENDREM»
        «IF title != "StateCompartment"-»
           «IF title != "RegionCompartment"-»
           result.getContentPane().setLayoutManager(new de.cau.cs.kieler.synccharts.custom
               .CustomCompartmentLayout());
           «REM» Add the compartment’s title as a new figure within the compartment «ENDREM»
              org.eclipse.gmf.runtime.draw2d.ui.figures.WrappingLabel
                  title = new org.eclipse.gmf.runtime.draw2d.ui.figures.WrappingLabel();
              title.setText(result.getCompartmentTitle() + " ");
              title.setForegroundColor(org.eclipse.draw2d.ColorConstants.black);
              result.getContentPane().add(title, 0);
           «ENDIF-»
        «ENDIF-»
        return result;
     }
   «ENDIF-»
«ENDDEFINE»
```

Figure 3.21: The modified CompartmentEditPart.xpt.

```
«DEFINE createInitialModelMethod FOR gmfgen::GenDiagram>»
   «EXPAND xpt::Common::generatedMemberComment(’Create a new instance of domain element associated with canvas.\n
       <!-- begin-user-doc -->\n<!-- end-user-doc -->’)»
   private static <EXPAND MetaModel::QualifiedClassName FOR
       domainDiagramElement> createInitialModel() {
     «REM» Add a root region with a state and another region to the diagram «ENDREM»
     de.cau.cs.kieler.synccharts.Region root = de.cau.cs.kieler.synccharts.SyncchartsFactory
         .eINSTANCE.createRegion();
     de.cau.cs.kieler.synccharts.State state = de.cau.cs.kieler.synccharts.SyncchartsFactory
         .eINSTANCE.createState();
     de.cau.cs.kieler.synccharts.Region region = de.cau.cs.kieler.synccharts.SyncchartsFactory
         .eINSTANCE.createRegion();
     state.setLabel("SyncChart");
     state.getRegions().add(region);
     root.getInnerStates().add(state);
     return root
   }
«ENDDEFINE»
```

Figure 3.22: The modified DiagramEditorUtil.xpt.

```
<extension point="org.eclipse.ui.commands">
    <category name="SyncCharts" description="Commands related
        to SyncChart diagrams."
        id="org.eclipse.gmf.category.synccharts"/>
    <command
        categoryId="org.eclipse.gmf.category.synccharts"
        defaultHandler="de.cau.cs.kieler.synccharts.diagram.custom
      .handlers.AddToStateHandler"
        description="Adds a new OnEntryAction"
        id="de.cau.cs.kieler.synccharts.custom.contextMenu
      .addEntryAction"
        name="Add OnEntryAction">
    </command>
  </extension>
```

Figure 3.23: A custom command.

3.5.3 Context Menu

The contributions to the context menu are contained in the project 'de.cau.cs.kieler.synccharts.custom.contextmenu'. First, the project has to make known that it extends the menu. This is done in the plugin.xml by registering commands. These commands are given an ID, the ID of the editor they are supposed to appear in, and a class which is responsible for executing the command. Figure 3.23 shows a sample command. Then the menu contributions are listed. The object contribution determines which type of element has to be selected for the new menu entry to appear. The element is defined by its edit part class. Then new submenus and seperators can be created and also new action which are put into a specified sumenu or group by simply specifying the path. The action also contains the name of the executing class. Figure 3.24 depicts a sample contribution.

The classes, which add new elements to the selected one are all structured in the same way. They feature a 'selectionChanged' method that updates the current selection in case it has been changed and a 'run' method that is responsible for adding the new element. First, it searches the selected element for the appropriate compartment to put the element in. Then a new command is created that adds the new element to the compartment. This compartment is put on the command stack and executed. Then the new element is put into edit mode so the user can modify it.

```
<objectContribution adaptable="false" id="de.cau.cs.kieler
    .syncharts.diagram.ui.objectContribution.RegionEditPart"
    objectClass="de.cau.cs.kieler.syncharts.diagram.edit
   .parts.RegionEditPart">
   <menu id="ElementInsert" label="Insert Element"
       path="additions">
      <separator name="group1"/>
   </menu>
   <action class="de.cau.cs.kieler.syncharts.custom
       .contextMenu.AddStateAction"
       definitionId="de.cau.cs.kieler.syncharts.custom
       .contextMenu.addState" enablesRor="1"
       id="de.cau.cs.kieler.syncharts.custom.contextMenu
       .AddStateActionID" label="Add State"
       menubarPath="ElementInsert/group1">
   </action>
</objectContribution>
```

Figure 3.24: A custom object contribution.

4 Conclusion

This section concludes the discussion with an evaluation of this work in Section 4.1, a summary in Section 4.2, and finally a look on future work in Section 4.3.

4.1 Evaluation

During the development of the ThinKCharts editor, Eclipse and the GMF framework have proven to be a rich, versatile platform for creating diagram editors. Many low level functionalities are automatically taken care of, auch as the creation of windows, notification of events, or graphical feedback. Also, some very convenient features come for free with every GMF editor. These include popup buttons, context menus, a properties view, and many others. In addition, the generated editors can be modified to a great amount, starting by simply changing some figures in the graphical definition model to integrating whole new views or modifying the editors behaviour by changing the underlying code, everything is possible.

However, those things are not always easy to achieve. GMF is very complex and provides only little literature to rely on. Much time has to be spent on experimenting with the features and debugging GMF and Eclipse itself, if an editor is to be heavily customized.

4.2 Summary

In this paper, the Eclipse platform along with its subprojects EMF, GEF, and GMF have been introduced; together with openArchitectureWare's XText project. It was shown how to put these projects together to create a SyncCharts editor; first the concept was discussed with the main problems attribute awareness, parsing, and validation and some smaller challenges, then the code generation process and implementation were explained in detail. Finally, the project was evaluated and the advantages and disadvantages of GMF were discussed.

4.3 Future Work

Although the ThinKCharts editor already provides a user friendly environment to graphically create syncCharts, there are some things that can be improved. The input method for signals and variables and the like is a little clumsy, since they have to be added one by one, constantly switching between the context menu and the label edit mode. Also, the layout of regions could be improved to fill all the space available in the parent state. Furthermore, the transition anchors dock onto the bounding boxes of figures, leaving a little gap when being attached to elliptic figures. Additionally, the top level state that is created automatically in empty diagrams is not adjusted to the canvas' size.

5 Bibliography

[1] C. André. Semantics of SyncCharts. Technical report, Laboratoire I3S - Sophia Antipolis, 2003.

[2] J. Bézivin, C. Brunette, R. Chevrel, F. Jouault, and I. Kurtev. Bridging the generic modeling environment (GME) and the eclipse modeling framework (EMF). In *Best Practices for Model Driven Software Development*, 2005.

[3] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[4] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[5] D. Harel. Statecharts: A visual approach to complex systems. In *Science of Computer Programming*, volume 8, pages 231–274, 1987.

[6] Á. Lédeczi, M. Maróti, and P. Völgyesi. The generic modeling environment. Technical report, Institute for Software Integrated Systems, Vanderbilt University Nashville, 2001.

[7] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Tech. J.*, 34:1045–1079, Sept. 1955.

[8] A. Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Dec. 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`.

[9] M. Spönemann, H. Fuhrmann, and R. von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0914.pdf`.

[10] J. White, D. C. Schmidt, and S. Mulligan. The Generic Eclipse Modeling System. In *Model-Driven Development Tool Implementors Forum*, 2007.