Christian-Albrechts-Universität zu Kiel

Diploma Thesis

A Generic Framework for Structure-Based Editing of Graphical Models in Eclipse

cand.inform. Michael Matzen

March 26, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Hauke Fuhrmann

Eidesstattliche Erklärung

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

Kiel,

Abstract

This thesis introduces the paradigm of structure-based editing to the Eclipse modeling platform to improve the efficency of graphical modeling.

The approach is based on the fundamentals of model based design, i.e. metamodels are used to define the structure of all their derived model instances. This architecture allows the definition of operations on the metamodel level and their execution on arbitrary model instances.

To actually perform an editing operation we will propose a transformational approach, which is based on the well-known technique of model transformation. Furthermore, we will evaluate some of the most common model transformation frameworks for their usability with a structure-based editing project.

Additionally, a reference implementation of the transformational approach to structure-based editing will be presented.

The implementation has been embedded into the Eclipse framework, in the context of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), which combines a range of innovative approaches to improve the overall process of editing graphical models.

Keywords: Eclipse, KIELER, model-based design, structure-based editing, model transformation

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
ATL	ATLAS Transformation Language
CASE	Computer-aided software engineering
DND	drag-and-drop
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JDT	Java Development Tooling
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIML	KIELER Infrastructure for Meta Layout
KITE	KIELER Textual Editing Framework
KITS	KIELER Textual SyncCharts
KLoDD	KIELER Layout of Dataflow Diagrams
KoData	KIELER of Dataflow

KSBasE	KIELER Structure-Based Editing
M2M	Eclipse Model To Model Transformation
MDA	Model Driven Architecture
MOF	Meta Object Facility
MVC	Model-View-Controller
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformation
QVTo	Operational QVT
QVTd	QVT Declarative
PDE	Plug-In Developer Environment
RCA	Rich Client Application
RCP	Rich Client Platform
ThinKCharts	The Thin KIELER SyncCharts Editor
UI	User Interface
UML	Unified Modeling Language
W3C	World Wide Web Consortium
WYSIWYG	What-You-See-Is-What-You-Get
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Chapter 1

Introduction

The White Rabbit put on his spectacles. “Where shall I begin, please your Majesty?” he asked.
“Begin at the beginning,” the King said gravely, “and go on till you come to the end: then stop.”

from: “Alice’s Adventures in Wonderland”

Since model based design emerged to a widely-used technique for creating software systems, the complexity and size of the models constantly grew and raised the need for sophisticated modeling tools and paradigms. Most of the state-of-the-practice editing tools provided by common editors, such as drag-and-drop (DND) editing, however, are well-suited for small and simple diagrams, but fail to enable fast and efficient editing of larger models, e.g. as the one shown in Figure 1.5.

The primary drawback in common editing techniques is that the user is working on a graphical representation, which is, in general, not as compact as a textual representation and therefore introduces additional effort for simple operations. Inserting new elements, for example, involves a large amount of, mostly unnecessary, mouse gestures for creating the element, placing it in the diagram area and adjusting the layout. Furthermore, operations like replacing a diagram element with a different one involves an entire sequence of editing steps, e. g. deletion of the existing element and creating a new one with the same properties and relations.

The aim of this thesis is to introduce the paradigm of structure-based editing into the widely-used Eclipse modeling platform to enhance an entire set of graphical editors by providing operations that are based on the underlying structure of the editor. The approach proposed here is based on the fundamentals of model based design, i. e. abstract models are used to define the structure of all their derived model instances, which may have a variety of characteristics. A model might, for example, describe a graphical or textual editor. Furthermore, the common technique of model transformation will be used as basic concept for defining and executing structure-based editing operations.

The first main chapter of this thesis will give a general introduction to model based design and an evaluation of existing model transformation frameworks. The second chapter introduces the technologies that have been used in this thesis, including an

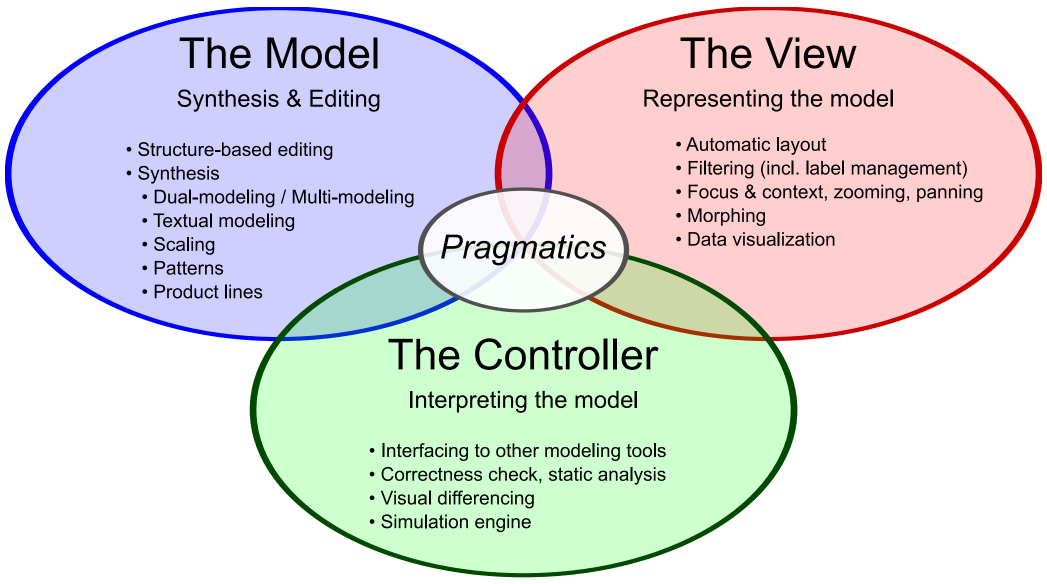


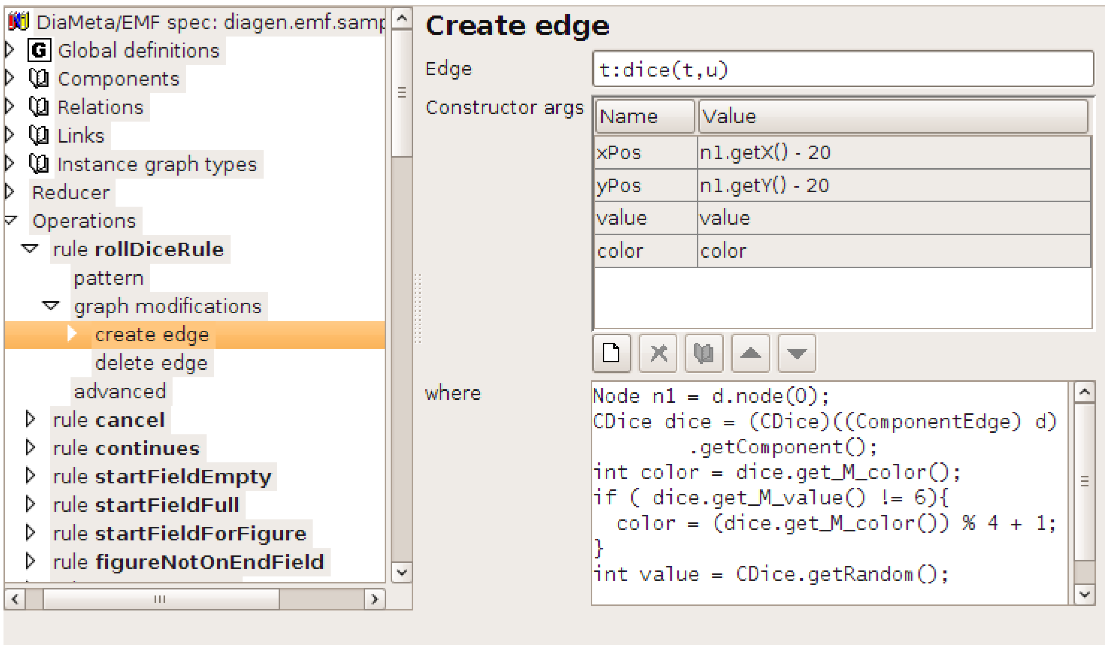
Figure 1.1.: An adaption of the MVC pattern for graphical models [13]

overview of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. In chapter 4 we will discuss an approach to structure-based editing, which is based on the technique of model transformation, followed by a reference implementation of the given approach in the context of the KIELER project. This chapter will also give example configurations for two editors, shipped with KIELER, as well as an evaluation of the improvements in editing graphical models with structure-based editing compared to other editing mechanisms. The thesis will close with a summary of the given approach and an outlook to future research topics. In the appendix we will give an additional user and developers guide for the KIELER structure-based editing framework.

1.1. Related Work

The primary reference for this thesis is the work of Fuhrmann and von Hanxleden [13], where the authors introduce pragmatics for model-based desing and modeling languages. The primary objective of these pragmatics is the enhancement of processes related to creating, editing and viewing graphical models. The authors adapted the common Model-View-Controller (MVC) pattern, illustrated in Figure 1.1, to create a sophisticated separation of concerns between the different aspects of model-based design. As a major part of how the editing of models can be improved, the authors proposed the paradigm of structure-based editing by performing in-place model transformations.

Structure-based editing, also known as structural- or syntax-directed editing, is common in textual editors and Integrated Development Environments (IDEs), where



Create edge

Edge: `t:dice(t,u)`

Constructor args

Name	Value
xPos	n1.getX() - 20
yPos	n1.getY() - 20
value	value
color	color

where

```
Node n1 = d.node(0);
CDice dice = (CDice)((ComponentEdge) d)
        .getComponent();
int color = dice.get_M_color();
if ( dice.get_M_value() != 6){
  color = (dice.get_M_color()) % 4 + 1;
}
int value = CDice.getRandom();
```

Figure 1.2.: DiaMeta Ludo-Example Rule Definition

the underlying language semantics are used to aid developers during the editing processes, e. g. by providing syntax-highlighting or context sensitive operations, such as the Eclipse template mechanism. Beside the use in textual editors, structure-based editing has also been introduced to graphical editing frameworks by several projects.

The diagram generator project DiaGen [20], developed at the University of Erlangen-Nürnberg and its successor DiaMeta [21], developed at the Universität der Bundeswehr München, provide tools to create standalone graphical editors either from a formal, hypergraph based grammar (DiaGen) or an abstract model (DiaMeta). Furthermore, both projects allow to define operations for the editor by creating graph transformations, as shown in Figure 1.2. These transformations are integrated into the editor, as shown in Figure 1.3, and may be executed on the current model. The integration of those transformations, however, is only rudimentary, e. g. if the diagram contains more than one object of a given transformation type, the target element is not chosen by the user but by the order of the elements in the diagram. The major drawbacks of this approach are the complex syntax of graph transformations, the missing flexibility when selecting a transformation target and that the editing operations are generated with the editor, so adding a new operation involves regenerating the entire editor. A project that uses a very similar approach is Generation of Graphical Environments for Design (GenGED) [6]. GenGED also provides a set of tools to create editors for arbitrary visual languages based on an alphabet and graph grammar rules. In contrast to the DiaGen and DiaMeta projects, GenGED allows

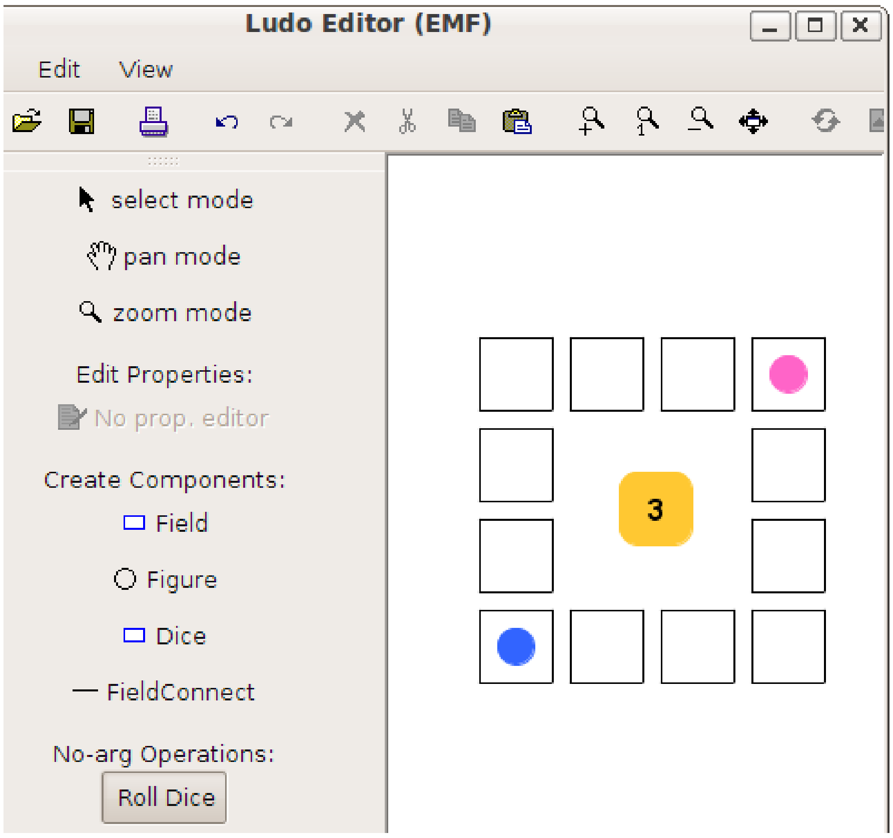


Figure 1.3.: DiaMeta Ludo-Example Application

to create model transformations graphically by defining preconditions, the left, and right hand side of the transformation, as shown in Figure 1.4.

Another project that focuses on the execution of in-place model transformations in graphical editors is the TIGER EMF Transformation Project (EMFTrans) [10]. The project comprises a custom transformation language as well as an Eclipse-based graphical editor for creating arbitrary in-place transformations on models that are built on top of a model, which is provided by the Eclipse Modeling Framework (EMF). Those model transformations may also be compiled into Java code and integrated into custom projects. The EMFTrans project, however, focuses on creating model transformations, whereas the structure-based editing framework introduced in this thesis aims at a generic framework for enhancing the actual modeling process by defining in-place transformations and integrating them transparently into editors that are based on the Graphical Modeling Framework (GMF). Since EMFTrans has not been integrated into the latest Eclipse version, it has not been considered as a possible transformation framework for the structure-based editing project, proposed later on. However, EMFTrans has lately been incorporated into the Eclipse Henshin[1]

[1] http://www.eclipse.org/modeling/emft/henshin/

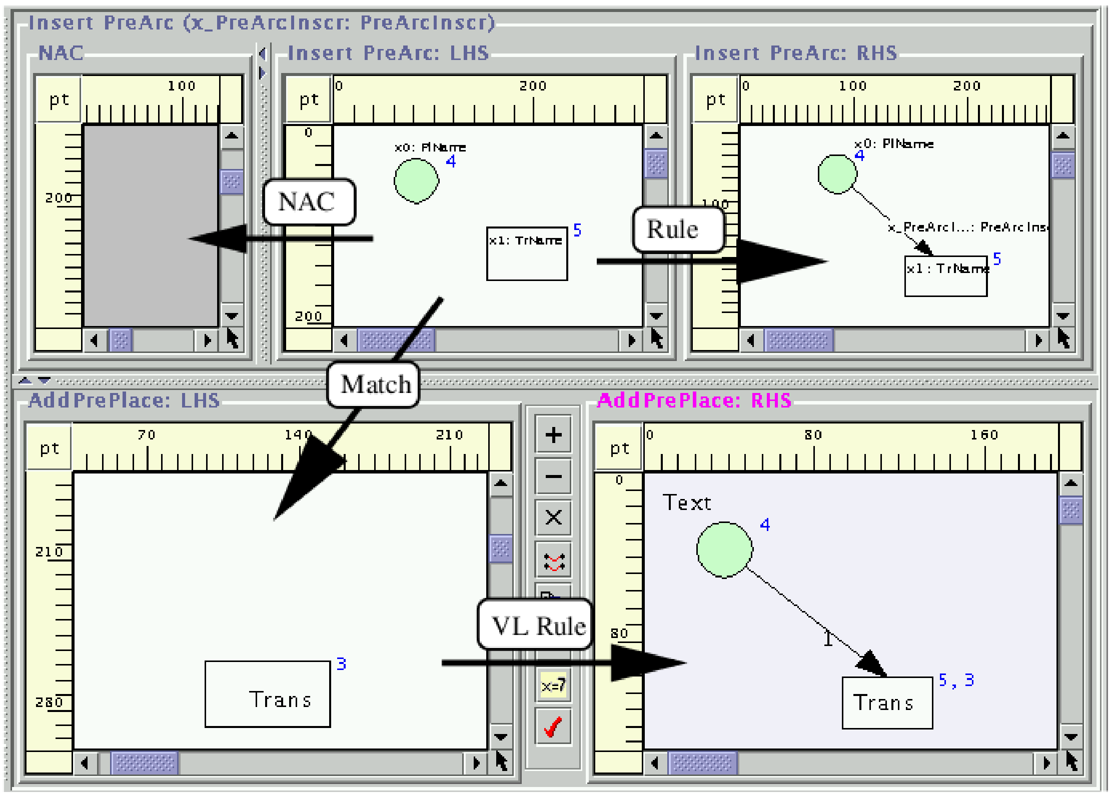


Figure 1.4.: GenGED grammar editor [1]

project, so it may be considered in future versions.

A work that is closely related to this thesis is the diploma thesis of Mirco Wischer [32], where a structure-based approach has been used to improve the Kiel Integrated Environment for Layout (KIEL) framework, which is the predecessor of the KIELER project in which this thesis has been created. Although the initial approach is very similar to the one taken in this work, there are major differences, as the KIEL framework is a monolithic tool for creating Statecharts whereas this thesis proposes a more generic way of creating structure-based features for an entire set of graphical editors.

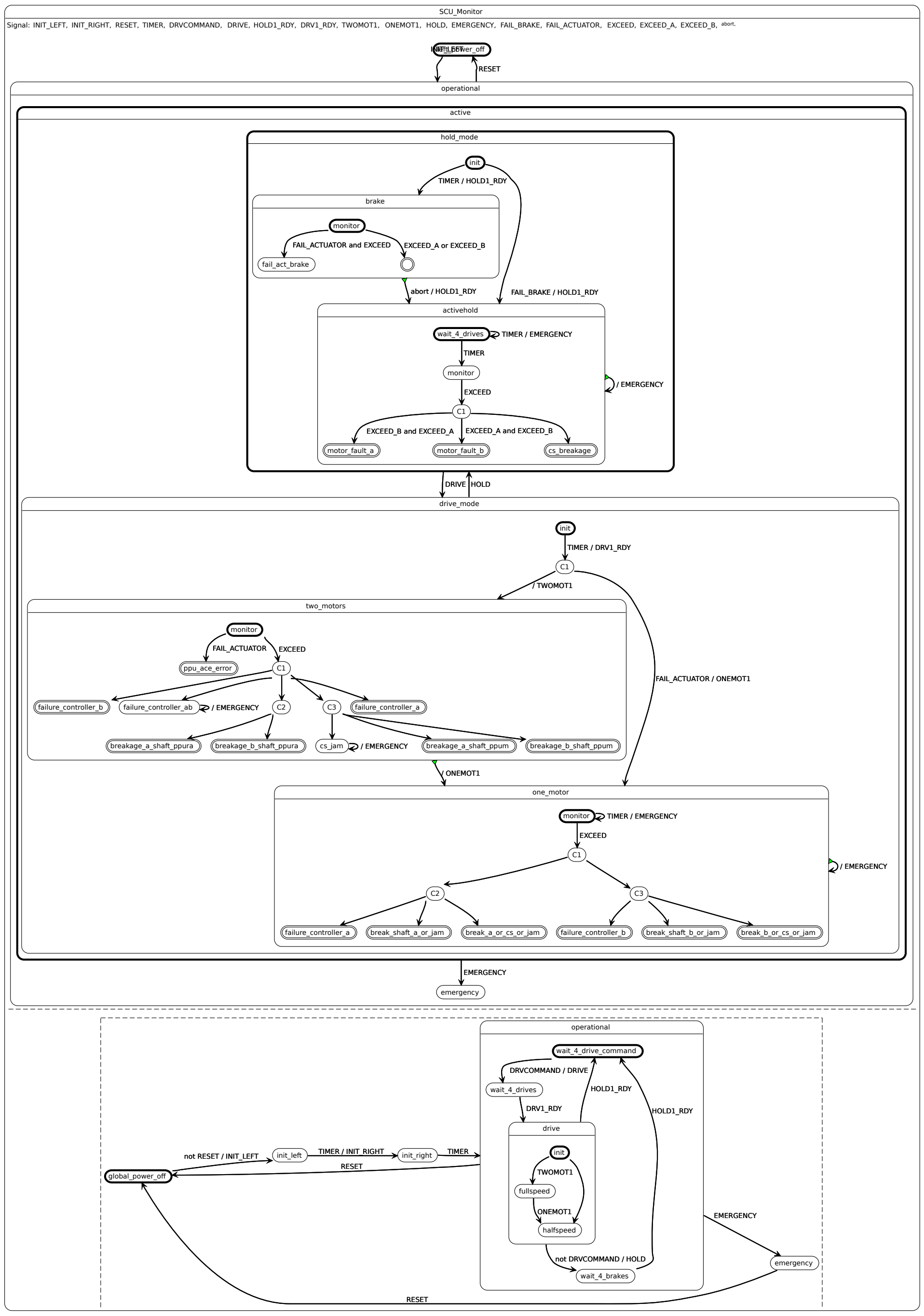


Figure 1.5.: More complex SyncChart [12]

Chapter 2

On Model-Based Design

This chapter gives an overview of the fundamentals of model-based design and the technique of model transformation, including examples for the different types of transformations. In Chapter 2.2 we will have a look at common model transformation frameworks and evaluate their use for a structure-based editing project.

The overall architecture of a model space, called the *technical space* [16], is divided into four layers of abstraction: the meta-metamodel, the meta- or semantic model, the model, and an object of a model instance. Additionally, technical spaces contain concepts, tools, techniques, and formalisms associated with a particular technology.

The most abstract layer, called M3-Level, defines the structure of all metamodels that are declared in a technical space by using a model, called the *meta-metamodel.* The M2-Level contains models that are defined by creating instances of the M3 meta-metamodel. These models are called *metamodels* because they are used to describe the actual model instances that can be defined in this technical space. All these model instances are contained in the M1-Level. The last layer, M0, comprises objects which are instances of model elements from the M1-Level. It may be very hard to describe the contents of the M0-Level for some technical spaces, e. g. for the Extensible Markup Language (XML), in these cases the M0-Level may be undefined.

Figure 2.1 shows two examples of common technical spaces, the Model Driven Architecture (MDA) [4] technical space, defined by the Object Management Group (OMG)[1] and the XML space, defined by the World Wide Web Consortium (W3C)[2]. The MDA space is based upon the Meta Object Facility (MOF) meta-metamodel and contains languages such as the Unified Modeling Language (UML). The XML space uses Extended Backus-Naur Form (EBNF) [14] as the meta-metamodel and contains languages such as XML, Extensible Stylesheet Language Transformations (XSLT)[3], and the HyperText Markup Language (HTML).

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

[2] http://www.w3.org/

[3] http://www.w3.org/TR/xslt

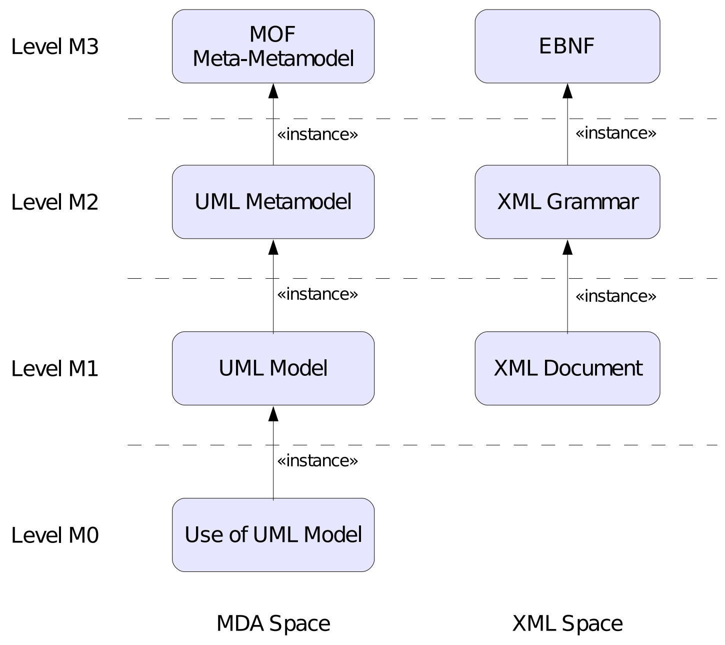


Figure 2.1.: Technical spaces for MDA and XML [16]

2.1. Model Transformations

One of the most important aspects of model-driven development is the technique of model transformation. It describes the process of converting models based on a set of transformation rules. Furthermore, the transformation of models is not restricted to metamodels or technical spaces, i. e. it is possible to convert models between arbitrary spaces; however, the transformation between different technical spaces may be difficult and require additional pre- and post-processing steps for creating a representation of the source models in the target space. For example, the transformation of an XML document into the UML space requires the definition of a MOF-based metamodel for XML grammar documents.

However, since we will not use the transformation between different technical spaces in the context of this thesis, we will assume from now on all models reside in the same technical space.

With this restriction, the term model transformation is simplified to conversion of models from source to target metamodels. Because these metamodels do not necessarily need to be different, two classes of transformations have been defined by Mens and van Gorp [19]: *endogenous* and *exogenous* transformations.

A transformation is called endogenous if source and target M1-models are based on the same M2-metamodel, whereas an exogenous transformation implies that they are different.

Additionally, if the source and target M1-models of an endogenous transformation are identical, the transformation is called *in-place*, i. e. the transformation will not create a new model, but the given input model will be modified based on the transformation rules. This technique is often used for maintenance of models, e. g. if a model needs to be optimized.

2.1.1. Example

To clarify the different aspects of model transformation we will have a look at an example with two small metamodels (shown in Figure 2.2) and perform exogenous and endogenous in-place model transformations.

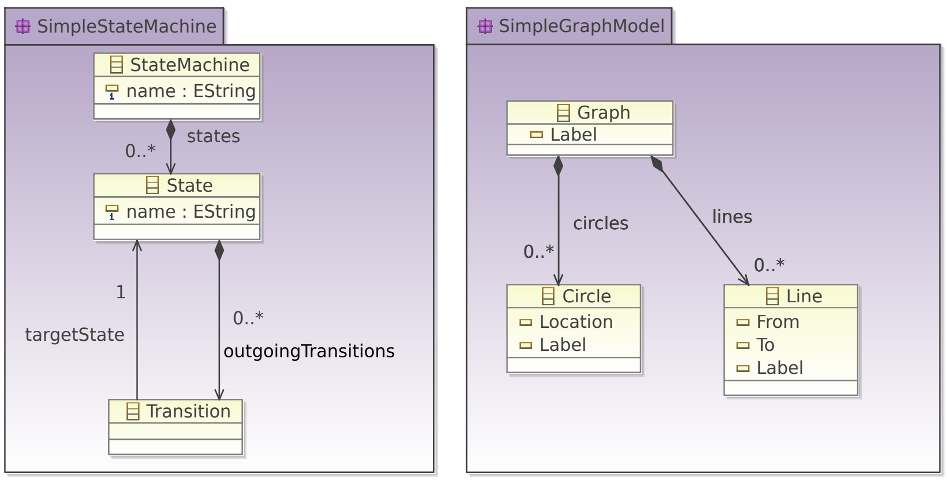


Figure 2.2.: Example Metamodels

Exogenous Transformation

First, we are going to perform an exogenous transformation on a model instance of the SimpleStateMachine metamodel (shown in Figure 2.3) to create a new instance of the SimpleGraphModel.

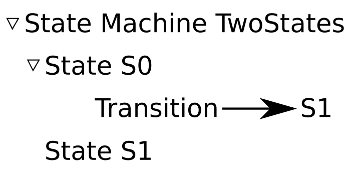


Figure 2.3.: SimpleStateMachine model

To transform the given model into the simple graph metamodel, we are applying the rules shown in Listing 2.1.

Listing 2.1: Exogenous transformation (pseudocode)

```
exogenousTransformation(input SimpleStateMachine t) : output SimpleGraphModel
   SimpleGraphModel g = new SimpleGraphModel;
   g.label = t.name;
   foreach State s in t
   {
      Circle c = new Circle;
      c.label = s.name;
      g.addCircle(c);
   }
   foreach Transition r in t
   {
      Line l = new Line;
      l.label = r.sourceState.name + r.targetState.name;
      l.source = g.getCircle(r.sourceState.name);
      l.target = g.getCircle(r.targetState.name);
   }
   return g;
```

After applying those rules to our example model, we will get a new model instance of the SimpleGraphModel and we may be able to visualize the resulting model as shown in Figure 2.4.

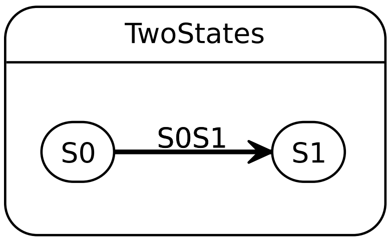


Figure 2.4.: Model instance of the SimpleGraphModel

Endogenous Transformation

In the next example, we will perform an endogenous in-place transformation, i. e. the metamodel will not be changed and the model output is equal to the input. This time we will use the model shown in Figure 2.4 and apply the rules shown in Listing 2.2.

After the transformation has been executed, the given model will contain four states and may be visualized as shown in Figure 2.5.

Listing 2.2: Endogenous transformation (pseudocode)

```
endogenousTransformation(input SimpleGraphModel g)
foreach Circle c in g
{
   Circle c2 = new Circle;
   c2.label = g.circles.count + 1;
   Line l2 = new Line;
   line.source = c;
   line.target = c2;
   line.label = c.label + c2.label;
}
```

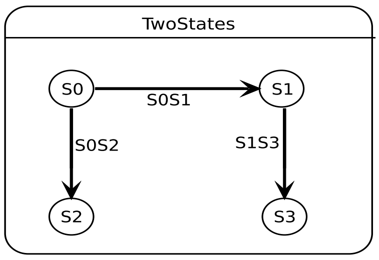


Figure 2.5.: The SimpleGraphModel after the execution of an endogenous in-place transformation

2.2. Evaluation of Existing Model Transformation Frameworks

Since model-driven development and model transformation have been introduced, several different frameworks for defining and executing model transformations have been developed. In this section we will have a look at some of the currently existing frameworks, which have been evaluated during the planning phase of this thesis. To visualize the differences between those frameworks, each section will contain a net diagram with the following criteria, which have been created to measure the usability of the frameworks for a structure-based editing project.

1. Declarative Rules

   Does the language support the use of declarative rules?

2. Imperative Rules

   Does the language support the use of imperative rules?

3. Simplicity

   The total number of keywords.

4. Eclipse Tool Support

   Does the framework comprise Eclipse features for creating, executing and debugging transformations ?

5. Error Handling

   Is there an appropriate console error message when a transformation failed, and is it possible to catch this message when executing a transformation programmatically?

6. Dynamic Execution

   Is it possible to address a specific target element for a transformation?

7. External Code Invocation

   Is it possible to call external methods from inside a transformation?

The programming paradigm, either declarative or imperative, is a more informal categorization and does not have any immediate influence on the evaluation of the framework in this thesis. Nevertheless, developers who are not familiar with declarative/imperative programming might feel more comfortable when using a framework that allows the use of imperative/declarative rules.

The Dynamic Execution criterion summarizes the processes that are related to the execution of transformations only for specific model elements. This requirement has been raised, because most transformation frameworks are built on the assumption that an entire model will be transformed. The transformational approach to structure-based editing, proposed later on, however, assumes that it is possible to execute a transformation for a specific model element. If a framework lacks the option of performing a transformation on a single element, the execution of an in-place transformation, e. g. to insert a successor element for a selected diagram object, may result in an unintended behavior, e. g. successor elements for all diagram objects. Those transformation frameworks will need an additional mechanism to address the elements that are currently selected in the graphical editor and perform the transformation on those elements. Therefore, the dynamic execution is an important performance and convenience requirement for a structure-based editing framework.

Because this project mainly focuses on the enhancement of GMF-based editors, the Eclipse tool support is another very important requirement. It comprises features such as the programmatic execution of transformations, debugging support, and an editor with the state-of-the-art features, e. g. syntax highlighting and code completion.

The invocation of external code, e. g. Java methods, may be very useful when working with graphical editors to retrieve more detailed information about the current state of a diagram model.

The total number of keywords and the error handling capabilities of QVT and ATL have been adopted from the work of Mitoussis and Macos [23].

2.2.1. QVT

With the aim to create an industrial standard for model transformation languages, the OMG published a request for a language proposal in 2002 [3]. This request comprises the proposal of frameworks, which allow to create queries, views, and transformations for models based on the MOF 2.0 standard.

During the next three years, the OMG received proposals from several different research institutes as well as from industrial OMG members. In 2008 the OMG finally released a first specification, which formalized and standardized the results from the given proposals and defined the Query/View/Transformation (QVT) standard [2].

The QVT standard consists of two different natures, a declarative and an imperative part (see Figure 2.6).

The declarative part itself is split into two layers, a relational and a core layer. The relational layer contains declarative rules to define model relations, which can be used to define complex pattern matches and templates. The core language is as powerful as the relations language, but is built with a smaller set of rules and therefore has simpler semantics. The QVT specification explains the difference between the relational and the core language by an analogy to the Java framework:

> "An analogy can be drawn with the Java^{TM}architecture, where the Core language is like Java Byte Code and the Core semantics is like the behavior specification for the Java Virtual Machine. The Relations language plays the role of the Java language, and the standard transformation from Relations to Core is like the specification of a Java Compiler,which produces Byte Code." [2]

In addition to the declarative languages, the QVT standard contains an operational mappings part. These operational mappings allow the definition of rules with an imperative syntax, which may be more common for developers who are not familiar with declarative programming.

To create a very flexible standard, the framework contains interfaces between the different layers. As visualized in Figure 2.6, the operational mappings can be interconnected either with the relational or the core layer. Additionally, it is possible to transform the relational mappings into the core language, as they are equally powerful.

As a last layer the standard contains a so-called *Black Box*, which may describe an arbitrary external module with a MOF binding. This black box enables the developer to access any type of domain-specific data, which may be impossible to express with the QVT semantics. A simple example of a black box is a Java class with a set of public methods, which can be called from inside a QVT transformation.

After the publication of this standard, several transformation frameworks, which claim to be fully compatible with QVT, have been released. Some examples are the ATLAS transformation language [15], the visual automated model transformation system (VIATRA) [30] or the Eclipse QVT Language projects (QVTo/QVTd). Furthermore, there are some frameworks that have been developed independently from

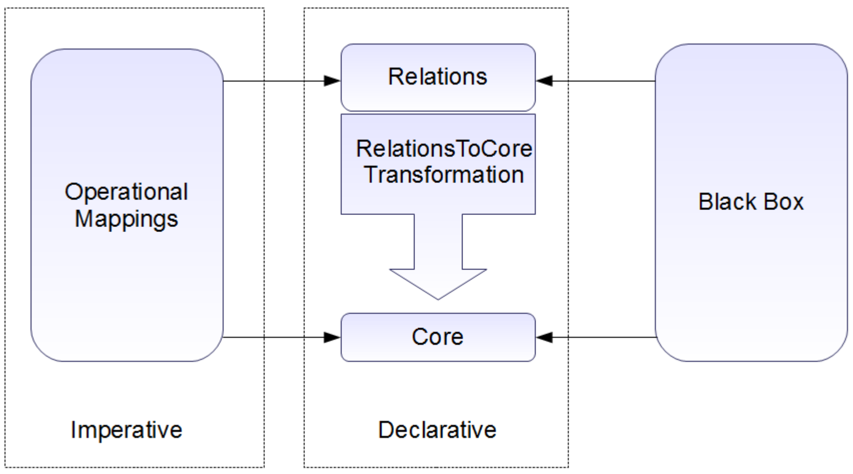


Figure 2.6.: Overview of QVT layers [2]

the given standard, for example the Xtend[4] transformation language.

2.2.2. QVTo/QVTd

The Operational QVT (QVTo) and the QVT Declarative (QVTd) projects have been designed for the Eclipse Model To Model Transformation (M2M) project[5] to provide model transformation frameworks that are implemented directly by following the QVT standard.

As defined by the OMG, the M2M project distinguishes between declarative and operational mappings. Therefore the QVTd project contains sub-projects for the relational and the core layer.

The QVTd project is at the present time still in an early development and design phase and therefore could not be evaluated. The QVTo project, however, is already fully integrated into the Eclipse modeling process and is used to generated Java code from models. Listing 2.3 shows an example in-place transformation, which adds an attribute to a given EClass object. In the current version of the QVTo project, the execution of transformations from Java code is fully implemented, but restricted to a single entry point per file (see Listing 2.3 lines 13-15); nevertheless, it is possible to execute those transformations on single model elements, so the QVTo language may possibly be used for a structure-based editing framework.

[4]http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html
[5]http://wiki.eclipse.org/M2M

Listing 2.3: Example in-place transformation for QVTO

```
modeltype Ecore uses 'http://www.eclipse.org/emf/2002/Ecore';

transformation addAttribute(inout ecore : Ecore);

mapping inout EClass::addAttribute() {
   eAttributes += createAttribute()
}

mapping createAttribute() : EAttribute {
   name := "New Attribute"
}

main() {
   ecore.rootObjects()[EClass]->map addAttribute();
}
```

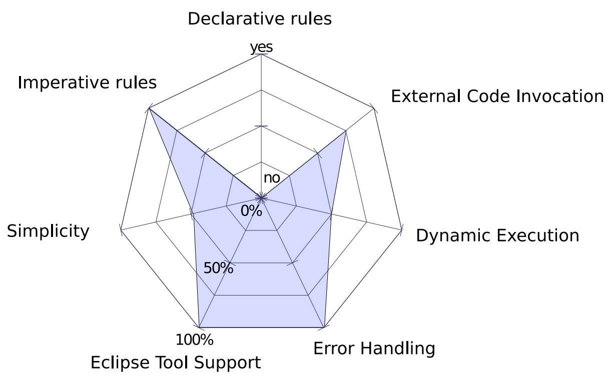


Figure 2.7.: QVTo classification

2.2.3. ATL

The ATLAS Transformation Language (ATL) has been developed by the ATLAS INRIA & LINA research group from the university of Nante in 2006 [15]. The language has been created as a part of the ATLAS Model Management Architecture (AMMA) and has been integrated into the Eclipse M2M project in 2008. The authors denote the language as 'a QVT-like transformation language' because most of the given QVT scenarios and requirements have been satisfied. ATL is capable of creating queries, views and transformations on models and compatible to the MOF, XML Metadata Interchange (XMI) and Object Constraint Language (OCL) standards. The language is a hybrid mixture of declarative and imperative programming paradigms, where the authors encourage the use of declarative constructs whenever it is possible and only use imperative parts when necessary.

As the ATL framework is based on the Eclipse platform, it contains a complete

Listing 2.4: Example in-place transformation for ATL

```
-- @atlcompiler atl2006
-- @nsURI Ecore=uri:http://www.eclipse.org/emf/2002/Ecore
module Extensions;
create OUT : Ecore refining IN : Ecore;

rule AddAttribute  {
   from class : Ecore!EClass
   to
      class2 : Ecore!EClass (
         eAttributes <- attrib
      ),
      attrib : Ecore!EAttribute (
         name <- 'New Name'
      )
}
```

set of editors and uses the existing technologies for executing and debugging transformations. The transformation programs are compiled by the ATL engine, into byte-code, which is afterwards executed by an ATL virtual machine. The major drawbacks, when using ATL for a structure-based editing framework, are the missing support for external code invocation and the transformation of single diagram elements. Combined with the comparatively complex syntax, ATL turned out to be an inappropriate framework for a structure-based editing project.

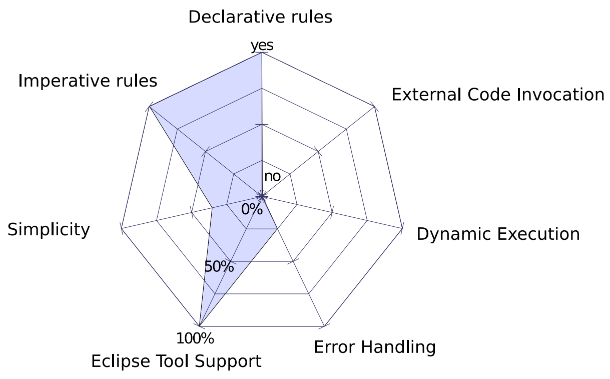


Figure 2.8.: ATL classification

2.2.4. Xtend

The Xtend transformation language has been developed as part of the Xpand project by the b+m group[6]. As stated before, the Xtend language has not been built upon

[6]http://www.bmiag.de

the QVT language standard; nevertheless it supports all models that are based on the MOF and several other common technical spaces, e. g. XML. Furthermore, the common expression language, defined especially for the Xpand project, allows the creation of model transformation with only a few lines of code, as shown in Listing 2.5. During the integration of the former openArchitectureWare framework into the Eclipse Modeling Project, the Xtend language has been tightly integrated into the Eclipse framework. It contains a full-featured editor as well as interfaces for executing transformations on arbitrary model elements and calling Java code from transformations, but is missing an integrated debugging feature. Nevertheless, since Xtend provides the best matches in the most important evaluation categories, it has been used for the reference implementation. Therefore, Chapter 3.3 contains a more detailed description of Xtend.

Listing 2.5: Example in-place transformation for Xtend

```
import ecore;

Void addAttribute(EClass c):
let attribute = new EAttribute:
 attribute.setName('New Name') ->
 c.eAttributes.add(attribute)
;
```

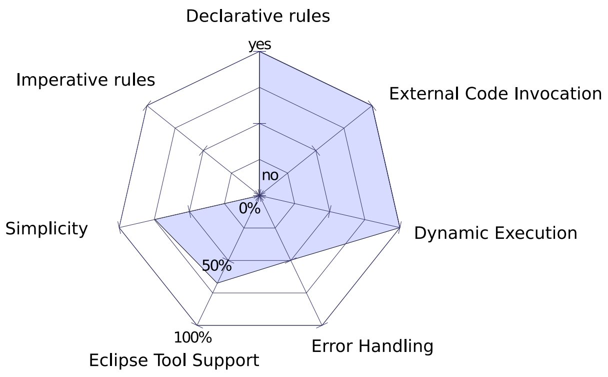


Figure 2.9.: Xtend classification

Chapter 3

Used Technologies

"It's dangerous to go alone! Take this."

from: "The Legend of Zelda"

This chapter will give an introduction to the technologies used in this thesis and the reference implementation. For most of these technologies there will be only a short overview, since a detailed in-depth description would go beyond the scope of this thesis.

3.1. The Eclipse Platform

The most obvious technology used in the context of this work is the Eclipse platform. Eclipse is a modular, open source development environment, which has been initially developed by IBM as an extensible IDE for Java and has evolved to an environment for a wide range of programming languages, e. g. C/C++, PHP and Ruby, including the development and deployment of applications for real-time and embedded systems, e. g. Google Android[1].

While the first Eclipse platforms have been built upon a custom plug-in architecture, the Eclipse foundation members decided in 2004 to use the open service gateway specifications, defined by the OSGi Alliance, as the basic platform concept. Figure 3.1 shows the platform runtime with two of the most important plug-ins: the Java Development Tooling (JDT) and the Plug-In Developer Environment (PDE). The JDT provides all components that are necessary to create and execute Java applications. Whereas the PDE contains the Eclipse adaption of the OSGi specifications and is used to provide the plug-in features. Note that in some cases plug-ins are called *bundles*, too. A bundle is the OSGi counterpart to an Eclipse plug-in.

3.1.1. Plug-In Development

The PDE provides tools and interfaces to create loose couplings between different components and to modify and extend the behavior of the Eclipse platform.

[1] http://www.android.com/

[2] http://www.ibm.com

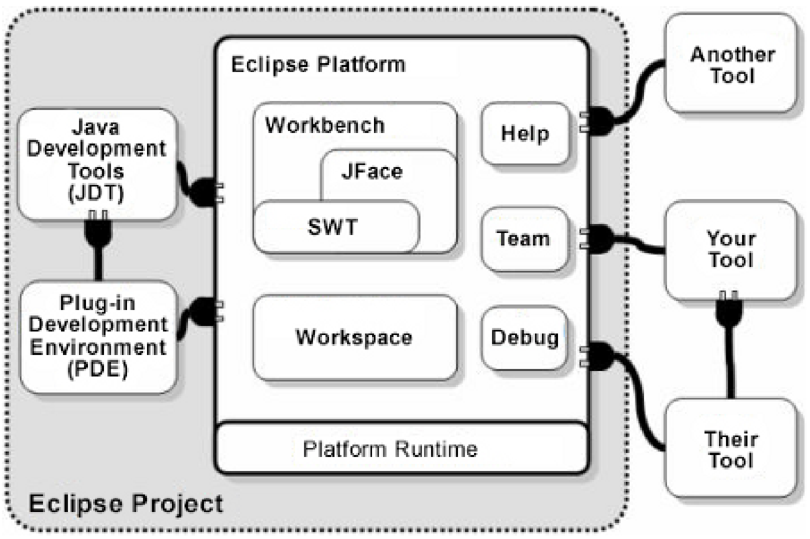


Figure 3.1.: Eclipse Platform Architecture [2]

To create a public interface to a project, or even to single components of a project, developers can define *extension points*, which are XML schema definitions, comprising a set of properties that are used to create instances of the given components. Figure 3.2 shows an example usage of the *org.eclipse.ui.menus* extension point, which is used to add a menu called *KIELER* to the Eclipse main menu.

Furthermore, the PDE can be used to create a Rich Client Application (RCA), which is basically a modified Eclipse application that has been created by using extension points provided by the Rich Client Platform (RCP). By using the RCP, developers can use the existing and established Eclipse UI and messaging components to create a GUI application while focusing on the logical parts of the application. These tools have already been used to create a wide range of applications[3].

3.1.2. The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is *the* common modeling framework in Eclipse. It contains a custom technical space, which is based on the Ecore M3 model and uses XMI for storage of models. The Ecore model has been created as an implementation of the MOF 1.0 specifications, and modified based on the experience gained while using Ecore for a number of tools. The current version of the MOF 2.0 specifications contains a subset, called the Essential MOF (EMOF) model, which is very similar to the Ecore M3 model and can be used transparently in EMF.

EMF contains several sub-projects for creating and processing models. The most common way of creating Ecore models is either to use the basic editor, included in the EMF core project, or to import a model from an external file. The EMF project

[3]http://www.eclipse.org/community/example_rcp_applications_v2.pdf

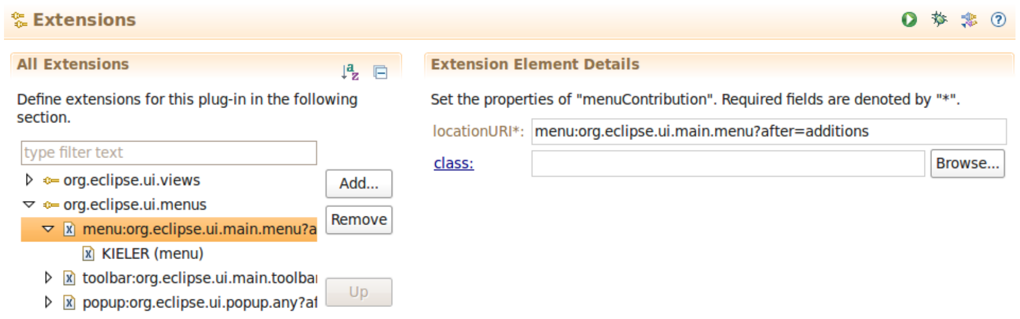

Extensions
All Extensions
Define extensions for this plug-in in the following section.
type filter text
org.eclipse.ui.views
org.eclipse.ui.menus
menu:org.eclipse.ui.main.menu?a
KIELER (menu)
toolbar:org.eclipse.ui.main.toolbai
popup:org.eclipse.ui.popup.any?af
Add...
Remove
Up
Extension Element Details
Set the properties of "menuContribution". Required fields are denoted by "*".
locationURI*: menu:org.eclipse.ui.main.menu?after=additions
class:
Browse...

Figure 3.2.: Usage of the org.eclipse.ui.menus extension point

supports several languages for importing models, e. g. XSLT, UML or annotated Java.

The second important part in the EMF core project is the code generator, which is capable of generating a set of Java projects from a single Ecore model, e. g. a simple editor that may be used to create and modify instances of the model.

In addition to the core project, EMF comprises projects for creating and executing queries against EMF models[4], validation of models[5] as well as comparing different models or different versions of the same model[6].

3.1.3. The Graphical Editing Framework

The Graphical Editing Framework (GEF) comprises two projects that have been created to aid developers in creating graphical editors for arbitrary application models. The first project, called Draw2d, contains an entire set of tools to display graphical content, including figures, connections, multiple transparent layers and rudimentary layout algorithms.

The second project, simply called GEF, provides a MVC architecture, used to interconnect the graphical part of an editor with the logical. As shown in Figure 3.3, the GEF components can be separated into three categories: The controller, the request/command interface and the event handlers.

The controller is responsible for creating graphical elements that are displayed in the view part, which are usually Draw2d elements. Furthermore, the GEF controller provides methods for accessing the model element that has been used to create a certain graphical element in the view part, which is a mandatory prerequisite for the structure-based editing framework proposed later on.

The link between model and view part of an element is called an *EditPart*. Beside this linkage, the EditPart is also responsible for providing editing operations by connecting an EditPart to a set of interfaces called *EditPolicies*. Those policies comprise all operations that may be executed on the corresponding EditPart.

[4] http://www.eclipse.org/modeling/emf/?project=query2
[5] http://www.eclipse.org/modeling/emf/?project=validation
[6] http://wiki.eclipse.org/index.php/EMF_Compare

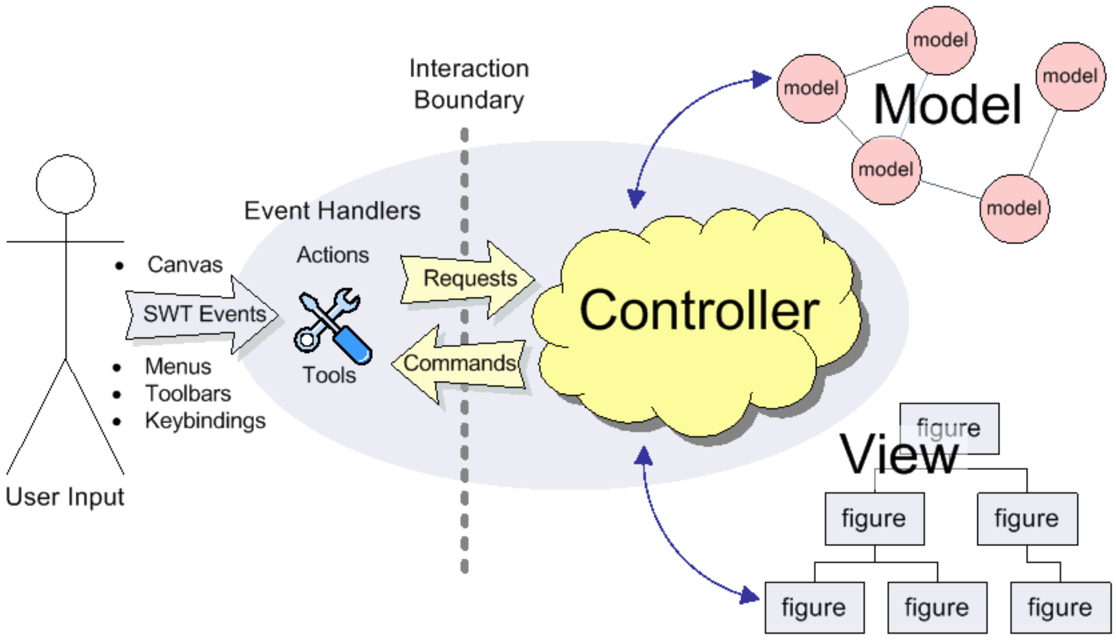


Figure 3.3.: The GEF MVC architecture

To query an EditPart of its supported operations, a *Request* is created by the request/command interfaces, illustrated in the center of Figure 3.3. Such a request is created every time an event occurs and is processed by the event handlers. This request is forwarded to all EditParts that are connected to the currently selected diagram elements. These EditParts will return a *Command* if the given operation is supported. Those commands may afterwards be executed either individually or by passing them to the GEF command stack.

The event handlers are the common interface for user interactions and contain a wide range of interaction capabilities[7], e. g. for capturing mouse and keyboard events.

3.1.4. The Graphical Modeling Framework

The Graphical Modeling Framework (GMF) is a project that has been built on top of the EMF and GEF projects to help developers creating enhanced graphical editors, commonly called *diagram editors*. What at first glance may seem just like a second GEF project that generates editors from application models is actually a framework for the development of extensive graphical applications, based only on a small set of configurations and templates.

Just like the EMF project, GMF contains a generative component that is used to generate Java code for a given configuration and a runtime infrastructure that comprises very useful tools for the development of graphical editors, e. g. reusable

[7] http://help.eclipse.org/ganymede/topic/org.eclipse.gef.doc.isv/guide/guide.html#interactions

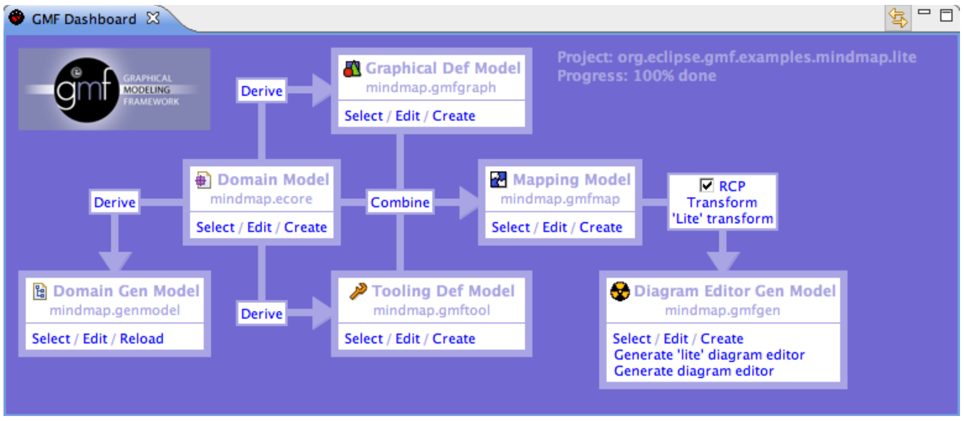


Figure 3.4.: The GMF dashboard

application components, and a command framework that is used to interconnect a GMF editor with both GEF and EMF.

Figure 3.4 shows the GMF dashboard that visualizes the required components and assists developers through the development process for a GMF editor.

Even though the generated diagram editor will be ready to use, the extensive usage of GMF showed that the development of enhanced graphical editors requires a large amount of additional work on the generated code; nevertheless, the benefits of using GMF over the development of an editor from scratch or by extending an GEF editor, are still worth the additional effort.

3.2. KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[8] is a research project developed in the real-time and embedded systems group at the Christian-Albrechts-University of Kiel. It aims at the development of techniques and tools to enhance graphical model-based design of complex systems and is based on the pragmatics of model-based design, introduced by Fuhrmann and von Hanxleden [13].

This project is the successor of the KIEL project[9], which is a standalone Java application for the construction of Statecharts. The KIEL project already contains several features that have been adopted and enhanced in KIELER, e. g. structure-based editing, automatic layout and textual editing; however, since the KIEL project is a monolithic tool, which is dedicated to Statecharts only, the KIELER project has been created to allow a more generic use of the paradigms implemented in KIEL. KIELER has been developed as an open-source Eclipse RCA to create a framework that may be used with a wide range of graphical editors.The following sections will

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

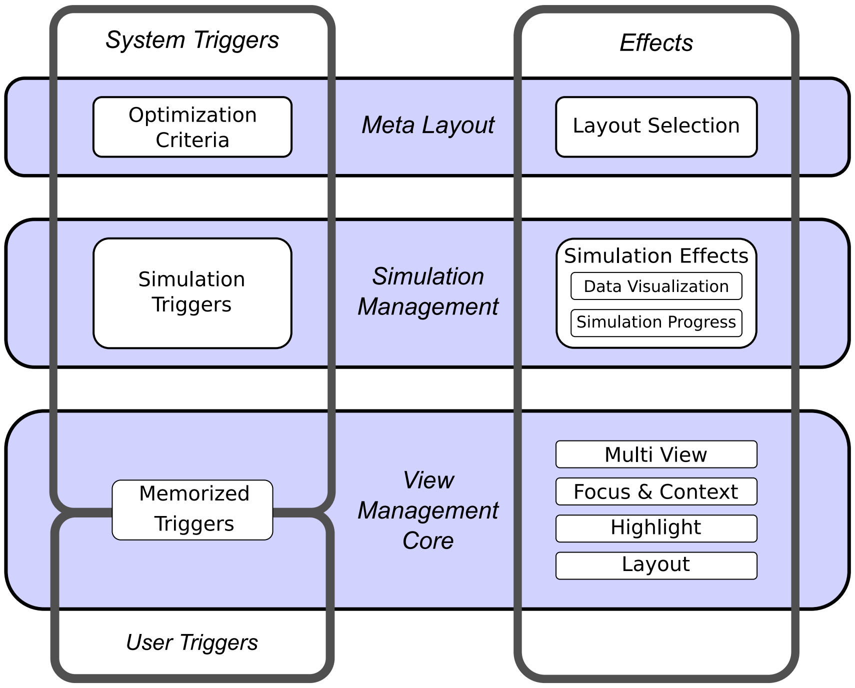


Figure 3.5.: The view management concept [13]

give a short overview of the features and editors that are provided by the KIELER project, focusing on features that have been used in this thesis.

3.2.1. KIML

The KIELER Infrastructure for Meta Layout (KIML) project, developed by Arne Schipper [26] and Miro Spönemann, addresses the automatic layout of diagrams and provides interfaces to common layout tools, e. g. the GraphViz[10] software, as well as implementations for the layout of different graph types, e. g. KIELER Layout of Dataflow Diagrams (KLoDD) [28].

Since the automatic layout of diagrams is one of the most important key enablers for structure-based editing, the KIML project is a mandatory requirement for the KIELER structure-based editing framework, introduced later on.

3.2.2. View Management

Another quite important project for the KIELER structure-based editing framework is the view management, developed by Nils Beckel [8].

The aim of the view management project is to aid users in creating or modifying graphical models by giving a visual feedback after an operation has been executed.

[10]http://graphviz.org

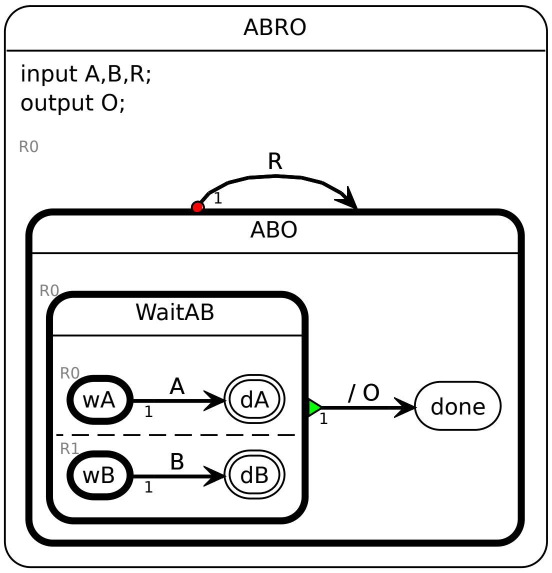


Figure 3.6.: SyncChart for ABRO

The visual feedback, called an *effect*, may vary from simple operations, e. g. highlighting a state by drawing a colored bounding box, to complex operations, e. g. hiding all graphical elements that are not part of the current editing context.

This project has been built on the listener pattern and contains the following extension points:

- Trigger

  Triggers represent the listener objects, which may be interconnected with arbitrary events, e. g. a user input, a simulation event or the execution of a structure-based editing operation.

- Effects

  Effects contain the actual Java code that will be executed when the assigned event has occurred.

- Combinations

  Combinations are the connections between a trigger and a set of effects. Whenever a trigger is activated, the combination, containing the trigger, will evaluate whether the effect should be executed and calls the associated methods in the effect class.

Due to its generic interface, the view management project has been tightly integrated into the KIELER structure-based editing framework.

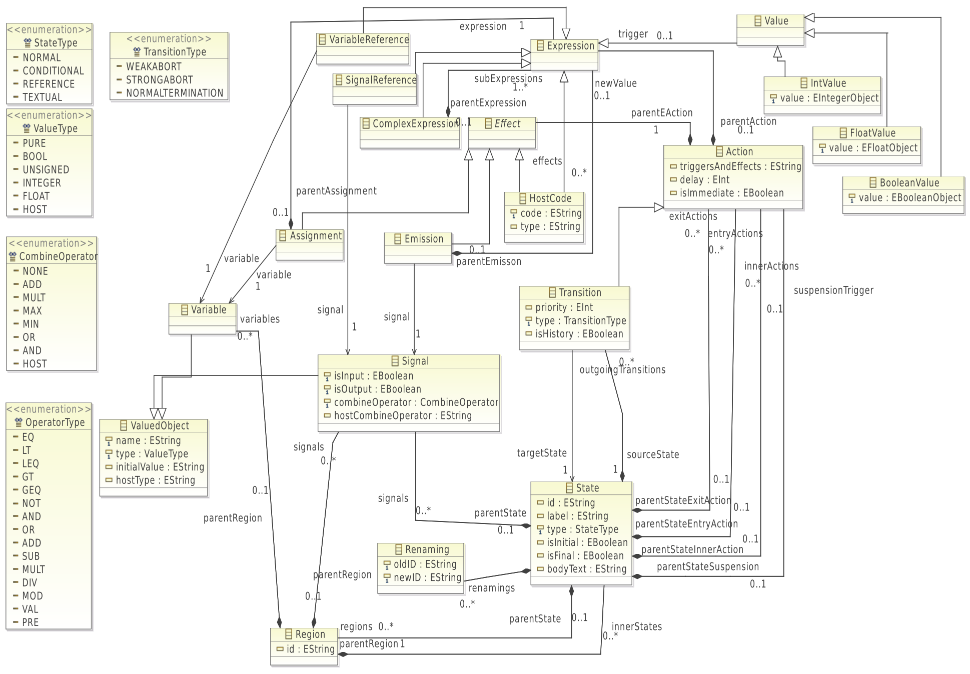


Figure 3.7.: The SyncCharts metamodel

3.2.3. ThinKCharts Editor

The Thin KIELER SyncCharts Editor (ThinKCharts) has been developed by Matthias Schmeling [27] as a graphical editor for SyncCharts [5] in Eclipse. SyncCharts are an advancement of Mealy Machines [18], which have been extended by introducing parallelism, hierarchy, and signal broadcast to model concurrent real-time behavior. Figure 3.6 shows an example of a simple SyncChart. The model of the ThinKCharts editor has been created in the EMF space, therefore it conforms to the Ecore meta-metamodel.

In general, the graphical representation of a SyncChart comprises only two different types of elements: states and transitions.

The states are of two different kinds: they are either simple, i. e. they may have a label and incoming and outgoing transitions, or complex, meaning that they contain an arbitrary number of inner regions and states.

Additionally, states and transitions may have a set of attributes that gives them special semantic meanings, e. g. initial or final states, priority and delay of transitions as well as some different types of transitions.

Because the ThinKCharts editor will be frequently used for examples in this thesis, the full meta-model, on which the ThinKCharts editor is build upon, can be found in Figure 3.7.

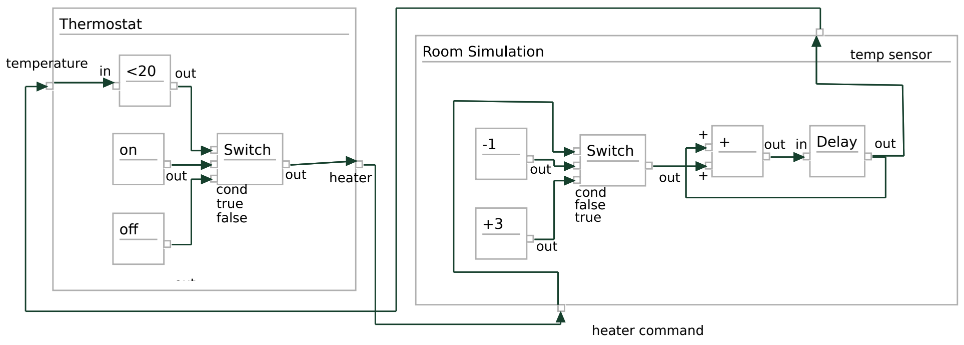


Figure 3.8.: Data flow diagram simulating a heater control

3.2.4. Data flow Editor

KIELER of Dataflow (KoData)[11] is a simple editor for the graphical editing of data flow diagrams. Data flow languages are a common tool for modeling the behavior of control systems, e. g. as shown in Figure 3.8, where a heater control system has been modeled.

In contrast to SyncCharts, data flow diagrams are built from nodes, sometimes referred to as operators, actors, subsystems or simple boxes, that may comprise an arbitrary number of input and output *ports*. These ports are used to interconnect boxes by inserting a transition between an input and an output port. Additionally, the boxes may also be used hierarchically, i. e. a box can contain other boxes. Furthermore, in some of the most common data flow modeling tools, e. g. Matlab/Simulink (The Mathworks) or SCADE (Esterel Technologies), the diagram boxes may comprise logic operators that are used to simulate a model or generate platform-specific code.

Because the KoData editor is only a small case study project, it only contains primitive boxes with ports and connections between those boxes; nevertheless, the KoData project has been used to create basic data flow operations with the KIELER structure-based editing framework.

3.2.5. Additional Projects

In addition to the projects that are related to the actual modeling process, KIELER also contains projects that are addressing the semantics and the execution of models [22], e. g. by using the Ptolemy [17] project, as well as the generation of code from models, for example to generate synchronous C (SC) / Java (SJ) [31] code.

Beside the graphical modeling, KIELER also focuses on the textual representation of models and the synchronisation between the graphical and the textual views. The KITS language and the KIELER Textual Editing Framework (KITE) project, both developed by Özgün Bayramoglu [7], are trying to build a bridge between a cus-

[11] http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KoData

tom textual Domain-Specific Language (DSL), called KIELER Textual SyncCharts (KITS), and the ThinKCharts editor.

3.3. Xtext/Xpand

The Xtext/Xpand projects have been developed by the b+m group[12] as part of the openArchitectureWare (oAW) framework. In 2009 those projects have been integrated into the Eclipse Modeling Project.

The purpose of the oAW framework is to give a complete set of tools for model-driven software development. It consists of two main components: Xtext and Xpand.

The Xtext project is a framework for the development of textual domain-specific languages (DSL). It provides tools for creating such a textual language by using an EBNF-like notation with additional features and generating a metamodel as well as ready-to-use Eclipse editors from the given description. The generated editor provides most of the state-of-the-art features of a textual editor, e.g. syntax highlighting and code completion.

The Xpand generator project contains a set of languages, namely Xpand, Xtend and Check, that are based on the same common expression language, which is a mixture of OCL and Java. The most important built-in elements of the expression languages are:

- Basic arithmetic operations and boolean operators, e. g. +,-,*,/,==, !=, <, >.
- Collection types

  Collection types comprise collections, lists and sets, which provide a wide range of very useful operations known from functional and declarative programming languages, e. g. filtering elements or reversing a list.
- Simple data types

  The simple data types provided by the Xpand languages are String, Boolean, Integer and Real.

Those languages are used to provide support for different parts of the model driven development process.

The Xpand language allows to define templates, which are used to generated arbitrary text from a given input model. This language is often used to create code generators, e. g. for generating a Java program from a model.

The Check language is used to define constraints for a model, based on its corresponding metamodel. These constraints can be used to check if a given input model is valid, e. g. after a transformation has been performed.

The last component of the Xpand generator framework is the Xtend language. Xtend is used to define so-called *extensions*, i. e. model transformations, which are used to extend a given input model based on the semantics of its metamodel. Because

[12] http://www.bmiag.de

Listing 3.1: Example transformation for Xtend

```
1  import syncharts;
2  import ecore;
3
4  //Changes the name of the given state
5  Void setStateName(State state, EString name):
6     state.setStateName(name);
7
8  //Creates a transformation between source and target state
9  Transition createTransformation(State source, State target):
10    let t = new Transition:
11     t.setSourceState(source)->
12     t.setTargetState(target)->
13     t;
14
15 //Sets the object that should be selected after the transformation is executed
16 Void setSelection(Object object):
17  JAVA de.cau.cs.kieler.ksbase.ui.utils.TransformationUtils.
        setPostTransformationSelection(java.lang.Object)
18 ;
```

Xtend is used as the default transformation framework in the reference implementation proposed later on, we will now have a more detailed look at the most important featuers of Xtend.

As shown in lines 1 and 2 of Listing 3.1, a custom M3 model is imported into an extension file by using the *import* statement. Since *import* is generally used to load namespaces into the common expression language, it is followed by the name of the actual namespace that has been defined by the metamodel, e.g. the *syncharts* namespace, declared in the corresponding Ecore file. In addition to the import of custom models, the Xtext/Xpand project comprises a set of default metamodel contributions, such as EMF, UML, and XML schema that may used in extensions.

Xtend provides two different types of extensions that are either used to create an object or to define an in-place transformation. The type of an extension is defined by its return value and may either be the keyword *Void*, which denotes an in-place extension or an arbitrary type value, which denotes the construction of a new object with the given type. Examples of the different types of transformations are shown in lines 5 and 9.

Due to the functional nature of the common expression language, Xtend methods are interpreted as single expressions without side-effects. However, since some model transformations require the use of side-effects, Xtend allows to create sequences of expressions, denoted by the -> keyword. Expressions comprised in a sequence will be evaluated step-by-step and return the value of their last expression, as shown in lines 9-13. An additional consequence of the functional nature is the variable handling. As shown in line 10, local variables are defined by using the *let* keyword, followed by an expression that defines the scope of the variable. Because of the absence of side-effects, it is not allowed to change the value of a variable, except for the use of setter methods.

Listing 3.2: A Java class that may be called from Xtend

```
/**
 * Utilities that may be called from Xtend.
 *
 * @author mim
 *
 */
public class TransformationUtils {

   /** The object to select after the transformation has been executed. **/
   private static volatile EObject selection = null;;

   /**
    * Sets the object that should be selected after the transformation has been
    * executed.
    *
    * @param e
    *            The object to select
    */
   public static void setPostTransformationSelection(final Object e) {
      TransformationUtils.selection = (EObject) e;
   }
}
```

Furthermore, Xtend provides *create* and *cached* extensions, which are an advancement of transformations that are used to create new objects. If an extension is marked with the *create* or *cached* keyword, the object that is created will be cached by Xtend. This mechanism is used to avoid the construction of the same object twice, e. g. if an existing element needs to be inserted as a reference. The difference between those keywords is that the result of a *create* extension will be created before the body is evaluated, which avoids possible circular dependencies and deadlocks.

In addition to the common expression language, Xtend allows the execution of static Java code, as shown in line 17 of Listing 3.1 and in Listing 3.2, which may be used to execute operations that can not be expressed directly in the expression language or access data that is not accessible via Xtend.

Chapter 4

A Transformational Approach to Structure-Based Editing

"That's the second biggest monkey head I've ever seen!"

from: "The Secret of Monkey Island"

In this chapter we will propose an approach for introducing structure-based editing of graphical models to the Eclipse modeling platform. This approach is based on the well-known technique of model transformation, as introduced in Chapter 2.1.1.

Before thinking about a novel editing approach, we will have to have a look at the state-of-the-practice paradigms, provided by existing graphical editors and modeling tools, to figure out which parts of the processes are the most time consuming and which of them may be improved by the introduction of structure-based editing.

After the evaluation process, a general overview of the approach to structure-based editing will be given, followed by a more detailed look at the different parts of the architecture, including some guidelines and requirements for a generic structure-based editing framework.

4.1. State-Of-The-Practice in Editing

One of the most important aspects to keep in mind when developing a graphical editor and choosing editing paradigms and tools is the preservation of the users *mental map* [24] of the diagram. The term *mental map* describes the psychological process of creating a representation of an image in the mind of an observer. In the case of a graphical diagram, the mental map may comprise the locations and names of the elements, so that an observer will be able to find elements, he or she has seen before, significantly faster every time he or she will look at the diagram and even know where to find elements that are currently out of sight.

Destroying this mental map while modifying an existing diagram, e. g. by creating overlapping elements or performing inconsistent automatic layout, should be carefully avoided, because the observer may be confused by the new arrangement and needs additional time to rebuild his or her mental map, what will most certainly lead to an increase of necessary modeling time.

There are at least two different approaches of how to preserve the observers mental map for a graphical diagram: the first one is that in each editing step the changes to the appearance of the model should be minimal [9], whereas the second approach proposes the definition of a *normal form* for models, meaning that models with the same content should always be visualized identically.

For a structure-based editing framework we propose the second approach, e. g. by performing an automatic layout after each editing step. This ensures the preservation of the mental map and that a trained observer can predict how the diagram will change after a structure-based operation has been executed.

Drag and Drop WYSIWYG

The common paradigm for editing graphical models is a mixture of What-You-See-Is-What-You-Get (WYSIWYG) and drag-and-drop (DND) editing. WYSIWYG describes a constant visual feedback and may include synchronisation between semantic and graphical data. This approach is widely used and the *de-facto* standard for graphical editors.

DND characterizes the common way of interacting with graphical applications. It is used in most of the modern operating systems and applications. The abstract workflow of DND in graphical modeling tools is that the user selects an element and uses the mouse cursor to drag it to the desired location where the element will be dropped and inserted into the diagram. The selected element may be an existing object, what, in most cases, will lead to a simple change of the elements location as well as an arbitrary representation of a new element, what will lead to the creation of a new graphical object. In some cases, the term DND also includes modification processes, e. g. connecting diagram elements, bending those connections or resizing elements [25]. In this context we will also use this broad definition of DND to describe a whole class of editing techniques.

The DND approach includes various, potentially time consuming, editing processes, for example the need to use the mouse to choose an appropriate location for a new element, which may lead to even more additional tasks for a user, e. g. creating free space for the element.

Figure 4.1 shows an example of how drag-and-drop editing may be used to modify a SyncCharts diagram by adding a new state element between two existing states.

The first required action is to resize the model to gain enough free space for the new state and its transition (a+b). After a new state has been inserted and connected to one of the states (c+d), the user has to select the existing transition and reassign its target to the new state (e).

When all states have been connected, the user needs to move and resize the diagram elements to obtain a satisfying layout. This may include the subconscious desire to preserve the mental map by creating a layout which is mostly identical to the original layout (f+g).

The final layout may also be achieved by using an automatic layout tool. Although this may lead to additional efforts for gaining free space before inserting a new state,

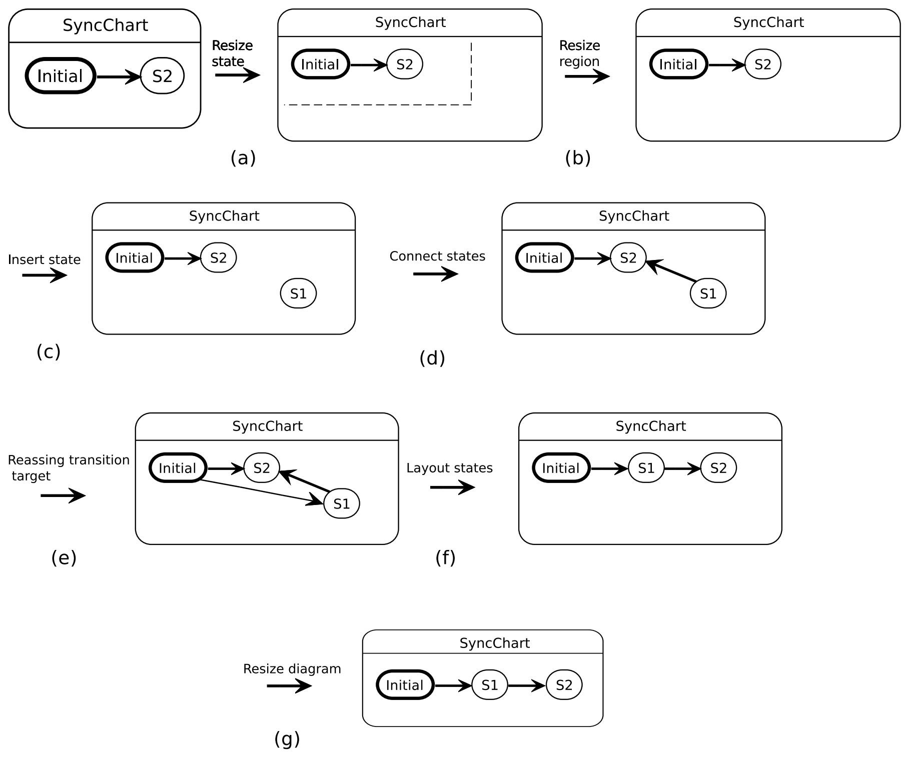


Figure 4.1.: Manually inserting a state

because most of the automatic layout tools are trying to create a preferably compact layout.

Furthermore, DND editing is normally restricted to single elements, which could lead to a large number of repetitive actions, e. g. for creating multiple successors to a state.

Summary

The evaluation of existing editing tools and paradigms has shown that there are at least five factors influencing the performance of editing graphical models:

1. Preservation of the mental map.

   This factor influences the users familiarity with a diagram. If the diagram constantly changes, users will be confused where elements should be added, especially when the diagram contains hierarchy.

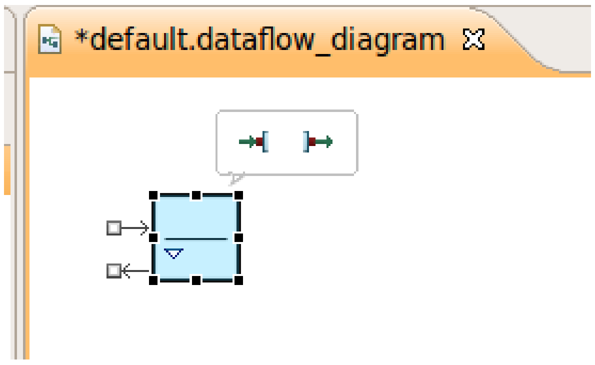


Figure 4.2.: PopupBar for the KoData editor

2. Minimal user interaction.

   When editing graphical models, a user should only have to use as few actions as possible, i.e. the number of necessary mouse gestures should be minimal.

3. Combine common, complex actions.

   Common actions that comprise a fixed set of operations should be combined to a new command. For example, the action of adding a new element to a diagram and connecting it to an existing element may be combined with an operation where an existing element is selected and both the new element and the transition between them, is inserted automatically.

4. Enable fast execution of actions.

   To allow a fast interaction with a diagram, the operations should be accessible as fast as possible. Some of the potential interaction tools are all kinds of menus and keyboard shortcuts.

5. Allow to add custom actions.

   To consider the different strategies of editing for individual users, it should be possible for everyone to define custom actions that fit to their personal needs.

4.2. Approach

As shown in the previous chapter, common modeling tools and the WYSIWYG DND editing paradigm fail to enable fast and efficient modeling. Some modeling tools are trying to solve those problems by introducing specialized solutions for common operations GMF, for example, contains a feature called *PopupBars*, which is a context sensitive popup menu that can be used to add child elements to diagram objects. This mechanism provides a faster way of adding elements compared to the GMF palette. Furthermore, the *PopupBar* comprises only valid child elements for the selected diagram object. Figure 4.2 shows a *PopupBar* for the KIELER data flow editor, which allows the addition of input and output ports to a box. Nevertheless, none of those features provide a sufficient solution to the requirements introduced in the previous section.

To develop a solution that provides satisfying solutions for all important modeling factors, we will introduce a technique that will be used in a graphical editor too, but only modifies the semantic structure of a model, without using any graphical information. This technique is called *structure-based editing*, or *structural editing* and has already been used for a wide range of graphical and textual editors [29]. Actually, most of the modern IDEs comprise structure-based textual editors for a programming language, because they are working on the language metamodel to support the developer when creating an application, e. g. by offering context sensitivity or code completion.

We will combine this structure-based editing approach with the technique of endogenous in-place model transformation, as introduced in Chapter 2.1.1, to propose a novel approach for editing graphical models in Eclipse. This transformational approach allows us to use established frameworks to define and execute transformations.

One key enabler for this approach to structure-based editing is the existence of an automatic layout framework that provides algorithms to create a layout that preserves the mental map. Because the automatic layout will be performed after each editing operation, a good performance of the layout algorithms, even for larger diagrams, is an important requirement. To provide a preferably high performance, it may be acceptable for layout algorithms to create fast, non-optimal results. Furthermore, the transformational approach raises an important requirement for the graphical framework: since only the semantic model is modified, the framework needs to be able to synchronize the graphical view with the semantic model to create or modify elements that have been changed during the execution of a model transformation. This is a special interest for UML editors, because the semantic UML model is often used to create a variety of diagrams, e.g. class, sequence or activity diagrams.

4.3. Overview

The transformational approach taken here can generally be seen as an interface between a graphical editor and a transformation framework, as shown in Figure 4.3. In respect to this interface structure, the approach has been split into three parts:

1. The interconnection to the graphical editor.

2. The interconnection to the transformation framework.

3. A link layer that is used for configuration, persistent storage and data conversion between the interfaces.

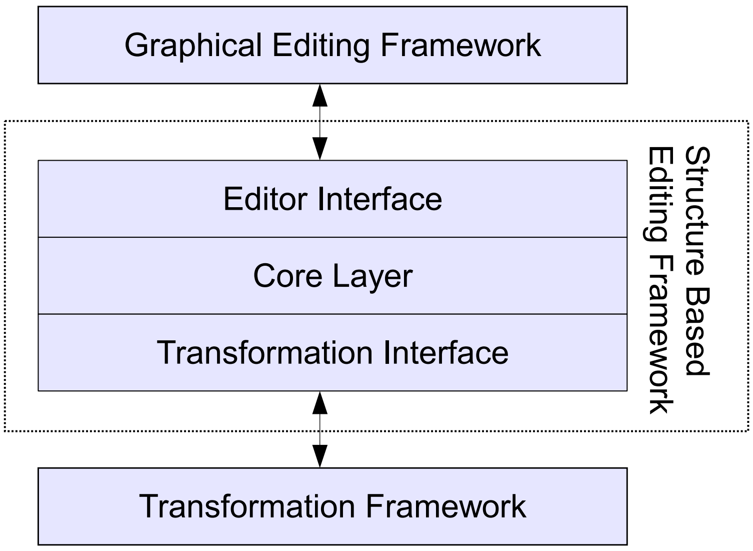


Figure 4.3.: Interfaces to structure-based editing framework

4.4. Editor Interface

The editor interface is responsible for the bi-directional interaction between the graphical editor and the structure-based editing framework. This includes all contributions to the user interface, which allows performing structure based editing operations, as well as the editor-dependent data structures that are necessary to execute those operations. Possible contributions are: menus, e. g. main menus, toolbars and context menus; keyboard shortcuts and other, advanced features, which may depend on the actual graphical editor.

Another important task for the editor interface is to provide a context dependency for those User Interface (UI) contributions, e. g. a user should not be able to perform operations, which are not defined for the semantic object that is represented by the currently selected diagram element. This may be achieved by either hiding or disabling the corresponding contributions.

4.5. Transformation Interface

The transformation interface is the corresponding counterpart to the editor interface and is responsible for transmitting the selected diagram elements to the transformation framework and execute the in-place method that is assigned to a selected structure-based editing operation.

The most complex part of the transformation interface is the mapping of diagram elements, resp. their semantic counterparts, provided by the editor interfaces and the core layer, to the actual parameters of the model transformation method. The mapping of simple types may be straightforward and only require an ordering

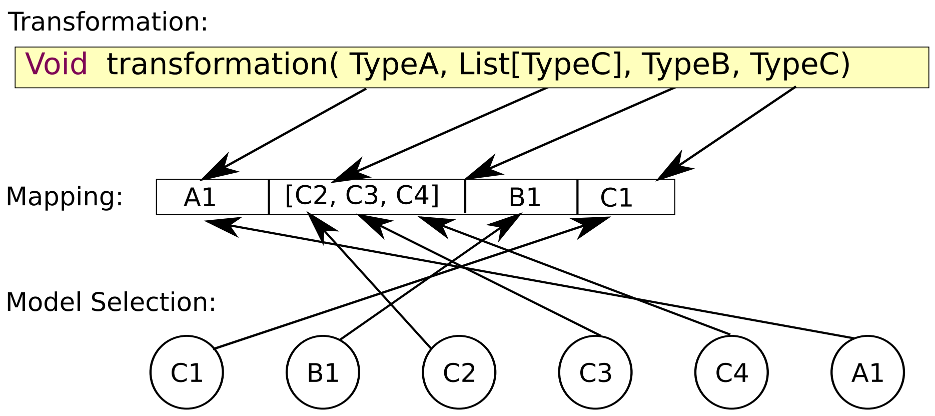


Figure 4.4.: Mapping and reordering Parameters

mechanism, depending on the underlying transformation framework. The handling of special types, however, requires an appropriate handling to provide a transparent use in the graphical editor. The particular amount and kind of special types also depends on the transformation framework that is used, but some of the most common types are: collections, lists, maps, arrays, etc. The combination of the mapping of simple and special types leads to the following rules for a mapping algorithm:

- The mapping algorithm has to ensure that the number of selected elements matches the number of required parameters.
- The order of the parameter types and selected diagram elements must be transparent, i. e. if the transformation requires parameter types A and B, the user is allowed to select the elements in arbitrary order, e. g. B, A.
- If a transformation contains a collection parameter, e. g. a list of elements, the mapping must find the best match for the list, i. e. collect either all selected elements of the matching type or select the first sequence of elements in the current selection.

The mapping algorithm that has been developed for this thesis solves the list mapping task by searching the first sequence of the list parameter type in the current selection, e. g. if the selection contains a single and a sequence of elements with the same type, separated by a different type, the algorithm will select the last sequence, as shown in Figure 4.4. This allows the mixture of single elements and lists with the same type in one transformation. Listing 4.1 shows the pseudocode of the implemented parameter mapping algorithm.

Figure 4.5 shows an example of how the algoritm is used for ordering and mapping diagram elements to transformation parameters. The transformation, as shown in Listing 4.2, is performed on a single state element, a transition, and a list of target states. The resulting model will comprise transitions between the source and the target states with the same triggers and effects string as the selected transition.

Listing 4.1: Parameter mapping algorithm

```
foreach{Transformation parameter p}
{
   if{p is simple parameter} {
      foreach{Selected element e}{
         if{e.type == t.type} {
            Assign e to p;
            break;
         }
      }
   }{
   elseif {p is collection parameter} {
      forEech{Selected element e} {
         if{e.type == t.type} {
            if{e followed by k with k.type == t.type} {
               Assign e to p;
               while {e.next.type == t.type } {
                  Add e.next to assignment;
               }
            }
            else if{Selected elements contains sequence f with
               f.types == t.type and f.size > 1 } {
                  Assign f to p;
               } else {
                  Assign e to p;
               }
               break;
            }
         }
      }
   }
    selectedElements.remove(p);
}
return selectedElements.isEmpty();
```

4.6. Core Layer

The core layer is responsible for the interconnection of the editor and transformation interfaces as well as for the editor dependant configurations and the persistent storage of user-defined settings.

A special interest for the core layer is the interchangeability of configurations, i. e. users and developers must be able to easily integrate structure-based editing operations from external sources to a custom graphical editor. How this interchangeability is actually implemented strongly depends on the graphical editor and the framework that are used. A detailed description of how it may be implemented by using the Eclipse and GMF features will be given in the next chapter.

Pre- & Post-transformation actions

An additional task for the core layer is the coordination of actions that need to be executed before or after a transformation has been performed. Those actions may comprise the temporary deactivation of features, which may interfere with the

Listing 4.2: Duplicate transition transformation

```
Void duplicateTransition(State source, Transition transition, List[State] targets)
    :
   targets.forAll(e|connectWithTriggers(source,transition,e))
;

Void connectWithTriggers(State source, Transition triggers, State target):
 let transition = new Transition:
   transition.setTriggersAndEffects(triggers.triggersAndEffects) ->
   transition.setSourceState(source) ->
   transition.setTargetState(target) ->
   source.outgoingTransitions.add(transition)
;
```

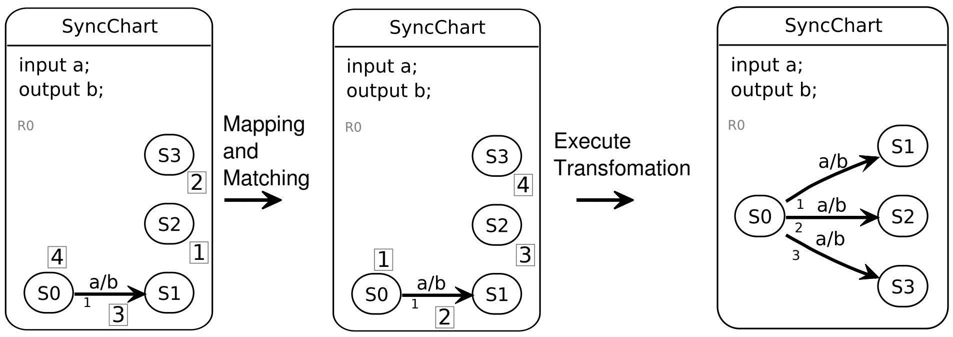


Figure 4.5.: Example parameter mapping. The numbers indicate the user selection on the left image and the computed mapping on the center image.

transformation, a validation of the transformed model or an arbitrary kind of visual feedback, e. g. selecting a newly inserted diagram element or performing an automatic layout.

4.7. Summary

In this chapter we proposed a generic approach to structure-based editing that is based on the technique of model transformation. Figure 4.6 shows how the insertion of a state element between two existing states, c. f. Figure 4.1, is achieved with structure-based editing and the pseudocode transformation shown in Listing 4.3, followed by the execution of automatic layout. This approach is used to provide fast and efficient modeling based on a set of efficiency factors. In contrast to existing editing paradigms, the given approach matches all these factors:

- The preservation of the mental map is achieved by performing automatic layout and morphing after a transformation has been executed.
- The user interaction is minimized to avoid additional efforts, achieved when

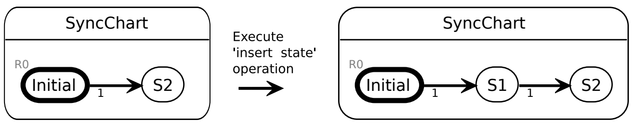


Figure 4.6.: Inserting a state with structure-based editing

Listing 4.3: Insert state transformation (pseudocode)

```
insertState(Transition t)
{
   State s = new State;
   Transition v = new Transition;
   v.setSourceState(s);
   v.setTargetState(t.getTargetState);
   t.setTargetState(s);
}
```

using DND editing by executing operations on the semantic model of a diagram instead of using the graphical representation.

- Since the operations are built by using model transformations, arbitrarily complex operations may be combined to a single action.
- The execution of operations may be assigned to a multitude of user actions, e. g. mouse clicks, keyboard shortcuts, etc.
- Because of the generic and extensible architecture of the structure-based editing approach, it is easily possible to allow the dynamic addition of operations by using the features of the underlying graphical framework.

Chapter 5

KIELER Structure-Based Editing

"I am Error"

from: "The Adventure of Link"

In this chapter we will present a reference implementation of the approach proposed in the previous chapter. The implementation has been realized in the context of the KIELER project, so some of its parts are currently available for GEF/GMF based editors only.

To follow the general naming guidelines of KIELER, the project has been named: KIELER Structure Based Editing (KSBasE - pronounced: Kay-Space).

Section 5.1 will give an overview of the structure and additional tools used in this implementation. In the following sections we will have a more detailed look at the most important parts of the KIELER Structure-Based Editing (KSBasE) framework, followed by example configurations for two editors, which have been developed with the KIELER project, and general transformations that may be used in UML editors.

In the last section of this chapter we will have a look at an evaluation of KSBasE, to examine the gain in performance when using structure-based editing over the default GMF DND editing.

5.1. Architecture

The KSBasE framework has been developed by following the 3-layered architecture proposed in the previous chapter. The architecture has been transported to the Eclipse environment by creating a set of plug-ins. Furthermore, the model transformation framework connection layer has been developed in the *de.cau.cs.kieler .core.model* project to provide an easy-to-use interface to model transformation for other KIELER projects. To create a preferably generic framework, the model project contains an interface that needs to be implemented to create a connection to an actual transformation framework. Furthermore, the project comprises an abstract transformation class that contains the general properties of a model transformation, which may also be extended to adapt to a specific framework. In addition to the

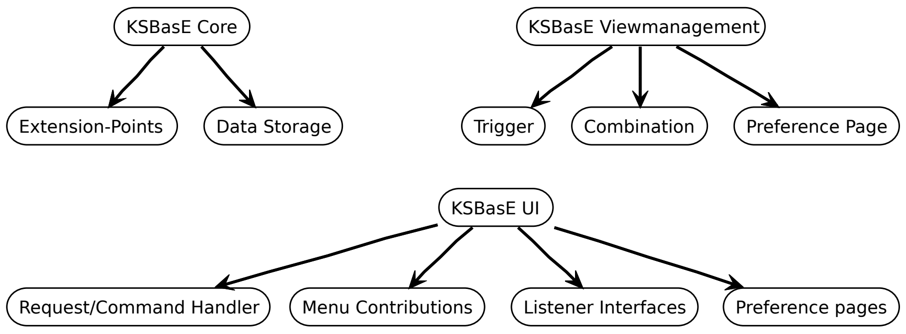


Figure 5.1.: Architecture of the KSBasE framework

sole interfaces, the model project contains implementations for Xtend, the default transformation framework that is used in the KSBasE project. Xtend has been chosen, because after evaluating the frameworks introduced in Chapter 2.2, Xtend turned out to be the best match for all given criteria. Furthermore, Xtend has already been used in other KIELER projects, which avoids adding more external project dependencies.

Figure 5.1 shows a structural overview of the KSBasE plug-ins and their most important components. In addition to the 3-layered architecture, the KSBasE framework comprises another project, called the KSBasE view management. This project is used to interconnect KSBasE with other projects, including the KIML framework for automatic layout. In the following sections, we will take a closer look at the plug-in projects as well as on the interconnections between the KSBasE framework and other KIELER projects.

5.2. KSBasE Core Project

The KSBasE core project contains the data storage classes and the main entry point for using KSBasE, called the *TransformationManager*. This manager is used to initialize both previously stored user settings and new editor configurations that have been added by using the KSBasE extension point schema definition.

Eclipse offers two mechanisms for storing user defined settings: a preferences file and a plug-in state location folder. The preferences files uses simple name-value pairs, which may be used to store default types, e. g. strings, numbers and boolean types. This mechanisms, however, turned out to be inappropriate for storing KSBasE user settings, e. g. because of the large number of transformations that may be defined for a single editor and the storage of a transformation file. Instead, the Java serialisation mechanism has been used to store user settings in the projects state location folder, which is an Eclipse-featured folder that either exists or will be created when first requested by a plug-in.

5.2.1. Extension Points

The KSBasE core projects contains two extension point schemata: one for configuring structure-based editing features for a target editor and one for activating projects without the need to introduce a strong dependency from the KSBasE projects. The latter extension point schema has been developed to allow developers to create projects, which contain implementations of a listener, provided by the KSBasE UI project, see Chapter 5.3.2, and register them to the framework without the need of manually activating the project by loading one of its classes.

The configuration extension point is the preferred way of adding structure based editing operations to an editor. It comprises all elements that are necessary for enhancing an editor as well as the possibilities to use custom transformations frameworks, e. g. for using QVT or ATL instead of Xtend. The full extension point scheme can be found in Appendix A. The most important elements of the configuration are:

Configuration

The configuration element is the main extension point component and is used to configure the target editor, the transformation file and the name of the metamodel, on which the editor is based on.

Additionally, the configuration element contains the lists of transformations and menu contributions as well as optional elements for configuring custom command handlers and transformation frameworks.

Transformation

The transformation elements are the interconnection between the KSBasE framework and the transformations. They contain attributes for an identifier and the name of the target method in the given transformation file. Due to the limitations of the Eclipse extension point mechanism, the user has to ensure that the method name is valid. Additionally, the element contains an optional attribute for transformations that are used for visibility validation. Those transformations are executed every time a menu is about to be shown and check if the transformation will return a *true* value. This validation may be used to introduce advanced restrictions for certain transformations, e. g. if a user should not be allowed to add a successor state to the root element of a diagram.

MenuContribution

The menu contribution element is used to create the contribution to the user interface. It comprises attributes and elements for creating a set of menu contributions, based on the standard Eclipse UI contribution mechanisms. To allow a general configuration, which is often desirable, it is possible to create new menu contributions as well as adding structure-based editing operations to existing menus.

CommandHandler

The *commandHandler* is an optional element that allows the definition of a custom command handler, which extends the Eclipse *AbstractHandler* class and is called whenever a menu item generated by the KSBasE framework is activated. This may be used to change the way the transformations are initialized and executed.

TransformationFactory

To create a generic framework for structure-based editing, the *transformationFactory* element can be used to replace the default transformation framework with a custom implementation, which is based on the *ITransformationFramework*. This interfaces comprises methods for initializing a transformation, assigning the parameter values as well as for the execution and the handling of possible results that are returned by the transformation framework. A detailed interface description can be found in Appendix A.

5.3. KSBaSE UI Project

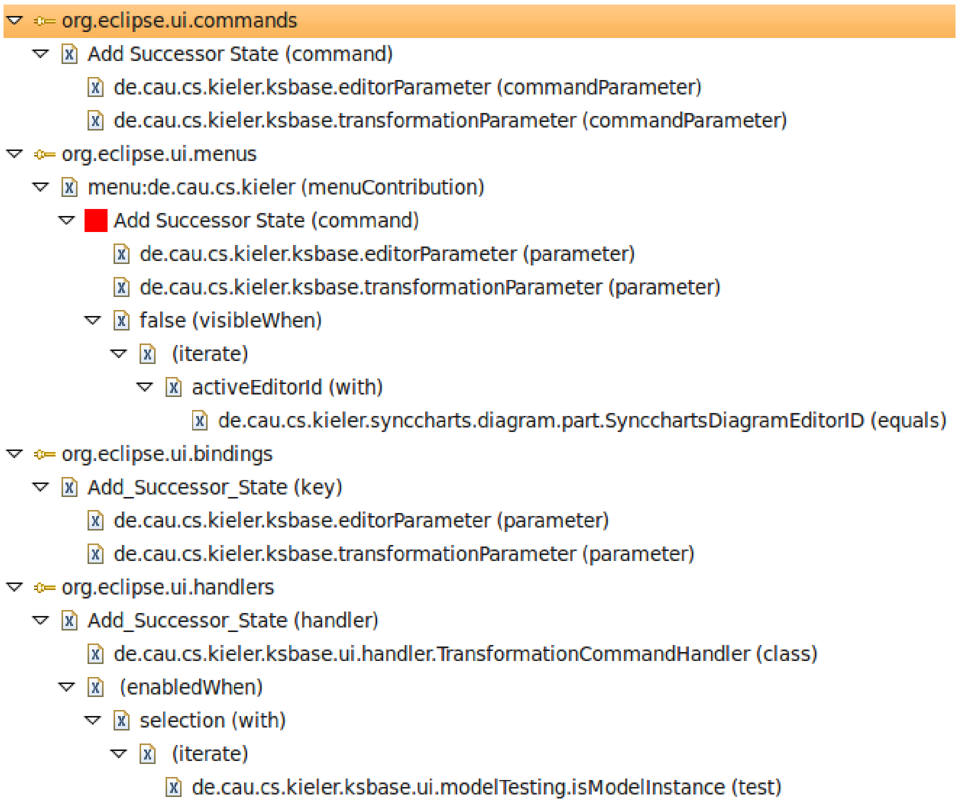
The KSBasE UI project comprises classes that are responsible for integrating the structure-based editing operations into the Eclipse platform. The integration includes creating menu contributions and keyboard bindings as well as the execution of commands, followed by the initiation of a refresh of the graphical contents from the semantic data. Furthermore, the UI project will notify the associated listeners before and after a transformation has been executed.

5.3.1. From KSBasE Extensions to Eclipse UI Contributions

The first task the UI project has to solve is to create menu contributions from the editor configurations, provided by the KSBasE core project. The simplest solution for this task is the requirement for an additional extension point, where developers have to create all contributions manually. Those manual configurations would have to use a fixed set of elements, e. g. command handlers, visibility expressions and classes. The major drawbacks of this solution are the large amount of elements a developer has to create and its inflexibility, i. e. if a transformation has been renamed, the name has to be changed manually in all existing contributions, e. g. main menus, tool bars and popup menus.

Figure 5.2 shows the necessary structures for a single transformation with one menu contribution. Adding an additional menu contribution, for example to create a toolbar icon, requires the creation of a new menu extension with almost the same content as in the existing *Add successor state* child of the *org.eclipse.ui.menus* extension.

To free developers from the unnecessary task of creating those extension manually, the KSBasE UI project comprises classes called *DynamicMenuContributions* and *Dy-*



```
org.eclipse.ui.commands
  Add Successor State (command)
    de.cau.cs.kieler.ksbase.editorParameter (commandParameter)
    de.cau.cs.kieler.ksbase.transformationParameter (commandParameter)
org.eclipse.ui.menus
  menu:de.cau.cs.kieler (menuContribution)
    Add Successor State (command)
      de.cau.cs.kieler.ksbase.editorParameter (parameter)
      de.cau.cs.kieler.ksbase.transformationParameter (parameter)
      false (visibleWhen)
        (iterate)
          activeEditorId (with)
            de.cau.cs.kieler.syncharts.diagram.part.SyncchartsDiagramEditorID (equals)
org.eclipse.ui.bindings
  Add_Successor_State (key)
    de.cau.cs.kieler.ksbase.editorParameter (parameter)
    de.cau.cs.kieler.ksbase.transformationParameter (parameter)
org.eclipse.ui.handlers
  Add_Successor_State (handler)
    de.cau.cs.kieler.ksbase.ui.handler.TransformationCommandHandler (class)
    (enabledWhen)
      selection (with)
        (iterate)
          de.cau.cs.kieler.ksbase.ui.modelTesting.isModelInstance (test)
```

Figure 5.2.: Manual creation of extensions

namicBundleLoader, to build the necessary structures automatically and inject them into the current Eclipse instance.

When the KSBasE UI plug-in is activated, the *DynamicMenuContributions* class will read the existing editor transformation settings, provided by the KSBasE core project, and create new Eclipse bundles, comprising the elements that are shown in Figure 5.2. When the bundles have been created, they will be inserted into a list of pending bundles in the *DynamicBundleLoader*, together with the ID of the corresponding editor. Whenever Eclipse opens a new editor window, the *DynamicBundleLoader* will check if there is a pending bundle for the editor and load it. This mechanism avoids unnecessary overload on activation of the KSBasE UI plug-in.

5.3.2. Listener Interfaces

To provide an extensible framework, as proposed in Chapter 4, which allows to define arbitrary pre- and postconditions for transformations, the KSBasE UI project contains a listener interface, called *ITransformationListener*. This interface comprises methods that are called right before and after a transformation has been executed. This is the recommended way of interacting with the KSBasE UI project and has been used

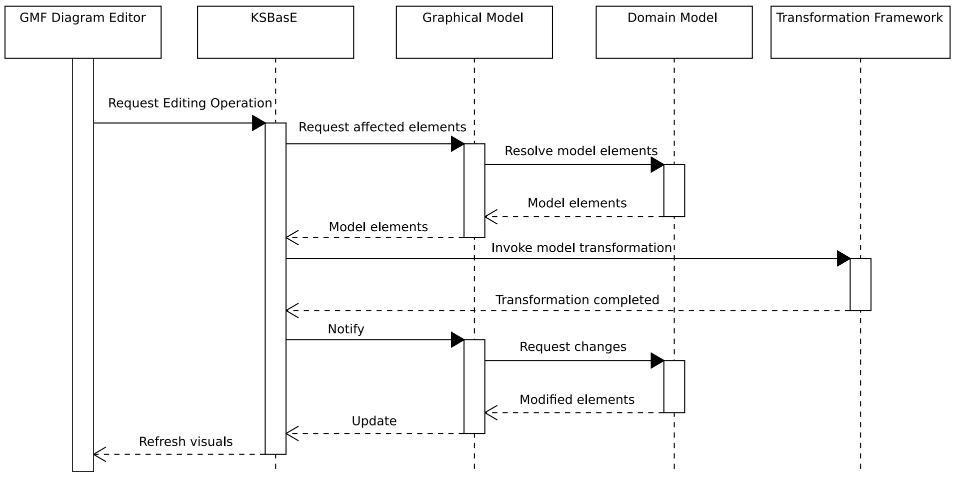


Figure 5.3.: Sequence of a structure-based editing operation

to integrate the KIELER view management and to create bridges to other projects, which may interfere with the execution of in-place model transformations.

5.3.3. Execution of Operations

For executing a structure-based editing operation in KSBasE, the GEF command pattern (see Figure 3.3) is used. This mechanism offers an easy way of connecting user interactions with the execution of arbitrary code. Figure 5.3 shows how a transformation command is executed and which objects are involved. For implementing the command mechanism, the following classes have been created:

ExecuteTransformationRequest

Whenever a user initiates a structure-based editing operation, GEF generates an *ExecuteTransformationRequest*, which contains the data that is necessary to execute a transformation, e. g. the name of the transformation method.

ExecuteTransformationEditPolicy

After the request has been created, GEF seeks edit policies that have been registered with the currently selected diagram element and are capable of fulfilling the generated request. This behavior is implemented in the *ExecuteTransformationEditPolicy*, which generates an appropriate transformation command.

ExecuteTransformationEditPolicyProvider

This class implements an *EditPolicyProvider* and is used to install an *ExecuteTransformationEditPolicy*, for each graphical edit part that has been created by GMF.

TransformationCommand

The *TransformationCommand* is used to actually start executing a transformation. The command will be passed to the GEF command stack and starts executing the correspondig in-place model transformation with the currently selected transformation framework.

5.3.4. Menu Visibility Handling

An important mechanism to provide transparent use of the KSBasE features in a GMF-based editor is the visibility handling for menu contributions. The aim of this component is that menu contributions should either be disabled or hidden, if the user selects diagram elements that are not valid for a structure-based editing operation.

Eclipse provides several options to hide or disable an element, e. g. by implementing a listener interface that is called whenever a menu is about to open or using a set of expressions in the extension point configuration.

Because of the dynamic bundle creation for editor configurations, the KSBasE UI plug-in is using the second approach by assigning a fixed set of expressions to the generated contributions. To follow the Eclipse guidelines, two different kinds of expressions have been added. The first one is assigned to the actual menu items and checks if the currently activated editor matches the target editor of the structure-based editing configuration. The second expression contains a *Test* element, which is used to call a Java class that implements an Eclipse *PropertyTester*.

The actual implementation of this *PropertyTester* is used to perform two different kinds of checks: the model-parameter match and the additional validation. As shown in Figure 5.4, the model-parameter match is used to determine if the currently selected diagram elements can be matched to the parameters of a given transformation, whereas the additional validation will execute the Xtend transformations that have been set in the KSBasE extension point configuration. These transformations can be used to introduce advanced constraints for a structure-based editing operation to prevent the execution of transformations for a subset of elements, e. g. root or simple states.

5.3.5. Preference Pages

In addition to the comparatively static approach of defining structure-based editing features with extension points, a second use case has been defined that aims at a more dynamic way of improving an editor by introducing a preference page for configuring KSBasE.

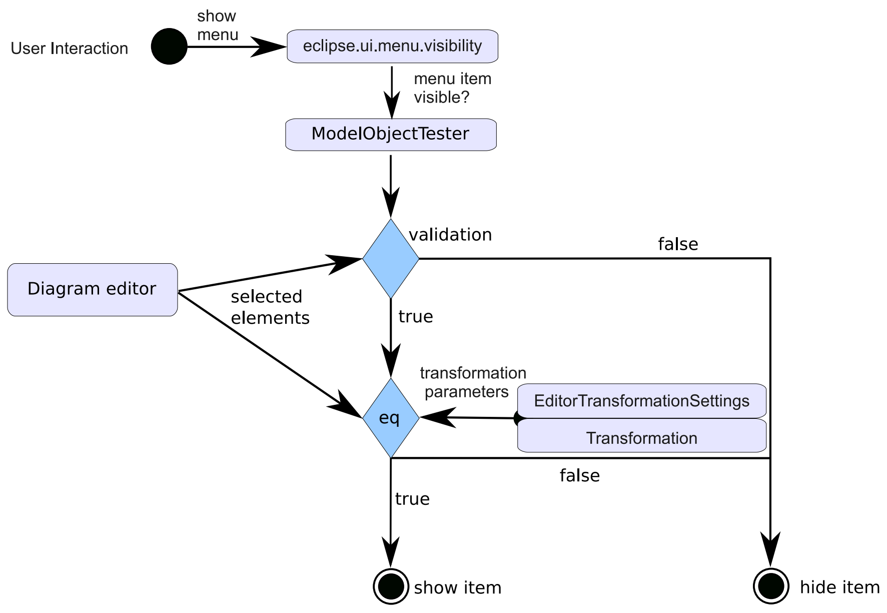


Figure 5.4.: UI visibility interaction

However, due to some restrictions of the Eclipse platform, the preference page configuration had to be limited to the addition of structure-based editing operations, with configurable names, to a predefined menu contribution, as shown in Figure 5.5. The editing operations are loaded into an editor by selecting an existing Xtend transformation file, which will be parsed automatically by the KSBasE framework.

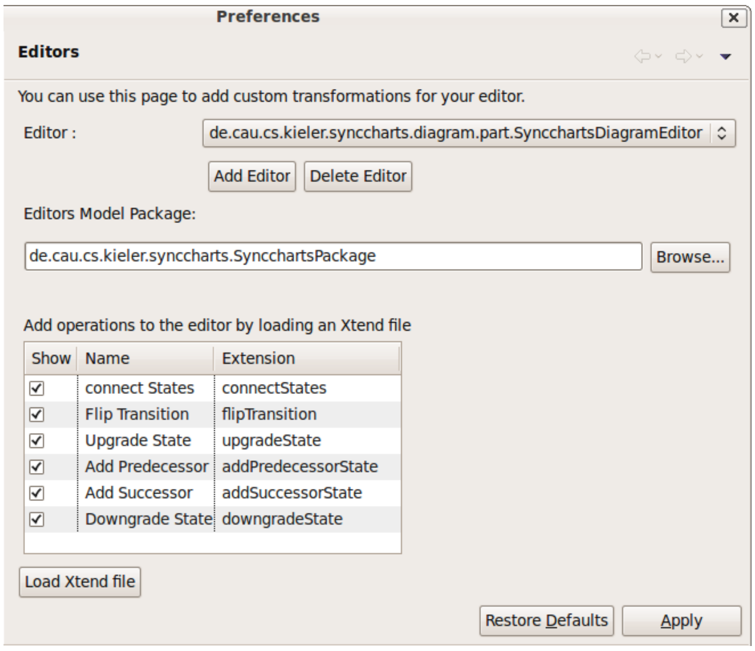
To contribute those editing operations to the Eclipse UI, the *dynamic* extension, from the *org.eclipse.ui.menus* extension point has been used. This extension allows developers to programmatically change the items that will be shown in a menu by extending the *ContributionItem* class.

However, because of some more restrictions of the Eclipse platform, the menu items, created by a *ContributionItem* class, cannot be assigned to keyboard short-cuts and will not be disabled if the currently selected diagram elements cannot be matched to the transformation parameters, as shown in Section 5.3.4. Nevertheless, the preference page configuration is an easy and fast way to test extensions for an editor without creating a new project and using the extension point mechanism.

5.4. Additional KIELER Interfaces

To unleash the full power of a structure-based editing framework it is necessary to integrate other features into the editing process. The most important features for this thesis are the automatic layout of diagrams and the use of a view management to perform additional actions when a transformation has been executed.

Both of these features have already been developed in the KIELER project, as



You can use this page to add custom transformations for your editor.

Editor : de.cau.cs.kieler.syncharts.diagram.part.SyncchartsDiagramEditor

Editors Model Package:

de.cau.cs.kieler.synccharts.SyncchartsPackage

Add operations to the editor by loading an Xtend file

Show	Name	Extension
✓	connect States	connectStates
✓	Flip Transition	flipTransition
✓	Upgrade State	upgradeState
✓	Add Predecessor	addPredecessorState
✓	Add Successor	addSuccessorState
✓	Downgrade State	downgradeState

Figure 5.5.: KSBasE Editors Preference Page

introduced in Chapter 3.2.

5.4.1. KSBasE View Management Integration

The view management project, introduced in Chapter 3.2.2, has been tightly integrated into the KSBasE framework to allow the execution of multiple effects after a transformation has been executed. Therefore a specialized plug-in has been added to the KSBasE project, which is solely responsible for interconnecting KSBasE and the view management.

Because of the flexible interface of the view management, it has also been used to integrate the KIML project into the KSBasE framework to apply an automatic layout after a transformation has been executed.

Figure 5.6 shows the interaction between KSBasE and the view management. The upper part visualizes the initialization phase where an activation event (which may be a user input or a source code invocation) causes the view management project to create all existing combination objects that have been defined by using the provided extension points. The KSBasE view management project contains the implementation of such a combination, which comprises the effects that have been activated by the user in the preference page. Furthermore, a KSBasE trigger has been created that also contains the implementation of an *ITransformationListener*. After the combinations have been created, the view management activates all existing trigger objects, which causes the KSBasE trigger to add itself to the list of transformation event listeners in

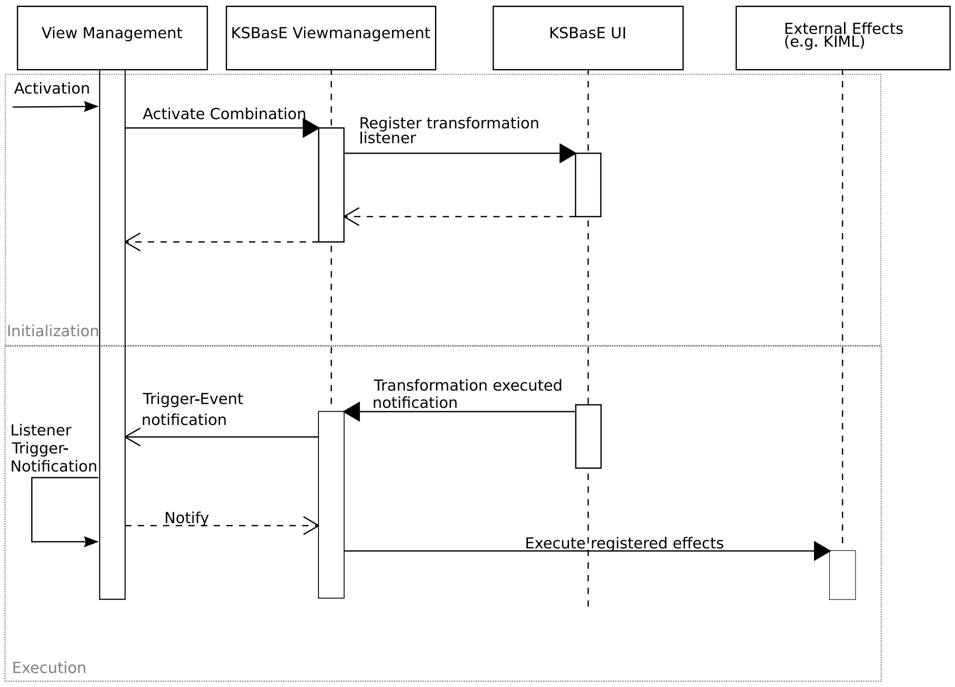

View Management
KSBasE Viewmanagement
KSBasE UI
External Effects (e.g. KIML)
Activation
Activate Combination
Register transformation listener
Initialization
Transformation executed notification
Trigger-Event notification
Listener Trigger-Notification
Notify
Execute registered effects
Execution

Figure 5.6.: Interaction of KSBasE and view management projects

the KSBasE UI plug-in.

In the execution phase, the KSBasE framework notifies the registered transformation event listeners every time a transformation has been executed. This causes the KSBasE trigger to notify its owning combination object, which will execute all its containing effects, e. g. the automatic layout of the current diagram.

Another important aspect when executing post-transformation effects is that the unordered execution may cause unsatisfying visual results, e.g. when a layout effect is combined with the execution of an effect that causes the diagram area to be zoomed to fit the diagram contents. Figures 5.7 and 5.8 visualize the problem that, when executing the zoom effect before the diagram layout has been updated, the inserted diagram contents may not be visible. In order to achieve the result shown in 5.8 reliably, a priority mechanism has been added to the KSBasE view management component. This mechanism allows users to select the sequence in which the effects are executed by ordering the list of elements in the preference page. This priority mechanism is based on the assumption that the view management components takes care of the ordered execution of effects, even though some of the effects are executed in separate threads.

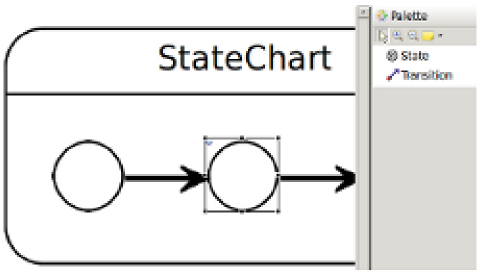


Figure 5.7.: Executing zoom before layout effect

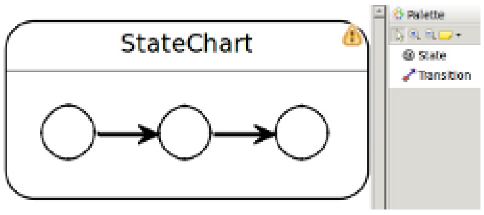


Figure 5.8.: Executing layout before zoom effect

5.5. Limitations of KSBasE

Due to the assumption that the graphical editing framework is constantly synchronising the entire semantic model with the graphical representation, the KSBasE project is currently limited to a subset of GMF editors, namely those who are using the GMF *CanoncialEditPolicy*, which is registered to the semantic model and listens for changes of the model content. Whenever the model changes, the *CanonicalEditPolicy* either creates a command that will create a new graphical object for added semantic elements or delete objects that are connected to non-existing semantic elements.

This synchronisation mechanism is a special interested for UML editors, where a semantic model may be used to create a variety of diagrams, e. g. class and sequence diagrams. Therefore most of the GMF UML editors, such as the Papyrus project[1], are built without a synchronisation mechanism, i. e. if a user is adding an interface element in a class diagram editor, it will not be shown in a sequence diagram unless it is manually inserted.

Solving this issue requires the development of a custom synchronisation mechanism that is used for editors that have been built without a *CanonicalEditPolicy*. A generic solution for arbitrary editors and model transformation frameworks involves a large amount of additional analyzing efforts and therefore is beyond the scope of this thesis.

[1] http://www.papyrusuml

5.6. Example Configurations

5.6.1. ThinKCharts Editor

As introduced in Chapter 3.2.3, the ThinKCharts editor has been developed for modeling SyncCharts [5] and is the main editor that is shipped with the KIELER project. Since SyncCharts can be used to model complicated systems, as shown in Figure 1.5, diagrams may become very large and complex, which will inevitably lead to time consuming navigation and layout tasks, even when performing simple editing operations as adding a successor to a state. This leads to the assumption that the use of structure-based editing operations will significantly improve the efficiency of editing SyncChart diagrams.

In the context of the KIEL project, Wischer [32] has investigated how structure-based editing can assist in the editing of SyncCharts.Because of the similar meta-models used by the KIEL and the ThinKCharts editor, some of the operations could be transferred into the KSBasE project; nevertheless, because of improvements in the SyncChart metamodel during the development of the ThinKCharts editor and the structural characteristics of editors created with GMF, the overall set of structure-based editing operations had to be adapted.

We will now have a look at the transformations that have been defined for the ThinKCharts editor. The entire set of transformations is shown in Appendix B.1.

State Operations

The following operations have been defined for editing states:

- Add successor states

  This operation may be used on an arbitrary number of states and will create one new state for each of the selected ones. Furthermore, it will create a transition from the existing states to the new.

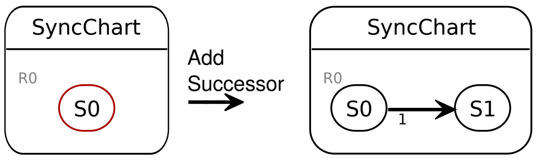


Figure 5.9.: Adding a successor state

- Add predecessor states

  This operation will also add new states to the diagram, but in contrast to the successor operation, the transitions will be drawn from the new states to the selected.

- Add choice

  This operation will add a new conditional state as a successor to the selected state. Additionally, it will create two subsequent states to the conditional with distinct priorities.

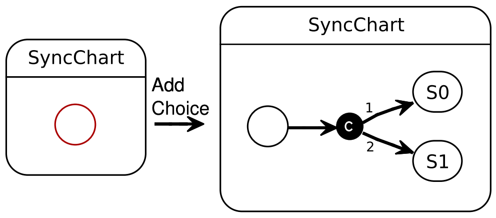


Figure 5.10.: Adding a choice construct

- Add region

  By using this operation on a state, the state will be transformed from a simple to a complex state, i. e. the state itself will be capable of containing states and regions. If this operation is performed on a state that is already a complex state, which means that is contains at least one inner region, a new, parallel region is added to the state.

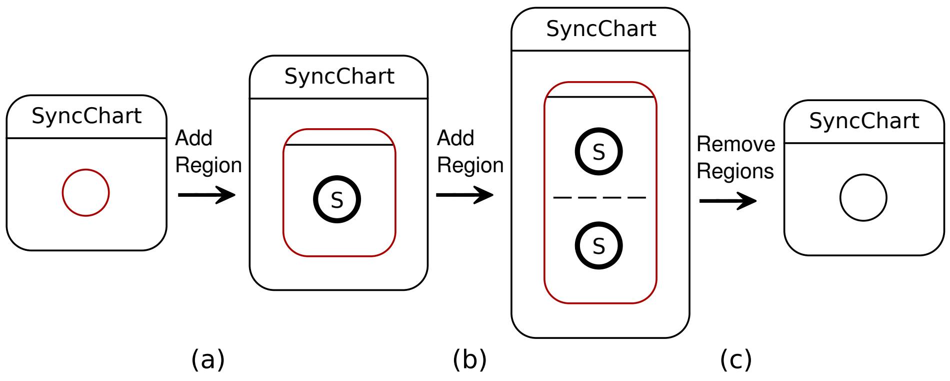


Figure 5.11.: Adding a region to a simple (a) and a complex (b) state. Followed by removal of all regions (c)

- Remove all regions

  This operation can only be used on complex states with at least one inner region. When executed on a state, this operation will remove all contained regions and their states and convert the complex state to a simple.

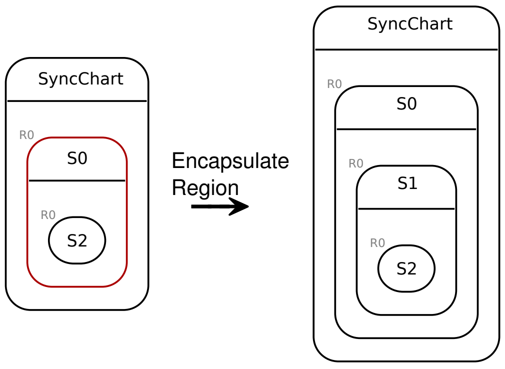


Figure 5.12.: Encapsulating a region

- Encapsulate region

  This operation is executed on a state and will add a new region to the state, followed by moving all the regions that are contained in the selected state to the new one.

- Extract state

  By executing this operation on a state, all its containing regions will be moved to the parent state and the selected state will be removed.

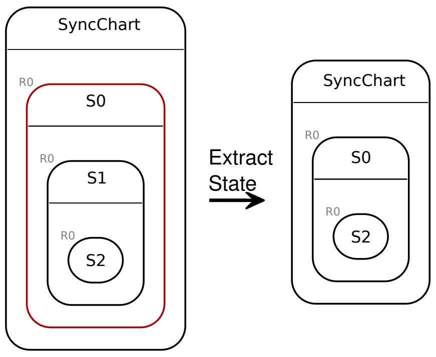


Figure 5.13.: Extracting the elements of a state

Transition Operations

The following operations have been defined for editing transitions:

- Connect states

  This operation is used on two states and will create a new transition between them. The source of the transition will be the state that has been selected first.

- Flip transition

  This operation will change the source and the target of the selected transition.

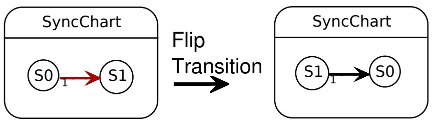


Figure 5.14.: Exchanging the source and target of a transition

- Set transition source

  This operation is used on a state and a transition and will set the source of the transition to the selected state.

- Set transition target

  This operation is very similar to the set transition source operation, but it will set the target of the selected transition to the selected state.

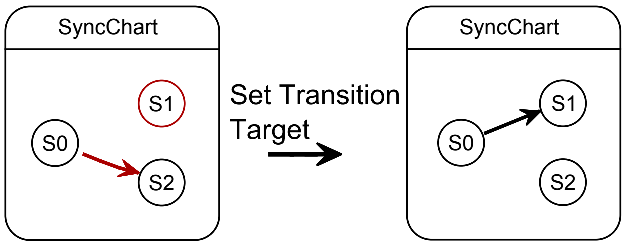


Figure 5.15.: Changing the target state of a transition

- Add Self Loop

  The add self loop operation can be used to add a self-transition to the selected state.

5.6.2. KIELER of Dataflow (KoData) Editor

As introduced in Chapter 3.2.4, the KoData editor provides a graphical editor for creating data flow diagrams. Although the editor is very rudimentary, it can be used to demonstrate how structure based editing may possibly be used for data flow diagrams. We will now have a look at some of the operations that have been created for the KoData editor. The entire set of transformations is shown in Appendix B.6

Data flow operations

- Add successors/predecessors

  The successor and predecessor operations are defined analogously to the operations in the ThinKCharts editor and are used to add successors and predecessors to the selected box.

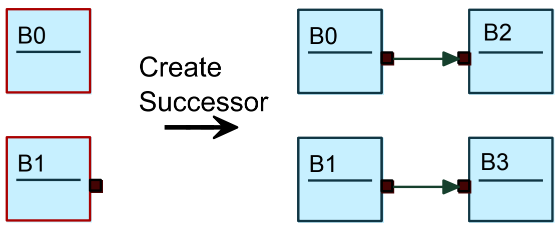


Figure 5.16.: Adding a successor to a box without ports (top) and with an existing output port (bottom)

- Connect boxes.

  The connect boxes operation is an important action for data flow diagrams and the results may differ, depending on the state of the selected boxes. If one of the boxes does not contain input or output ports, those ports are created and attached to the box. If there are more output ports on the source state than input ports on the target box, only the existing ports will be interconnected. Furthermore, it is very important for data flow diagrams in which order the ports are connected, because the boxes may represent an arbitrary operator whose output depends on the order of inputs. Common data flow modeling tools solve this problem by providing different approaches for connecting boxes, e. g. by using the name or the rank of ports; however, since the KoData editor does not provide an ordering mechanism for ports, they will be connected by their position in the semantic file, i. e. the order of the corresponding XML elements.

- Connect open ports

  The connect open ports operation is executed on two boxes and will connect all unconnected input and output ports. If the target box has less input ports than the output ports contained in the source box, only the available ports will be connected.

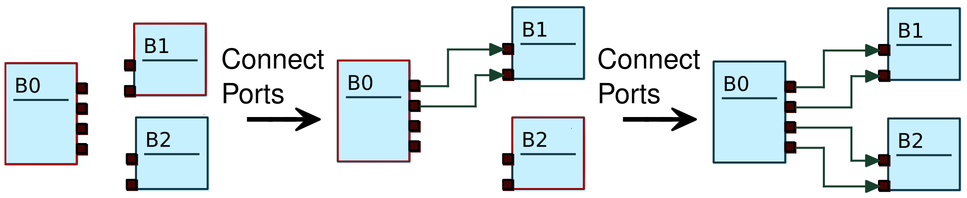


Figure 5.17.: Subsequent execution of connect ports operation

- Insert box

  The insert box operation is used on a connection between two boxes and will add a new box that contains an input and an output port, which are connected to the source and target boxes of the selected connection.

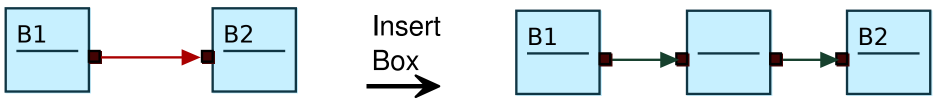


Figure 5.18.: Adding a box on a connection

5.6.3. UML

Even though there are technical problems when using the KSBasE project with UML editors, such as Papyrus, researches of how structure-based editing could be applied to UML editors have already been made [11]. We will now have a look at some of transformations that may be applied to UML diagrams.

Class Diagram Operations

- Extract interface

  This operation may be applied to classes and will create a new interface that contains all methods that are contained in the class as well as an *realization* relationship between them.

- Create implementation

  The create implementation operation is used on interfaces and will create a class, containing all interface methods, as well as an *realization* relationship.

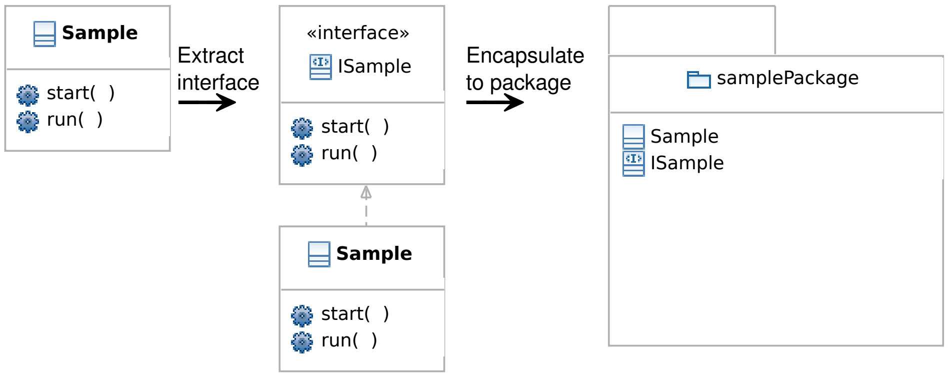


Figure 5.19.: Extracting an interface from a class and encapsulating the result to a package

- Encapsulate to package

  This operation may be performed on an arbitrary number of classes and interfaces and will create a new package that will comprise all selected elements.

Activity Diagram Operations

- Add successor activity

  In analogy to the SyncCharts and data flow diagrams, this operation is executed on an activity and will create a new activity and a control flow from the selected activity to the new one.

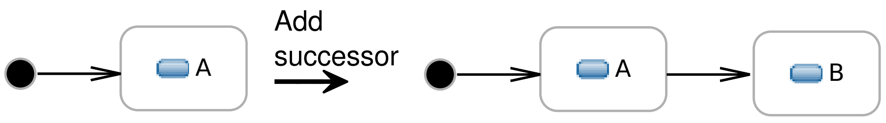


Figure 5.20.: Creating a successor activity

- Add decision

  This operation is also executed on an activity and will create a decision node as well as three case activities with the corresponding control flow connections.

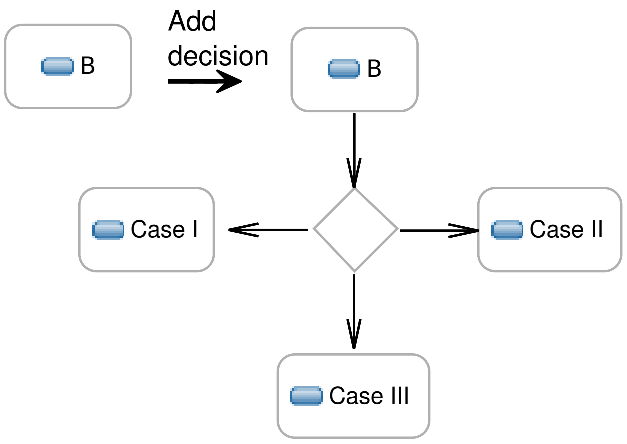


Figure 5.21.: Creating a decision node

5.7. Evaluation: Improvements in Editing Graphical Models with KSBasE

After having a detailed look at the architecture of KSBasE and some example operations, we will now have to evaluate if there is an improvement in editing graphical models with structure-based editing operations in comparison to DND editing.

First we will do a step-by-step comparison of creating of a small model in KSBasE versus DND editing. Afterwards, we will have a look at a statistical evaluation, which will visualize the gain of performance when using the KSBasE framework for graphical modeling.

5.7.1. Step-by-Step comparison

To compare both editing paradigms, we will create the ABRO SyncChart (see Figure 3.6) from scratch. Table 5.1 shows the necessary modeling steps for both editing types. Please note that the term *perform layout* for DND editing has been used as an placeholder and the actual meaning depends on the features that are used with the editor, e. g. an automatic layout tool or manual layout. In the latter case, the additional effort, described with *perform layout*, may include various manual editing operations for resizing and moving elements, as shown in Chapter 4.1. Furthermore, the use of an automatic layout tool may also introduce additional efforts, because most of the layout tools are trying to achieve a preferably compact layout. The comparison clearly shows how the use of structure-based editing reduces the number of actions, even though the model is very small. Creating the model with DND requires 37 subsequent editing steps, including 7 layout adjustments, whereas the structure-based editing involves only 18 steps.

Table 5.1.: Creating ABRO with DND and structure-based editing features.

	Drag and Drop	**KSBasE**
1	Drag and drop state	Execute 'create Default' operation
2	Set state label to 'ABRO'	Change state label from 'SyncChart' to 'ABRO'
3	Add signals A,B,R,O	Add signals A,B,R,O
4	Add region to the new state	
5	Add a state to the region	
6	Perform layout	
7	Set state flag *initial*	
8	Set state label to 'ABO'	Change state label from 'Initial' to 'ABO'
9	Add a region to the 'ABO' state	Execute 'add Region' on the 'ABO' state
10	Add another state to the region	
11	Perform layout	
12	Set state flag *initial*	
13	Set state label to 'WaitAB'	Change state label from 'S' to 'WaitAB'
14	Add a region to the 'WaitAB' state	Execute 'add Region' on the 'WaitAB' state
15	Add a state to the region	
16	Perform layout	
17	Set state flag *initial*	
18	Set state label to 'wA'	Change state label from 'S' to 'wA'
19	Add another state to the region	Execute 'create Successor' on the 'wA' state
20	Perform layout	
21	Drag and drop a transition and connect the states	
22	Set state flag to *final*	Set state flag to *final*
23	Add a parallel region to the 'WaitAB' state	Execute 'add Region' on the 'WaitAB' state
24	Add a state to the region	
25	Perform layout	
26	Set state flag *initial*	
27	Set state label to 'wB'	Change state label from 'S' to 'wB'
28	Add another state to the region	Execute 'create Successor' on the 'wB' state
29	Perform layout	
30	Drag and drop a transition and connect the states	
31	Set state flag to *final*	Set state flag to *final*

Continued on the next page

Table 5.1 - Continuation

	Drag and Drop	**KSBasE**
32	Add another state to the 'ABO' region	Execute 'create Successor' on the 'waitAB' state
33	Drag and drop a transition to connect the 'waitAB' state with the new one	
34	Set transition type to *normal termination*	Set transition type to *normal termination*
35	Perform layout	
36	Create a transition from 'ABO' to itself on signal 'R'	Execute 'add self loop' and set the trigger to 'R'
37	Set transition type to *strong abort*	Set transition type to *strong abort*

5.7.2. Statistical Evaluation

As it is not sufficient to compare only the necessary editing steps to achieve a meaningful statement about the suitability of a structure-based editing framework, we will now have a look at a statistical evaluation of the KSBasE project that has been performed with of a group of students.

Due to the interconnection of the KSBasE framework with the use of an automatic layout, three different stimuli had to be created to extract the actual gain in performance when using automatic layout only.

Stimuli and Equipment

The experiment has been performed under observation on prepared workstations. Each of those stations contained a set of three Eclipse instances, with different configurations: one with the default GMF DND features, one with the KIML auto-layout feature and the last one with the KSBasE feature and automatic layout after each of the operations. Furthermore, the subjects received a textual representation of the models, which have been created by using the KITS [7] language, as shown in Listings 5.1, 5.2 and 5.3. All of those models are designed to have an approximately equivalent complexity; nevertheless, due to the structure of the ThinKCharts editor and SyncCharts in general, some of the models are more difficult to create, e.g. model 1 emerged as the most complex model when created by using simple DND mechanisms, because of the level of hierarchy used in this model. To achieve meaningful results, e. g. to neutralize training effects, the order of models and features has been permuted so every model has been created with all of the three configurations.

Subjects

The subjects that participated in the evaluation can be separated into three different groups. The first group are students who participated in a class on synchronous languages. Most of those students are neither experienced in creating graphical models nor in using the ThinKCharts editor. The second group are students who took part in a practical course about modeling in Eclipse. The subjects from this group had used or seen the ThinKCharts editor for a couple of times and therefore had some experience before starting the experiment. The last group comprises members of the KIELER team. Those subjects are experienced in graphical modeling and in using the ThinKCharts editor.

Task

In the beginning of the experiment, the subjects received a short introduction to the ThinKCharts editor, the structure-based editing operations and the textual representation of the models.

Then, each subject has been assigned to a particular order of models and feature configurations. The procedure of the experiment has been the same for each of the subjects, regardless of the assigned configuration. The subject started by creating model 1 and measuring the time until all model elements had been created. The same procedure applies to model 2 and 3 with the corresponding feature configurations.

In addition to the existence of all elements, two requirements have been added to denote a modeling task as completed. First, all structural errors in the diagram editor must be removed. Some of the most common mistakes were missing state attributes or invalid transitions. The second requirement was a validation from the experiment administrator, where the model has been checked for completeness. Figures 5.28(b), 5.29(b) and 5.30(b) visualize the preferred results of the modeling process.

Due to the time constraints of the experiment, the requirement for a *good* layout, i. e. as compact as possible and avoidance of overlapping transitions, of the manually created diagrams has been dropped. This led to models which comprise all required elements but typically have an inappropriate layout, i. e. require a large amount of time to understand the structure of the model. Some examples of models with insufficient manual layout can be found in Figures 5.28(a), 5.29(a) and 5.30(a).

Results

The Figures 5.22, 5.23 and 5.24 visualize the modeling times for each of the three groups. The differences between the subjects can be ascribed to the diverse levels of training and slight differences between the complexity of the models. Summaries of the timing measurements for each of the three models are shown in Figure 5.25, 5.26 and 5.27.

Furthermore, reading the textual representation of the models and creating their actual graphical representation has been a complex process for some of the subjects, mostly for those who are very unfamiliar with StateCharts. Beside those individual differences, the experiment showed that structure-based editing significantly increases the effectivity of graphical modeling. As shown in Table 5.2, the average modeling times is nearly 48% lower when using structure-based editing in comparison to the editing with DND, even though the layout of the DND models has not been brought to perfection. The editing with automatic layout also decreased the necessary modeling times by nearly 33% compared to DND. The differences between modeling with KSBasE and auto-layout are greatly influenced by the subjects experience with the editor, i. e. a trained user, who may use a combination of keyboard shortcuts and the context menu, is usually faster than a user who is unfamiliar with the environment and therefore does not use keyboard shortcuts and needs to search for the menu contributions.

In addition to the objective timing measurements, most of the subjects emphatically preferred the structure-based editing over DND editing with or without automatic layout

	Group I	Group II	Group III	Average
DND	15:18	19:11	17:18	17:16
Auto-Layout	11:43	13:35	09:39	11:39
KSBasE	08:26	10:17	08:27	09:03

Table 5.2.: Average modeling times

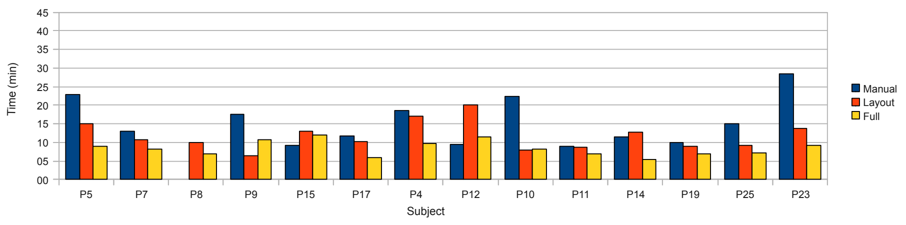


Figure 5.22.: Results for group I

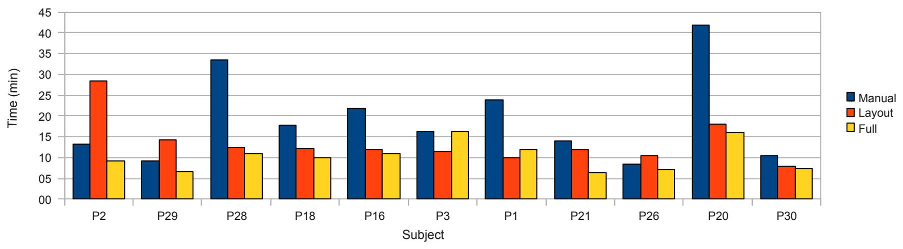


Figure 5.23.: Results for group II

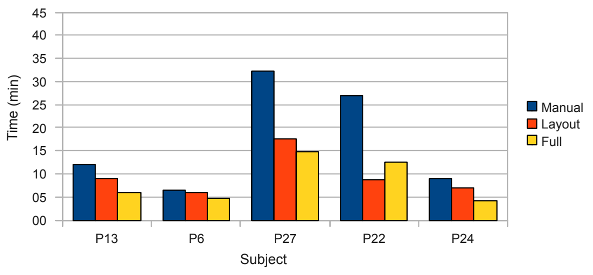


Figure 5.24.: Results for group III

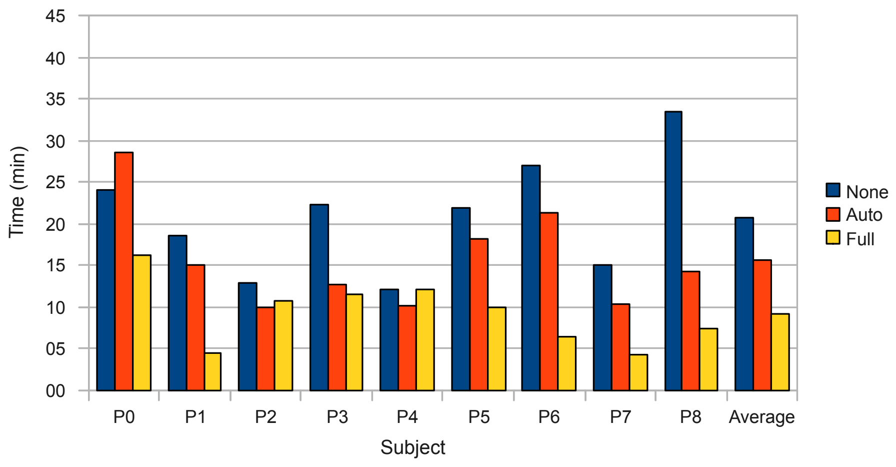


Figure 5.25.: Results for model I

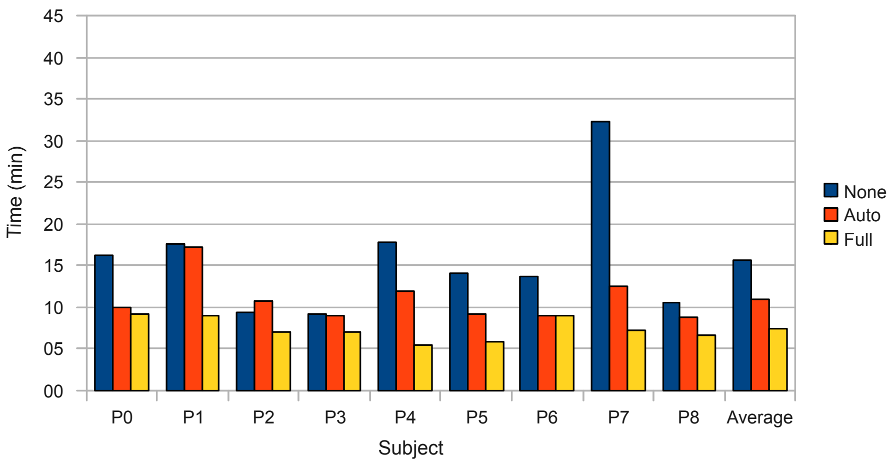


Figure 5.26.: Results for model II

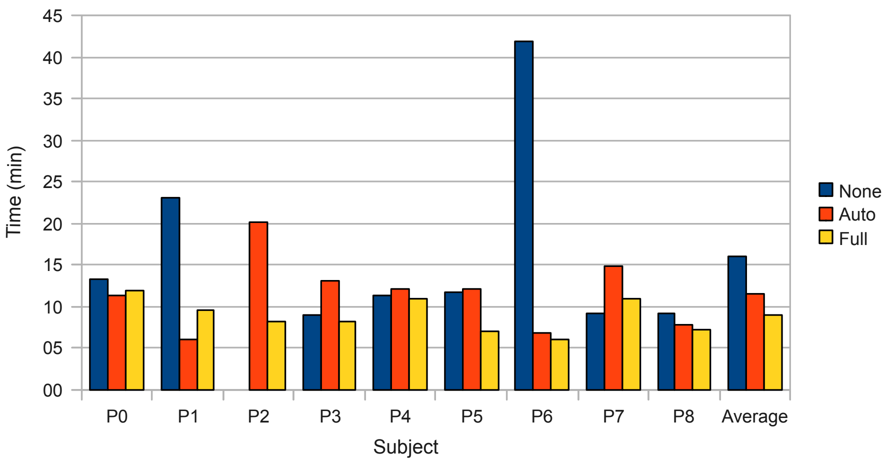
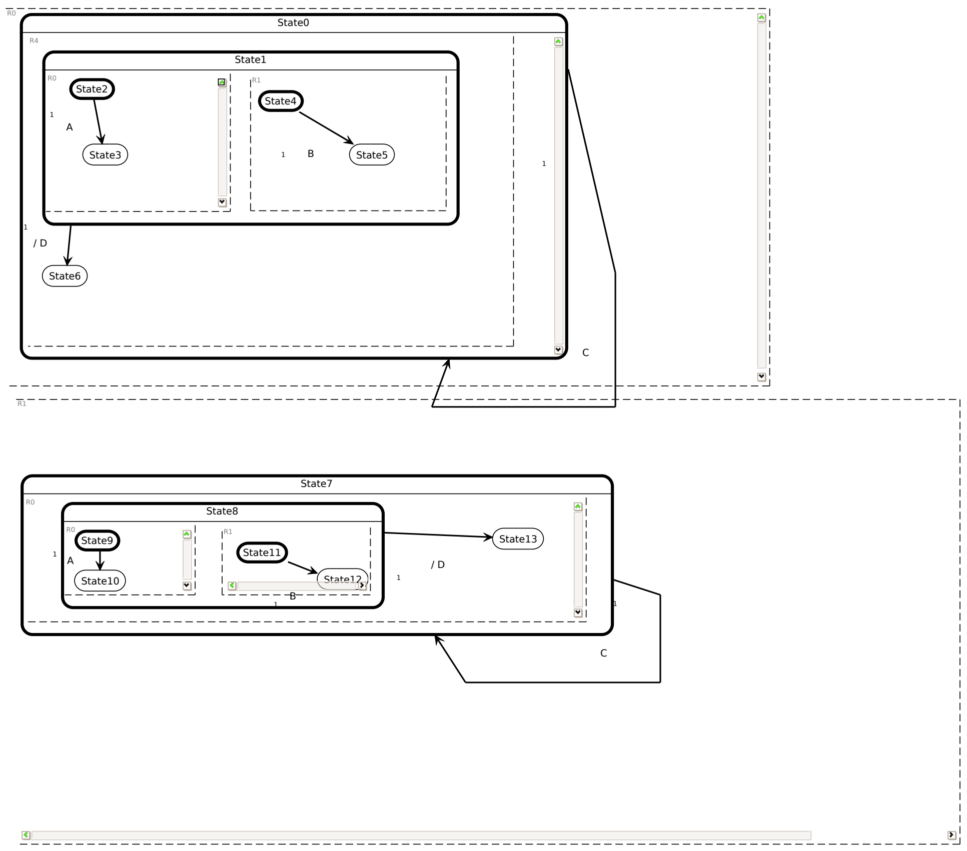
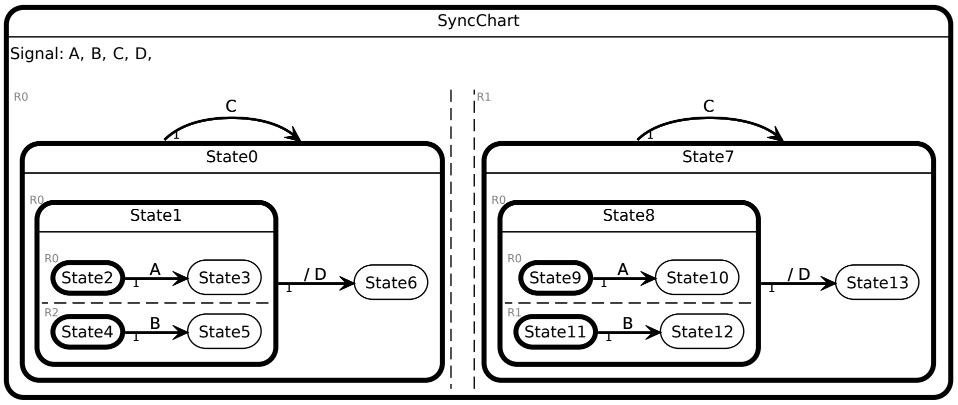


Figure 5.27.: Results for model III

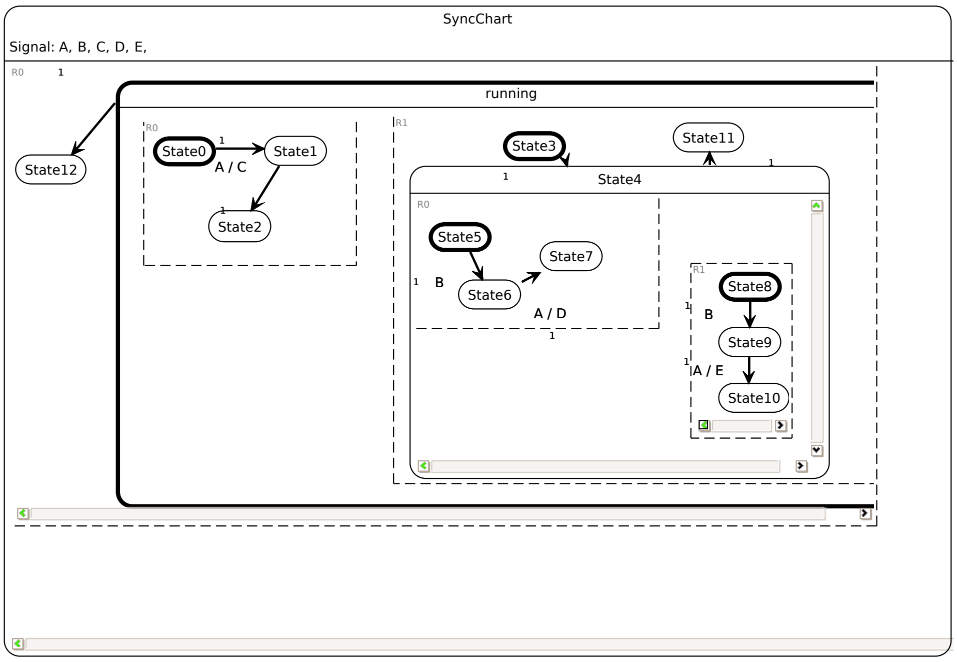


(a) Sample manual layout

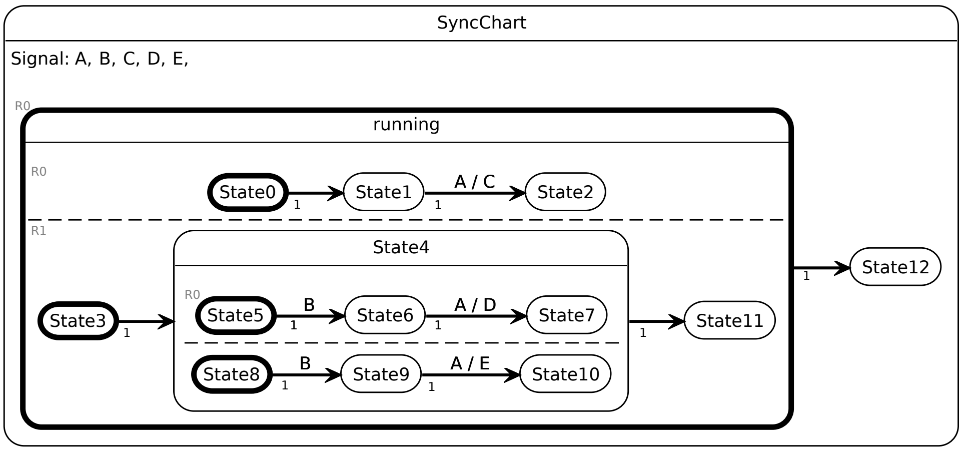


(b) Automatic layout

Figure 5.28.: Model 1

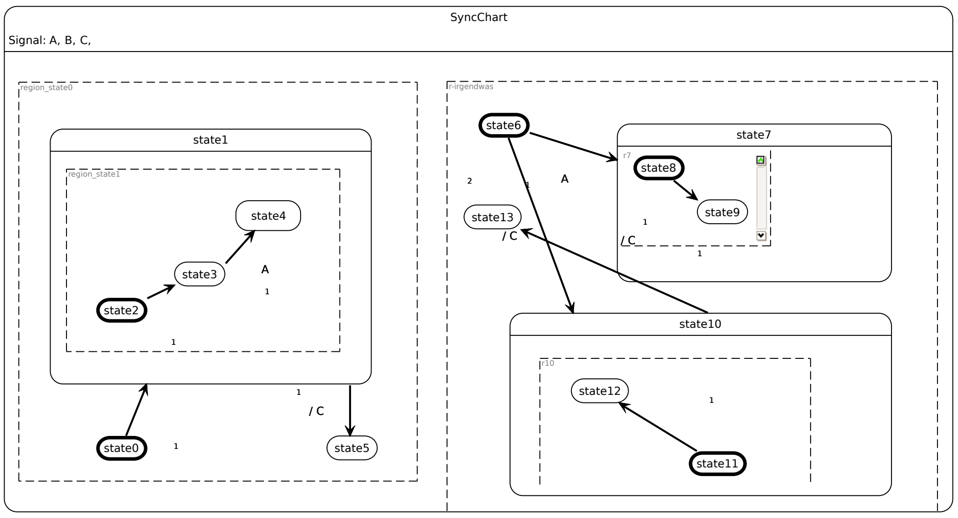


(a) Sample manual layout

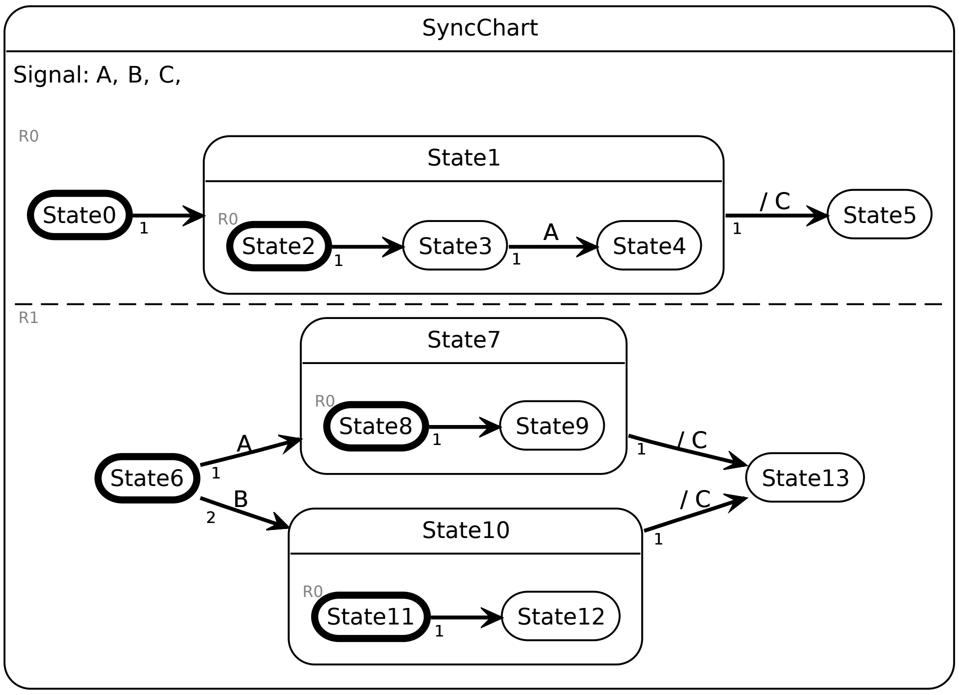


(b) Automatic layout

Figure 5.29.: Model 2



(a) Sample manual layout



(b) Automatic layout

Figure 5.30.: Model 3

Listing 5.1: Model 1

```
input A
input B
input C
output D
init Syncchart {
  init State0 {
    init State1 {
      init State2 --> State3 with A / ;
      state State3
      ||
      init State4 --> State5 with B / ;
      state State5
    } --> State6 with / D ;
    state State6
  } --> State0 with C / ;
  ||
  init State7 {
    init State8 {
      init State9 --> State10 with A / ;
      state State10
      ||
      init State11 --> State12 with B / ;
      state State12
    } --> State13 with / D ;
    State13
  } --> State7 with C / ;
}
```

Listing 5.2: Model 2

```
input A
input B
output C
output D
output E
init Syncchart  {
  init running {
    init State0 --> State1;
    state State1 --> State2 with A / C;
    state State2
    ||
    init State3 --> State4;
    state State4 {
      init State5 -->  State6 with B/;
      state State6 -->  State7 with A / D ;
      state State7
      ||
      init State8 --> State9 with B / ;
      state State9 --> State10 with A / E ;
      state State10
    } --> State11;
    state State11
  } --> State12;
  state State12
}
```

Listing 5.3: Model 3

```
input A
input B
output C

init SyncChart {
   init State0 --> State1;
   state State1 {
      init State2 --> State3;
      state State3 --> State4 with A;
      state State4
   } --> State5 with /C;
   state State5
   ||
   init State6 --> State7 with A;
   --> State10 with B;
   state State7 {
      init State8 --> State9;
      state State9
   } --> State13 with /C;
   state State10 {
      init State11 --> State12;
      state State12
   } --> State13 with /C;
   state State13

}
```

Chapter 6

Conclusion

"Thank you Mario! But our princess is in another castle!"

from: "Super Mario Bros"

This thesis aimed at enhancing the processes of creating and modifying graphical models by removing efforts that are introduced by common editing paradigms, such as drag-and-drop (DND) editing. This improvement is based on the paradigm of structure-based editing and uses the well-known technique of model transformation to create and execute operations defined on the abstract model of the model-under-development.

Besides the transformational approach and the proposal of a possible architecture for a structure-based editing framework, we introduced the KSBasE project as a reference implementation of the given approach. KSBasE comprises all tools and technologies that are necessary to enhance nearly arbitrary graphical editors that are built with GMF.

The evaluation of the KSBasE framework demonstrated that structure-based editing is a very viable alternative to common graphical editing paradigms, especially when combined with a sophisticated automatic layout.

Creating the model transformations and the structure-based editing configuration may introduce additional efforts, but in most cases those efforts are kept within reasonable limits and the gain of performance when actually using the structure-based editing operations exceeds the one-time efforts of creating them.

6.1. Future Work

During the development of the KSBasE framework several ideas for future works have emerged.

Creating structure-based editing operations for other editors

At the present time, the KSBasE framework has mainly been used for enhancing the KIELER ThinkCharts editor. A research on which structure-based editing operations may be defined for other common model spaces could be done. A special interest are UML editors, because of their widespread use. An initial approach on the enhancement of editing UML diagrams has been proposed by Fuhrmann, Spönemann, Matzen and von Hanxleden [11]. Furthermore, participants of the practical course on Eclipse modeling at the real-time and embedded systems group at the Christian-Albrechts-University of Kiel[1] started integrating concepts of the KIELER framework into the Papyrus UML[2] project.

However, as described in Chapter 5.5, there are still technical difficulties when using KSBasE with UML editors, such as Papyrus, which require the development of a custom synchronisation mechanism for this class of graphical editors.

Other uses for the KSBasE framework

In addition to the common use of structure-based editing, the technical base of the KSBasE framework may possibly be used for other parts of the graphical editing process. For example, one could create a library mechanism for data flow languages by defining model transformations that are used to create common data flow objects, e. g. logical or mathematical operators. Another idead would be use model transformations to define a set of optimizations for a diagram type; e. g. a SyncChart diagram may be optimized before a code generator, such as SC, creates an executable from the model.

[1] http://rtsys.informatik.uni-kiel.de/trac/09ws-eclipse/wiki/Projects/Papyrus

[2] http://www.papyrusuml

Appendix A

Documentation

Structure-based editing feature

Identifier: de.cau.cs.kieler.ksbase.configuration

Since: 0.1

Description: Extension point for configuring KIELER structure-based editing features for an editor.

Configuration Markup:

```
<!ELEMENT extension (configuration)+>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED
>
```

```
<!ELEMENT configuration (menus , transformations , commandHandler? ,
transformationFactory?)>
<!ATTLIST configuration
  editorId           IDREF #IMPLIED
  TransformationFile CDATA #IMPLIED
  packageName        CDATA #IMPLIED
  contextId          IDREF #IMPLIED
  defautlIcon        CDATA #IMPLIED
  XtendFile          CDATA #IMPLIED
>
```

Configures structure based editing features for an editor

- **editorId** -
- **TransformationFile** - The transformation file.
- **packageName** - The name of the main package that represents the underlying meta model. This class has to extend 'emf.ecore.EPackage'
- **contextId** - The contextID which is used to assign keyboard shortcuts. If this property is empty, the default context is used.
- **defautlIcon** - If you want to add Toolbar contributions, you can select an icon which will be used by default for all items
- *Deprecated* **XtendFile** - Please use the 'TransformationFile' Attribute. The Attribute 'XtendFile' will be removed in future versions.

```
<!ELEMENT transformation EMPTY>
<!ATTLIST transformation
  transformationId CDATA #REQUIRED
  name             CDATA #REQUIRED
  transformation   CDATA #REQUIRED
  keyboardShortcut CDATA #IMPLIED
  icon             CDATA #IMPLIED
  validation       CDATA #IMPLIED
>
```

The transformation properties.

- **transformationId** - The id of this transformation
- **name** - The name of the transformation which is used in the menus
- **transformation** - The name of the transformation method (without parameters). This method has to exist in the transformation file else it will be ignored
- **keyboardShortcut** - The keyboard shortcut for this transformation
- **icon** - Select an ico or png file, if you want to use an icon for the toolbar
- **validation** - An optional transformation that may be used to add additional validations for menu visibility and enablement. Make sure that the transformation is short and fast, it is not recommended to call Java code for validation!

```
<!ELEMENT menuContribution (transformationCommand*)+>
<!ATTLIST menuContribution
  locationURI CDATA #REQUIRED
>
```

- **locationURI** - The locationURI of this contribution

```
<!ELEMENT transformationCommand EMPTY>
<!ATTLIST transformationCommand
  transformationId IDREF #IMPLIED
>
```

Used to add a transformation command to a menu

- **transformationId** - The id of the referenced transformation

```
<!ELEMENT menu (transformationCommand+)+>
<!ATTLIST menu
  id    CDATA #REQUIRED
  label CDATA #IMPLIED
>
```

- **id** - The id of this menu
- **label** -

```
<!ELEMENT menus (menuContribution+)+>
```

```
<!ELEMENT transformations (transformation)+>
```

```
<!ELEMENT commandHandler EMPTY>
<!ATTLIST commandHandler
  class CDATA #REQUIRED
>
```

Sets the command handler that will be called when a KSBasE menu contribution is activated. If this field is not set, the default handler de.cau.cs.kieler.ksbase.ui.handler.TransformationCommandHandler is used. The command handler will receive two parameters representing the active editor and the transformation that has been activated.

- **class** - Sets the command handler that will be called when a KSBasE menu contribution is activated. If this field is not set, the default handler de.cau.cs.kieler.ksbase.ui.handler.TransformationCommandHandler is used. The command handler will receive two parameters representing the active editor and the transformation that has been activated.

```
  <!ELEMENT transformationFactory EMPTY>
  <!ATTLIST transformationFactory
    class CDATA #REQUIRED
>
```

- **class** - Used to add an alternate transformation framework for this editor. The given class needs to implement the de.cau.cs.kieler.core.model.transformation.ITransformationFramework interface.

Examples:

The `plugin.xml` file in the `de.cau.cs.kieler.synccharts.ksbase` plugin makes extensive use of the `de.cau.cs.kieler.ksbase` extension point.

API Information:

By using this extension point, the `de.cau.cs.kieler.ksbase` and `de.cau.cs.kieler.ksbase.ui` will use this configuration to create `org.eclipse.ui.menus` extensions for each menu transformation command. Furthermore, a set of visibility and enablement expressions will be added to ensure the menus are visible in the given editor only.

In the current version of the `de.cau.cs.kieler.ksbase` plug-ins the transformations are executed with the Xtend facade by default. The mapping of parameters to diagram objects is done automatically.

de.cau.cs.kieler.core.model.transformation

Interface ITransformationFramework

All Known Implementing Classes:

XtendTransformationFramework

`public interface` **`ITransformationFramework`**

Interface for creating a bridge between a transformation framework and KIELER. Every transformation framework that is used by the KSBasE Plug-in has to implement this interface. The `XtendTransformationFramework` class contains an implementation for the Xtend framework.

Author:

mim

Method Summary	
`java.lang.Object`	**`executeTransformation`**`()` Executes a transformation with the parameters set with the initalizeTransformation method.
`java.lang.String`	**`getFileExtension`**`()` Returns the default file extension for this framework without the leading dot.
`boolean`	**`initializeTransformation`**`(java.lang.String fileName, java.lang.String operation, java.lang.String... basePackages)` Initializes a transformation.
`java.util.List<AbstractTransformation>`	**`parseInPlaceTransformations`**`(java.net.URL fileName)` Parses a transformation file and returns the existing in-place transformations.
`void`	**`setParameters`**`(java.lang.Object... parameter)` Sets the transformation parameters.
`boolean`	**`setParameters`**`(java.lang.String... parameter)` Sets the transformation parameters by

	matching the current selection with the given list of types.

Method Detail

getFileExtension

```
java.lang.String getFileExtension()
```

Returns the default file extension for this framework without the leading dot.

Returns:
A string representing the file extension

---

executeTransformation

```
java.lang.Object executeTransformation()
```

Executes a transformation with the parameters set with the initalizeTransformation method.

Returns:
A return value from the transformation. May be null

---

setParameters

```
boolean setParameters(java.lang.String... parameter)
```

Sets the transformation parameters by matching the current selection with the given list of types. The framework may return 'false' if the parameters could not be matched.

Parameters:
`parameter` - The list of parameter types.
Returns:
True if all parameters could be set.

---

setParameters

```
void setParameters(java.lang.Object... parameter)
```

Sets the transformation parameters.

Parameters:
`parameter` - The actual parameters

---

initializeTransformation

```
boolean initializeTransformation(java.lang.String fileName,
                                 java.lang.String operation,
                                 java.lang.String... basePackages)
```

Initializes a transformation. This includes the parameter mapping, if necessary. The parameter 'parameter' is only a string representation of the parameter types of the given operation.

Parameters:
`fileName` - The transformation file name
`operation` - The operation to execute
`basePackages` - The class name of the editors EPackage

Returns:
False if an error occurred.

---

parseInPlaceTransformations

```
java.util.List<AbstractTransformation> parseInPlaceTransformations(java.net.URL fileName)
```

Parses a transformation file and returns the existing in-place transformations.

Parameters:
`fileName` - a URL to the transformation file

Returns:
a list of abstract transformations.

KSBasE - KIELER Structure Based Editing

K'SBASE

KIELER STRUCTURE BASED EDITING

- Responsible: Michael Matzen

Topics

The KSBasE ***[Kay-space]*** project enables developers to add structure based features to an EMF-based editor. The term *structure based* refers to the fact that defined features are based on the editors EMF meta-model.

Attention: The KSBasE project is in development and testing phase, so names, files, projects and workflows may still change. Be sure to use the latest repository versions only!

Java Projects

The following Java projects belong to this project:

- de.cau.cs.kieler.ksbase - Base classes of KSBasE feature, including extension points
- de.cau.cs.kieler.ksbase.ui - UI contributions, e.g. a preference page (found under Preferences->KIELER->KSBasE)
- de.cau.cs.kieler.ksbase.feature - Feature file with the latest KSBasE-Editor
- de.cau.cs.kieler.ksbase.viewmanagement - Interface to KIELER Viewmanagement (found under Preferences->KIELER->KSBasE->Post-Transformation Settings)

Example Projects

- de.cau.cs.kieler.synccharts.ksbase - Syncharts example project
- de.cau.cs.kieler.dataflow.ksbase - DataFlow example project
- de.cau.cs.kieler.uml2tools.ksbase - UML2 example project

Requirements

- Eclipse Galileo
- Eclipse Modeling Tools (EMF, GMF, Xtend, Xpand, Xtext)

Install

- The KSBasE project is now available in the KIELER RCP (http://rtsys.informatik.uni-kiel.de/~kieler/files/nightly/) and the KIELER update site (http://rtsys.informatik.uni-kiel.de/~kieler/updatesite/nightly/).
- Or simply check out the KSBasE projects and add it to your workspace.
- If you want to use the KSBasE UI / Viewmanagement project, you will also need the KIML and the Viewmanagement projects. Be sure to check out all sub projects too.

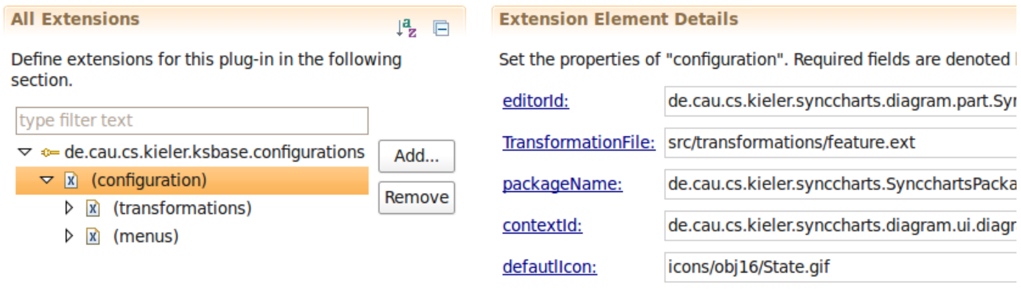
How it works

Creating structure based features using extensions

All examples shown here are using the Xtend transformation framework, shipped with KIELER. If you want to use a different framework, you need to implement the ITransformationFramework

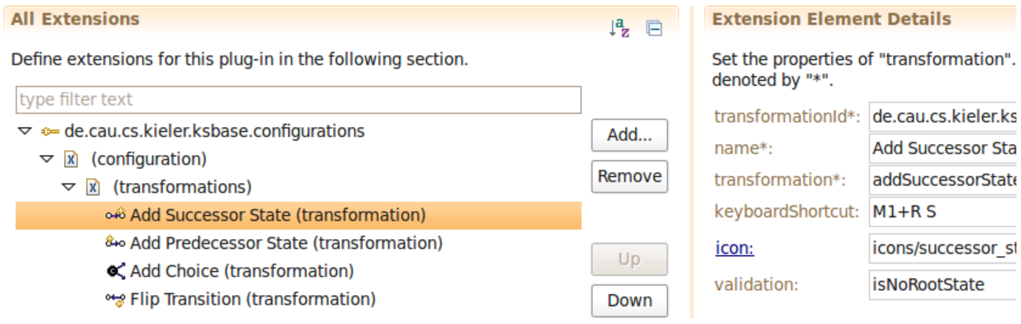
It is recommend to create a new plug-in project for adding structure based editing features to your editor!
To add fancy new features to a diagram editor you can use the eclipse extension point mechanism:

- First thing you might want to do is: Write the transformation file (see the examples below).
- Now to be able to use the extensions you need to add the **de.cau.cs.kieler.ksbase** project to your project dependencies (in the plugin.xml)
- If you are using a model file which is located in another project, you have to add this project to the dependencies, too.
- Last but not least: to select a diagram editor, you have to add the diagram project of the target editor to the dependencies (look for a project called *youreditor.diagram*)
- You can now create extensions by opening the tab 'Extensions' and click the 'Add' button.
- Select **de.cau.cs.kieler.ksbase.configuration** from the extension points list.
- Now add a configuration by right clicking the new entry and selecting 'New' -> 'configuration':


All Extensions
Define extensions for this plug-in in the following section.
type filter text
de.cau.cs.kieler.ksbase.configurations
(configuration)
(transformations)
(menus)
Add...
Remove
Extension Element Details
Set the properties of "configuration". Required fields are denoted

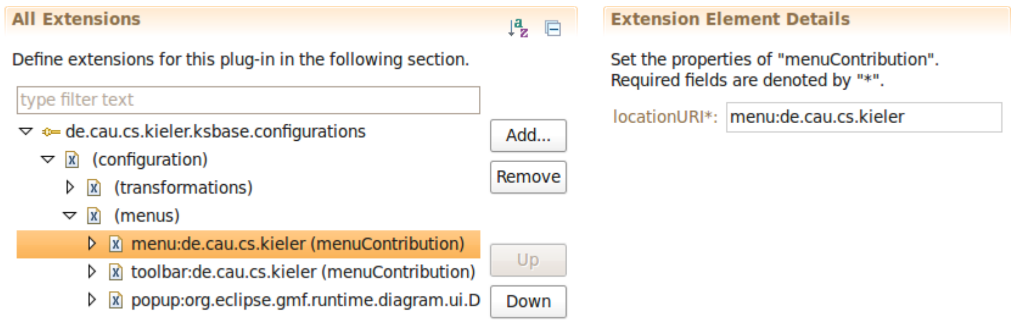
editorId:	de.cau.cs.kieler.syncharts.diagram.part.Syı
TransformationFile:	src/transformations/feature.ext
packageName:	de.cau.cs.kieler.syncharts.SyncchartsPacka
contextId:	de.cau.cs.kieler.syncharts.diagram.ui.diagr
defaultIcon:	icons/obj16/State.gif

- When you open the new entry, you will see the details page. Please set all properties and note the tool tip informations.
- To define transformations, you have to add some more elements to the configuration. To do this click on 'configuration' and select 'New' -> 'transformation'.
- Again, you will have to set all the properties in the detail page. Please note that the property 'transformation' needs to be the **exact name** of the method in the transformation file you selected in the configuration element.
- If you want to add icons to your commands, be sure to copy them to the current project.
- If the transformation is only valid for a subset of the parameters, e.g. a root element may not have a successor, you can insert additional validations in the *validation* attribute, you can enter multiple methods by separating them with commas. Note that those transformations **need to return a boolean value** that is used by the KSBasE framework to disable or hide the corresponding UI contributions.


All Extensions
Define extensions for this plug-in in the following section.
type filter text
de.cau.cs.kieler.ksbase.configurations
(configuration)
(transformations)
Add Successor State (transformation)
Add Predecessor State (transformation)
Add Choice (transformation)
Flip Transition (transformation)
Add...
Remove
Up
Down
Extension Element Details
Set the properties of "transformation". denoted by "*".

transformationId*:	de.cau.cs.kieler.ks
name*:	Add Successor Sta
transformation*:	addSuccessorState
keyboardShortcut:	M1+R S
icon:	icons/successor_st
validation:	isNoRootState

- Last but not least, you have to configure where the transformations should appear in your editor, so right click on the new entry and select 'New' -> 'menuContribution'.
- If you wish to add your commands to an existing menu, you can enter an existing 'locationURI' in the menuContribution.
- Additionally, you can use the special URI 'popupBar' to create GMF PopupBar contributions (**experimental**).
- To assign a transformation to the menu contribution, you need to add a 'transformationCommand' and set the id attribute to the corresponding transformationId.
- When you are done defining all transformations and menus, you are ready to use the extended diagram editor by launching a new eclipse instance with the necessary plug-ins.


All Extensions

Define extensions for this plug-in in the following section.

type filter text

- de.cau.cs.kieler.ksbase.configurations
  - (configuration)
    - (transformations)
    - (menus)
      - menu:de.cau.cs.kieler (menuContribution)
      - toolbar:de.cau.cs.kieler (menuContribution)
      - popup:org.eclipse.gmf.runtime.diagram.ui.D

Extension Element Details

Set the properties of "menuContribution".
Required fields are denoted by "*".

locationURI*: menu:de.cau.cs.kieler

Configuring the post-transformation actions

If the view management plug-in is installed, you can configure which effects should be applied when a transformation has been executed. Screenshots and more explanations will follow soon.

Creating transformations

We are now creating some nice and simple features for the Thin Kieler SyncCharts Editor:

If you'd like to see some more complex examples, you can have a look at the DataFlow editor Xtend file

The first step for extending an editor is to create the model2model transformations. For the KSBasE features, those transformations are defined using Xtend.
To create the transformations, we are using a new package called *de.cau.cs.kieler.synccharts.transformations* and create a file called *feature.ext*.
Attention: You can use any package name, but the package **must be included in the build path** or else the Xtend code completion will not work.
Now we can start creating the transformation file. For now we will only create 2 transformations:

- Add a new state to an existing
- Flip the source and target of a transition

(If you'd like to see all transformations currently defined for the SyncCharts Editor, you can have a look at the repository file)

The implementation of these transformations is easy:

```
import synccharts; //First import the synccharts metamodel

//Connects two states
Void connectStates(State source, State target):
let transition = new Transition:                 //Create new transition
transition.setSourceState(source) ->             //Set source
transition.setTargetState(target) ->             //and target state
setSelection(transition)                         //Select the transition.
;

//Adds a successor to the given state
Void addSuccessorState(State source):
 let target = new State:                               //Create a new target state
 connectStates(source, target) ->                      //Call the connectStates extension
 source.parentRegion.innerStates.add(target) ->        //Add the new state
 setSelection(target)                                  //Select the new state
;

//Creates a default SyncChart
Void createDefault(Region rootRegion):
let state = new State:                            //Create a new root state
let innerState = new State:                       //Create a new inner state
let region = new Region:                          //Create a new region for the root state
state.setLabel("SyncChart") ->                    //Set name of the state
state.regions.add(region) ->                      //Add region to the state
innerState.setLabel("Initial") ->                 //Set label of the inner state
innerState.setIsInitial(true) ->                  //The state type to initial
region.innerStates.add(innerState) ->             //Add inner state
rootRegion.innerStates.add(state) ->              //Add root state
setSelection(innerState)                          //Select inner state
;
```

A few hints on creating Xtend in-place model-to-model transformations:

- You always have to use *Void* as return value, since everything else will define non in-place transformations
- You can define local variables using the *let* keyword (no other type assignments are allowed!)
- Because Xtend is a declarative language, consecutive commands are separated using the -> operator

FAQ

- **The transformation seems to be executed but the diagram does not change**:
  - Thats an annoying GMF bug, there are two things you can do :
    - Open you diagrams plugin.xml, search the extension 'org.eclipse.gmf.runtime.diagram.core.viewProviers" and remove the contents from the 'semanticHint' property from all children.
    - If you don't want to modify your diagram plugin, you can insert an element to the diagram, e.g. by dragging it from the toolbar. Afterwards it will work until eclipse has been restarted.
  - The bug is already documented and will be fixed (https://bugs.eclipse.org/bugs/show_bug.cgi?id=261188)
- **The syntax highlighting and code completion does not work when editing the .ext file!**
  - Did you forgot to add the 'Xtend nature' to your project? Right click on the project and select 'Configure -> Add Xpand/Xtend Nature'
- **I've added the Xtend nature, but the code completion is not working!**
  - Did you put the Xtend file in a non-source folder? Either right click the folder that contains the Xtend file and select 'Build path -> Use as source folder, or copy the file to a folder which already is a source folder.
- **Ok, now the code completion works, but Xtend does not recognize my meta model**
  - You have to add the project that contains the meta model ecore file to your project dependencies and you need to import the meta model by using the import statement with the exact name of the root element of your model (e.g. *import synccharts;* with synccharts.ecore)
  - Please remember that it's not possible to import meta models by their namespace URI !

The good, the bad and the ugly (ToDo/Bug list)

- There are a whole lot of 'undefined context' exceptions when activating the plug-in. This is caused by eclipse when activating a plug-in during runtime with the extension registry and can only be fixed if the jar file, which is created by KSBasE, is imported into the workspace.
- The editor preference page has been deactivated, but will be back someday with a fixed menu structure.

Appendix B

Transformations

Listing B.1: KSBasE Transformations for ThinKCharts (continued on the next page)

```
import synccharts;
import utilities;

//Adds a successor to the given state
Void addSuccessorState(State source):
 let target = new State:
 connectStates(source, target) ->
 source.parentRegion.innerStates.add(target) ->
 setSelection(target)
;

Void addSuccessorStates(List[State] states):
 states.addSuccessorState()
;

//Adds a predecessor to the given state
Void addPredecessorState(State target):
let source = new State:
connectStates(source,target) ->
target.parentRegion.innerStates.add(source) ->
setSelection(source)
;

//Adds a choice to the given state
//by adding a conditional with two
//targets.
Void addChoice(State source):
let choice = new State:
let opt1 = new State:
let opt2 = new State:
let t1 = new Transition:
let t2 = new Transition:
choice.setLabel("C1") ->
choice.setType(StateType::CONDITIONAL) ->
opt1.setLabel("S0") ->
opt2.setLabel("S1") ->
connectStates(source,choice) ->
t1.setSourceState(choice) ->
t1.setTargetState(opt1) ->
t1.setPriority(1) ->
t2.setSourceState(choice) ->
t2.setTargetState(opt2) ->
t2.setPriority(2) ->
source.parentRegion.innerStates.add(choice) ->
source.parentRegion.innerStates.add(opt1) ->
source.parentRegion.innerStates.add(opt2)
;
```

Listing B.2: KSBasE Transformations for ThinKCharts (continued on the next page)

```
//Flips source and target of the
//given transition.
Void flipTransition(Transition t):
 let source = t.sourceState:
 let target = t.targetState:
 t.setSourceState(target) ->
 t.setTargetState(source)
;

//Connects two states
Void connectStates(State source, State target):
let transition = new Transition:
transition.setSourceState(source) ->
transition.setTargetState(target) ->
setSelection(transition)
;

// create a self loop of one state
Void addSelfLoop(State state):
  connectStates(state, state)
;

//Reroutes the target of the given transition
//to the given state
Void rerouteTransitionTarget(Transition t, State target):
t.setTargetState(target)
;

//Reroutes the source of the given transition
//to the given state
Void rerouteTransitionSource(Transition t, State source):
t.sourceState.outgoingTransitions.remove(t) ->
t.setSourceState(source) ->
source.outgoingTransitions.add(t)
;

//Adds a parallel region to the given state
Void upgradeState(State parentState):
let region = new Region:
let state = new State:
state.setIsInitial(true) ->
state.setLabel("Initial") ->
region.innerStates.add(state) ->
parentState.regions.add(region) ->
setSelection(state)
;
```

Listing B.3: KSBasE Transformations for ThinKCharts (continued on the next page)

```

Void downgradeState(State parentState):
if parentState.regions.size > 0 then
 parentState.regions.removeAll(parentState.regions) ->
 parentState.signals.removeAll(parentState.signals)
;

//Returns true if the state is a complex state,
//i.e. it has at least one inner region
Boolean isRegionState(State state):
   state.regions.size > 0
;

//Sets the object that should be selected after the transformation is executed
Void setSelection(Object object):
 JAVA de.cau.cs.kieler.ksbase.ui.utils.TransformationUtils.
     setPostTransformationSelection(java.lang.Object)
;

//Checks if the state is the root state.
Boolean isNoRootState(State state):
   state.parentRegion.parentState != null
;

Boolean isNoRootState(List[State] states):
   states.notExists(e|e.parentRegion.parentState == null)
;

// Add a state to the parent state and add all Regions
// to that new State
Void encapsulateRegions(State parentState):
  let regions = parentState.regions:
  let newRegion = new Region:
  let newState = new State:
  (parentState.regions.size > 0) ?
    (newState.regions.addAll(regions) ->
    parentState.regions.add(newRegion) ->
    newRegion.innerStates.add(newState)) :
   null
;

// Delete the state and move all its regions to the
// parent state
Void flattenState(State state):
let parentRegion = state.parentRegion:
let parentState = parentRegion.parentState:
let stateSize = parentRegion.innerStates.size:
parentState.regions.addAll(state.regions) ->
state.setParentRegion(null) ->
(stateSize <= 1) ?
   parentState.regions.remove(parentRegion) :
null
;
```

Listing B.4: KSBasE Transformations for ThinKCharts (continued on the next page)

```
////////////////////////
//     Templates      ///
/////////////////////////

//Creates a default StateChart
Void createDefault(Region rootRegion):
let state = new State:
let innerState = new State:
let region = new Region:
state.setLabel("SyncChart") ->
state.regions.add(region) ->
innerState.setLabel("Initial") ->
innerState.setIsInitial(true) ->
region.innerStates.add(innerState) ->
rootRegion.innerStates.add(state) ->
setSelection(innerState)
;

//Good ol' ABRO
Void addABRO(Region rootRegion):
let waitABToFinal = new Transition:
let reset = new Transition:
let ABOFinal = new State:
let wATrans = new Transition:
let wBTrans = new Transition:
let wAInitial = new State:
let wAFinal = new State:
let wBInitial = new State:
let wBFinal = new State:
let wA = new Region:
let wB = new Region:
let waitAB = new State:
let ABORegion = new Region:
let ABO = new State:

let innerRootRegion = new Region:
let sigA = new Signal:
let sigB = new Signal:
let sigR = new Signal:
let sigO = new Signal:
let root = new State:
//ABRO Root State
root.setLabel("ABRO") ->
//ABRO Signal definition:
sigA.setName("A") ->
sigA.setIsInput(true) ->
sigB.setName("B") ->
sigB.setIsInput(true) ->
sigR.setName("R") ->
sigR.setIsInput(true) ->
sigO.setName("O") ->
sigO.setIsOutput(true) ->
root.signals.add(sigA) ->
root.signals.add(sigB) ->
root.signals.add(sigR) ->
root.signals.add(sigO) ->
```

Listing B.5: KSBasE Transformations for ThinKCharts

```

//Inner State ABO
ABO.setLabel("ABO") ->
ABO.setIsInitial(true) ->
//Inner state Wait A and B
waitAB.setLabel("WaitAB") ->
waitAB.setIsInitial(true) ->
//Region for 'wait for a'
wAInitial.setLabel("wA") ->
wAInitial.setIsInitial(true) ->
wA.innerStates.add(wAInitial) ->
wAFinal.setLabel("dA") ->
wAFinal.setIsFinal(true) ->
wA.innerStates.add(wAFinal) ->
wATrans.setSourceState(wAInitial) ->
wATrans.setTargetState(wAFinal) ->
wATrans.setTriggersAndEffects("A") ->
waitAB.regions.add(wA) ->
//Region for 'wait for b'
wBInitial.setLabel("wB") ->
wBInitial.setIsInitial(true) ->
wB.innerStates.add(wBInitial) ->
wBFinal.setLabel("dB") ->
wBFinal.setIsFinal(true) ->
wB.innerStates.add(wBFinal) ->
wBTrans.setSourceState(wBInitial) ->
wBTrans.setTargetState(wBFinal) ->
wBTrans.setTriggersAndEffects("B") ->
waitAB.regions.add(wB) ->
//ABO final state
ABOFinal.setLabel("done") ->
waitABToFinal.setSourceState(waitAB) ->
waitABToFinal.setTargetState(ABOFinal) ->
waitABToFinal.setType(TransitionType::NORMALTERMINATION) ->
waitABToFinal.setTriggersAndEffects("/O") ->
ABORegion.innerStates.add(waitAB) ->
ABORegion.innerStates.add(ABOFinal) ->
ABO.regions.add(ABORegion)->
reset.setSourceState(ABO) ->
reset.setTargetState(ABO) ->
reset.setType(TransitionType::STRONGABORT) ->
reset.setTriggersAndEffects("R") ->
innerRootRegion.innerStates.add(ABO) ->
root.regions.add(innerRootRegion) ->
rootRegion.innerStates.add(root)
;
```

Listing B.6: KSBasE Transformations for KoData(continued on the next page)

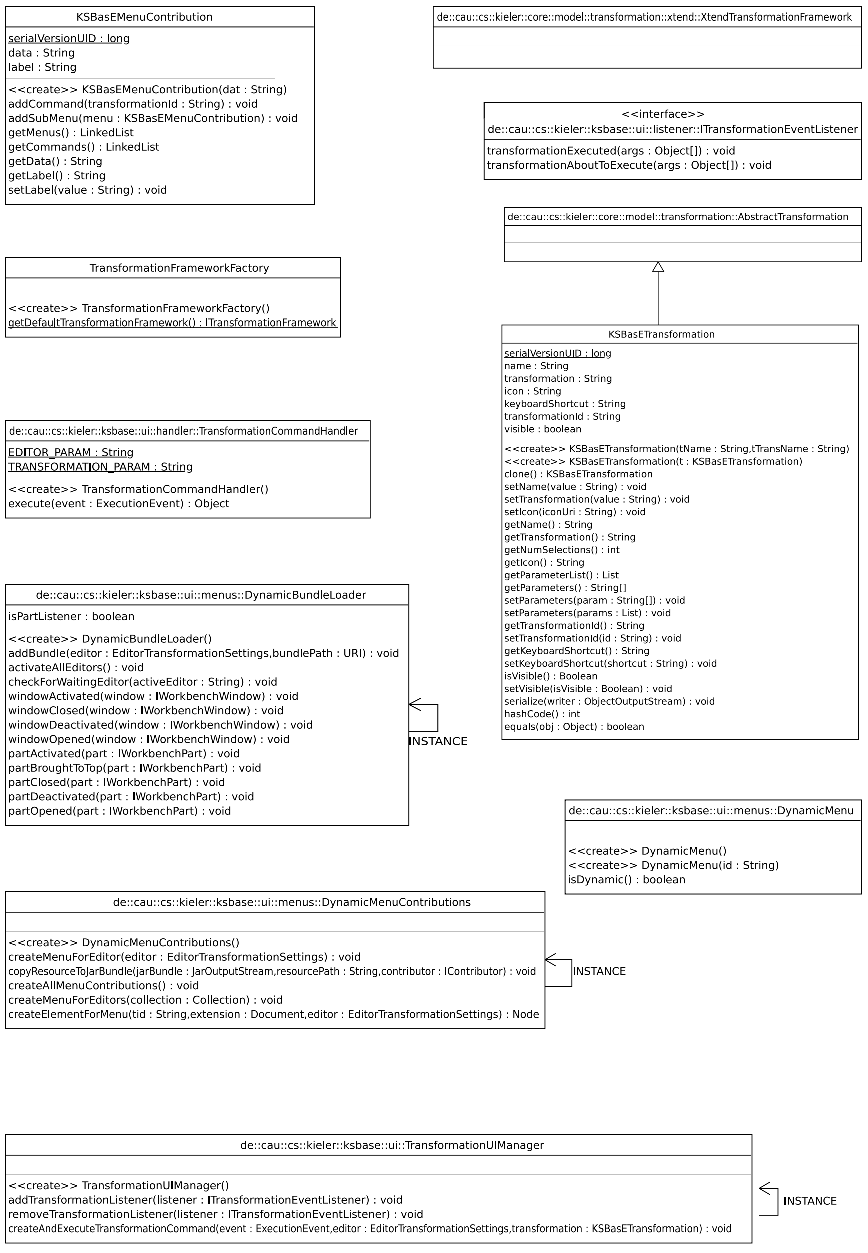
```
import dataflow;

/**
 * Simply connects two ports
 * by adding a connection
 */
Void connectPorts(OutputPort out, InputPort in):
 let con = new Connection:
 con.setSourcePort(out) ->
 con.setTargetPort(in) ->
 out.parentBox.connections.add(con)
;

/**
 * Connects two boxes
 * If any of the boxes has an unused port, the connection is using this port
 */
Void connectBoxes(Box source, Box target):
 let out = new OutputPort:
 let in = new InputPort:
 let sourcePorts = source.outputs.select(e|  !source.connections.exists(c|c.
     sourcePort == e) ) :
 let targetPorts = target.inputs.select( in | !((Box)source.eContainer).boxes.
     connections.exists(c| c.targetPort == in)):

 if (sourcePorts.size == 0 ) then {
   source.outputs.add(out) ->
   if (targetPorts.size == 0) then {
      target.inputs.add(in) ->
      target.inputs.add(in) ->
      connectPorts(out,in)
   }
   else {
      connectPorts(out,targetPorts.first())
   }
 } else {
   if (targetPorts.size == 0) then {
      target.inputs.add(in) ->
      connectPorts(sourcePorts.first(),in)
   }
   else {
      connectPorts(sourcePorts.first(),targetPorts.first())
   }
 }
;

/**
 * Connects an output port to this first input port
 * given. This method is called by ConnectAllBoxPorts
 */
Void connectOutputsToInputs(OutputPort out,List[InputPort] in):
if (in.size > 0) then {
 out.connectPorts(in.first())->
 in.remove(in.first())
 }
;
```

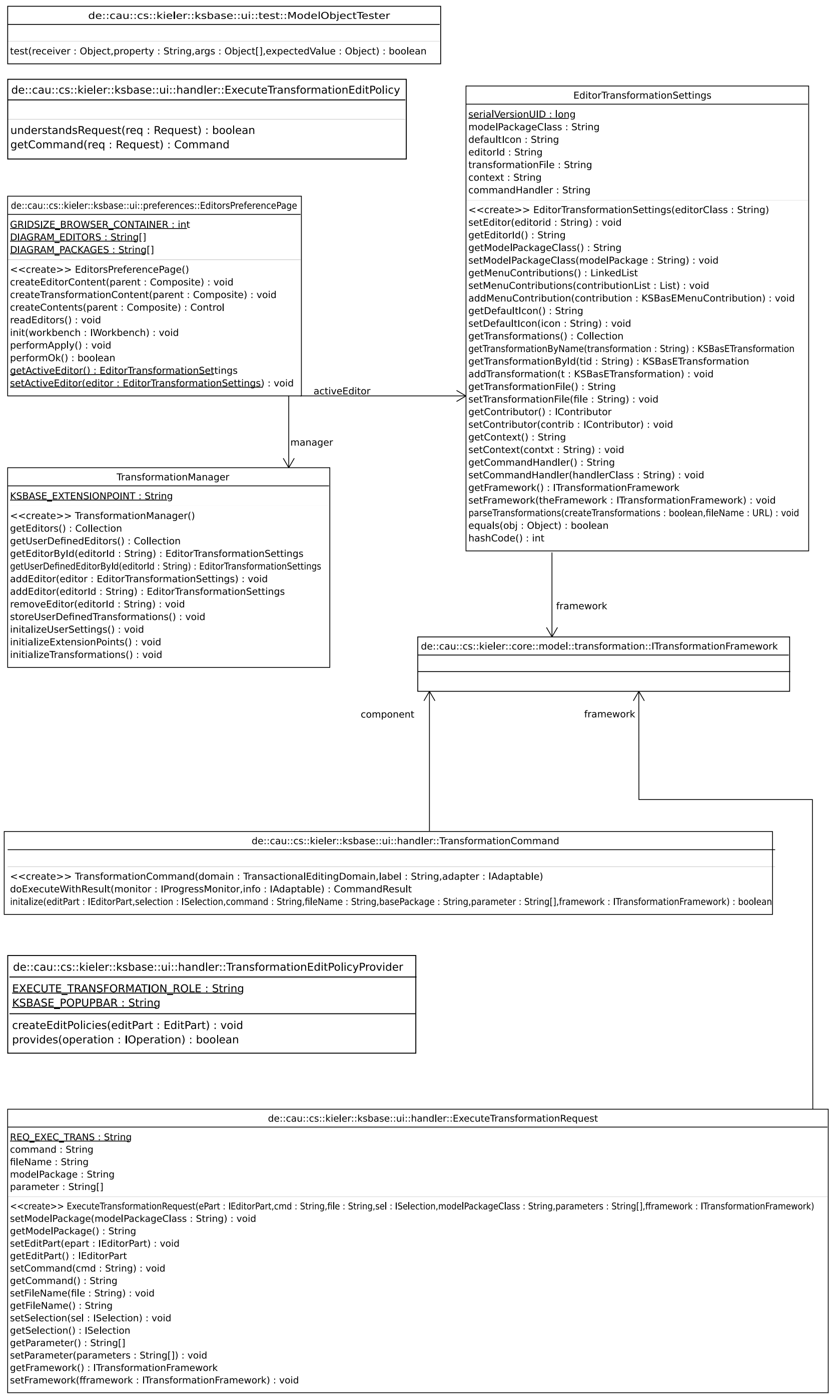
Listing B.7: KSBasE Transformations for KoData

```
/**
 * Connects all open output ports of the source box
 * to all open input ports of the target box.
 * No ports are created here, if one of the boxes has
 * more empty ports they won't be touched.
 * Since there is no enumeration of ports in a
 * Box, this will mess up the source<->target port order
 */
Void connectAllBoxPorts(Box source, Box target):
 let sourceList = source.outputs.select(e| !source.connections.exists(c|c.
     sourcePort == e)):
 let targetList = target.inputs.select( in | !((Box)source.eContainer).boxes.
     connections.exists(c| c.targetPort == in)):
 sourceList.connectOutputsToInputs(targetList)
;

/**
 * Creates a successor box to the source box
 */
Void createSuccessor(Box source):
 let target = new Box:
 connectBoxes(source,target) ->
 ((DataflowModel)source.eContainer).boxes.add(target)
;

/**
 * Creates a predecessor box to the target box
 */
Void createPredecessor(Box target):
 let source = new Box:
 connectBoxes(source,target) ->
 ((DataflowModel)target.eContainer).boxes.add(source)
;

/**
 * Splits the given connection an inserts a new box in
 * between source and target.
 */
Void insertBoxInConnection(Connection source):
let box = new Box:
let con = new Connection:
let tempPort = source.targetPort:
let inP = new InputPort:
let outP = new OutputPort:
box.inputs.add(inP) ->
box.outputs.add(outP) ->
source.setTargetPort(inP) ->
con.setSourcePort(outP) ->
con.setTargetPort(tempPort) ->
((DataflowModel)source.sourcePort.eContainer.eContainer).boxes.add(box) ->
box.connections.add(con)
;
```

Appendix C

Class Diagrams


KSBasEMenuContribution
serialVersionUID : long
data : String
label : String
<<create>> KSBasEMenuContribution(dat : String)
addCommand(transformationId : String) : void
addSubMenu(menu : KSBasEMenuContribution) : void
getMenus() : LinkedList
getCommands() : LinkedList
getData() : String
getLabel() : String
setLabel(value : String) : void
de::cau::cs::kieler::core::model::transformation::xtend::XtendTransformationFramework
<<interface>>
de::cau::cs::kieler::ksbase::ui::listener::ITransformationEventListener
transformationExecuted(args : Object[]) : void
transformationAboutToExecute(args : Object[]) : void
de::cau::cs::kieler::core::model::transformation::AbstractTransformation
TransformationFrameworkFactory
<<create>> TransformationFrameworkFactory()
getDefaultTransformationFramework() : ITransformationFramework
KSBasETransformation
serialVersionUID : long
name : String
transformation : String
icon : String
keyboardShortcut : String
transformationId : String
visible : boolean
<<create>> KSBasETransformation(tName : String,tTransName : String)
<<create>> KSBasETransformation(t : KSBasETransformation)
clone() : KSBasETransformation
setName(value : String) : void
setTransformation(value : String) : void
setIcon(iconUri : String) : void
getName() : String
getTransformation() : String
getNumSelections() : int
getIcon() : String
getParameterList() : List
getParameters() : String[]
setParameters(param : String[]) : void
setParameters(params : List) : void
getTransformationId() : String
setTransformationId(id : String) : void
getKeyboardShortcut() : String
setKeyboardShortcut(shortcut : String) : void
isVisible() : Boolean
setVisible(isVisible : Boolean) : void
serialize(writer : ObjectOutputStream) : void
hashCode() : int
equals(obj : Object) : boolean
de::cau::cs::kieler::ksbase::ui::handler::TransformationCommandHandler
EDITOR_PARAM : String
TRANSFORMATION_PARAM : String
<<create>> TransformationCommandHandler()
execute(event : ExecutionEvent) : Object
de::cau::cs::kieler::ksbase::ui::menus::DynamicBundleLoader
isPartListener : boolean
<<create>> DynamicBundleLoader()
addBundle(editor : EditorTransformationSettings,bundlePath : URI) : void
activateAllEditors() : void
checkForWaitingEditor(activeEditor : String) : void
windowActivated(window : IWorkbenchWindow) : void
windowClosed(window : IWorkbenchWindow) : void
windowDeactivated(window : IWorkbenchWindow) : void
windowOpened(window : IWorkbenchWindow) : void
partActivated(part : IWorkbenchPart) : void
partBroughtToTop(part : IWorkbenchPart) : void
partClosed(part : IWorkbenchPart) : void
partDeactivated(part : IWorkbenchPart) : void
partOpened(part : IWorkbenchPart) : void
INSTANCE
de::cau::cs::kieler::ksbase::ui::menus::DynamicMenu
<<create>> DynamicMenu()
<<create>> DynamicMenu(id : String)
isDynamic() : boolean
de::cau::cs::kieler::ksbase::ui::menus::DynamicMenuContributions
<<create>> DynamicMenuContributions()
createMenuForEditor(editor : EditorTransformationSettings) : void
copyResourceToJarBundle(jarBundle : JarOutputStream,resourcePath : String,contributor : IContributor) : void
createAllMenuContributions() : void
createMenuForEditors(collection : Collection) : void
createElementForMenu(tid : String,extension : Document,editor : EditorTransformationSettings) : Node
INSTANCE
de::cau::cs::kieler::ksbase::ui::TransformationUIManager
<<create>> TransformationUIManager()
addTransformationListener(listener : ITransformationEventListener) : void
removeTransformationListener(listener : ITransformationEventListener) : void
createAndExecuteTransformationCommand(event : ExecutionEvent,editor : EditorTransformationSettings,transformation : KSBasETransformation) : void
INSTANCE


de::cau::cs::kieler::ksbase::ui::test::ModelObjectTester
test(receiver : Object,property : String,args : Object[],expectedValue : Object) : boolean
de::cau::cs::kieler::ksbase::ui::handler::ExecuteTransformationEditPolicy
understandsRequest(req : Request) : boolean
getCommand(req : Request) : Command
de::cau::cs::kieler::ksbase::ui::preferences::EditorsPreferencePage
GRIDSIZE_BROWSER_CONTAINER : int
DIAGRAM_EDITORS : String[]
DIAGRAM_PACKAGES : String[]
<<create>> EditorsPreferencePage()
createEditorContent(parent : Composite) : void
createTransformationContent(parent : Composite) : void
createContents(parent : Composite) : Control
readEditors() : void
init(workbench : IWorkbench) : void
performApply() : void
performOk() : boolean
getActiveEditor() : EditorTransformationSettings
setActiveEditor(editor : EditorTransformationSettings) : void
activeEditor
manager
EditorTransformationSettings
serialVersionUID : long
modelPackageClass : String
defaultIcon : String
editorId : String
transformationFile : String
context : String
commandHandler : String
<<create>> EditorTransformationSettings(editorClass : String)
setEditor(editorid : String) : void
getEditorId() : String
getModelPackageClass() : String
setModelPackageClass(modelPackage : String) : void
getMenuContributions() : LinkedList
setMenuContributions(contributionList : List) : void
addMenuContribution(contribution : KSBasEMenuContribution) : void
getDefaultIcon() : String
setDefaultIcon(icon : String) : void
getTransformations() : Collection
getTransformationByName(transformation : String) : KSBasETransformation
getTransformationById(tid : String) : KSBasETransformation
addTransformation(t : KSBasETransformation) : void
getTransformationFile() : String
setTransformationFile(file : String) : void
getContributor() : IContributor
setContributor(contrib : IContributor) : void
getContext() : String
setContext(contxt : String) : void
getCommandHandler() : String
setCommandHandler(handlerClass : String) : void
getFramework() : ITransformationFramework
setFramework(theFramework : ITransformationFramework) : void
parseTransformations(createTransformations : boolean,fileName : URL) : void
equals(obj : Object) : boolean
hashCode() : int
TransformationManager
KSBASE_EXTENSIONPOINT : String
<<create>> TransformationManager()
getEditors() : Collection
getUserDefinedEditors() : Collection
getEditorById(editorId : String) : EditorTransformationSettings
getUserDefinedEditorById(editorId : String) : EditorTransformationSettings
addEditor(editor : EditorTransformationSettings) : void
addEditor(editorId : String) : EditorTransformationSettings
removeEditor(editorId : String) : void
storeUserDefinedTransformations() : void
initalizeUserSettings() : void
initializeExtensionPoints() : void
initializeTransformations() : void
framework
de::cau::cs::kieler::core::model::transformation::ITransformationFramework
component
framework
de::cau::cs::kieler::ksbase::ui::handler::TransformationCommand
<<create>> TransformationCommand(domain : TransactionalEditingDomain,label : String,adapter : IAdaptable)
doExecuteWithResult(monitor : IProgressMonitor,info : IAdaptable) : CommandResult
initalize(editPart : IEditorPart,selection : ISelection,command : String,fileName : String,basePackage : String,parameter : String[],framework : ITransformationFramework) : boolean
de::cau::cs::kieler::ksbase::ui::handler::TransformationEditPolicyProvider
EXECUTE_TRANSFORMATION_ROLE : String
KSBASE_POPUPBAR : String
createEditPolicies(editPart : EditPart) : void
provides(operation : IOperation) : boolean
de::cau::cs::kieler::ksbase::ui::handler::ExecuteTransformationRequest
REQ_EXEC_TRANS : String
command : String
fileName : String
modelPackage : String
parameter : String[]
<<create>> ExecuteTransformationRequest(ePart : IEditorPart,cmd : String,file : String,sel : ISelection,modelPackageClass : String,parameters : String[],fframework : ITransformationFramework)
setModelPackage(modelPackageClass : String) : void
getModelPackage() : String
setEditPart(epart : IEditorPart) : void
getEditPart() : IEditorPart
setCommand(cmd : String) : void
getCommand() : String
setFileName(file : String) : void
getFileName() : String
setSelection(sel : ISelection) : void
getSelection() : ISelection
getParameter() : String[]
setParameter(parameters : String[]) : void
getFramework() : ITransformationFramework
setFramework(fframework : ITransformationFramework) : void

Appendix D

Bibliography

[1] *GenGED User Manual.* http://user.cs.tu-berlin.de/~genged/Manual/UserManual/UserManual.pdf. ix, 5

[2] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification.* http://www.omg.org/spec/QVT/1.0/. ix, 13, 14

[3] *Request for Proposal: MOF 2.0 Query / Views / Transformation RFP.* OMG Document ad/2002-04-10. 13

[4] Model driven architecture., November 2002. http://www.omg.org/cgi-bin/doc?omg/00-11-05. 7

[5] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003. 26, 52

[6] Roswitha Bardohl. Genged - a generic graphical editor for visual languages based on algebraic graph grammars. In *VL '98: Proceedings of the IEEE Symposium on Visual Languages*, page 48, Washington, DC, USA, 1998. IEEE Computer Society. 3

[7] Özgün Bayramoglu. The KIELER textual editing framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf. 27, 61

[8] Nils Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf. 24

[9] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. A framework for the static and interactive visualization for statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002. 32

[10] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Towards graph transformation based generation of visual editors using eclipse. visual languages and formal methods (vlfm. In *Visual Languages and Formal Methods,*

volume 127 of Electronic Notes in Theoretical Computer Science, pages 127–143. Elsevier, 2004. 4

[11] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. Automatic layout and structure-based editing of UML diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2010)*, Dresden, March 2010. To appear. 57, 72

[12] Hauke Fuhrmann and Reinhard von Hanxleden. Enhancing graphical model-based system design—an avionics case study. In *Conjoint workshop of the European Research Consortium for Informatics and Mathematics (ERCIM) and Dependable Embedded Components and Systems (DECOS) at SAFECOMP'09*, Hamburg, Germany, September 2009. ix, 6

[13] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009. ix, 2, 23, 24

[14] ISO. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. 1996. 7

[15] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, USA, 2006. ACM. 13, 15

[16] Ivan Kurtev, Jean BÃĺzivin, and Mehmet Aksit. Technological spaces: An initial appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, 2002. ix, 7, 8

[17] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003. 27

[18] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34:1045–1079, September 1955. 26

[19] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). 8

[20] M. Minas and G. Viehstaedt. Diagen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. VL '95*, pages 203–210, 1995. 3

[21] Mark Minas. Generating visual editors based on Fujaba/MOFLON and DiaMeta. In Holger Giese and Bernhard Westfechtel, editors, *Proc. Fujaba Days 2006, Bayreuth, Germany, September 28-30, 2006*, pages 35–42, 2006. Technical Report tr-ri-06-275 Universität Paderborn, Fakultät für Elektrotechnik, Informatik und Mathematik, Institut für Informatik. 3

[22] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf. 27

[23] Dragan Macos Nils Mitoussis. Mda transformation languages. In *3rd Workshow on Model-Driven Software Engineering (MDSE 2008)*, 2008. http://www.sig-mdse.org/web/cms/web_OLD/export/sites/sig-mdse/informationen/dokumente/20081211_Mitoussis_Macos.ppt. 12

[24] Kazuo Misue Kozo Sugiyama Peter Eades, Wei Lai. Preserving the mental map of a diagram. Technical report, International Institute for Advanced Study of Social Information Science, Fujitsu Laboratories LTD., 1991. 31

[25] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007. 32

[26] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, December 2008. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf. 24

[27] Matthias Schmeling. ThinKCharts, The Thin KIELER SyncCharts Editor. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf. 26

[28] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf. 24

[29] Gerd Szwillus and Lisa Neal, editors. *Structure-based editors and environments.* Academic Press, Inc., Orlando, FL, USA, 1996. 35

[30] Dániel Varró, Gergely Varró, and András Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205 – 227, 2002. 13

[31] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proceedings of the International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009. 27

[32] Mirko Wischer. Textuelle Darstellung und strukturbasiertes Editieren von Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/miwi-dt.pdf. 5, 52