

CHRISTIAN-ALBRECHTS-UNIVERSITÄT ZU KIEL

Diploma Thesis

Effective Visualization of IEC 61499 Function Blocks

With the CAKeFEED Function Blocks Editor

Matthias Schmeling

April 17, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl. Inf. Miro Spönemann
Dr. Partha S. Roop

Eidesstattliche Erklärung

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

Auckland,

Abstract

Function blocks are a commonly used language for the description of distributed control systems. The new standard IEC 61499 provides a mighty and comfortable means of function blocks design, especially for embedded real-time systems. Currently, there is a limited number of tools available that allow to create software systems with function blocks. These are either commercial and not open source or have only restricted usability. The CAKeFEED¹ function blocks editor is a subproject of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) and provides a comfortable open source tool to visually manipulate function blocks together with dynamically updated graphical representations, automatic layout for function block networks, and a simulation mechanism for counter examples.

¹Collaboration between Auckland and Kiel on effective Function Block Examination, Editing, and Depiction

I would like to thank the following people without whom I would not have accomplished this:

Özgün Bayramoglu for always being so supportive.

Falk Starke for making that funny distinctive noise.

Michael Matzen for being the funny guy he is.

Miro Spönemann and Hauke Fuhrmann for being the best advisors I could have run into.

Partha S. Roop and Prof. Reinhard v. Hanxleden for giving me the chance to come to Auckland.

And all the others of the Real-Time and Embedded Systems group for being such nice people.

Contents

1	Introduction	1
2	IEC 61499 Function Blocks	3
2.1	The Basic Function Block	4
2.1.1	The Execution Control Chart	4
2.1.2	The Hierarchical Concurrent Execution Control Chart	5
2.1.3	The Algorithms	5
2.1.4	Association of events and variables	5
2.2	The Composite Function Block	7
2.3	The Service Interface Function Block	7
2.4	The Function Block XML Format	9
3	Related Work	13
3.1	FBDK	13
3.1.1	Views	13
3.1.2	Graphical and textual development	13
3.1.3	User interaction	13
3.1.4	Library mechanism	15
3.1.5	Summary	15
3.2	4DIAC	15
3.2.1	Views	15
3.2.2	User interaction	15
3.2.3	Library mechanism	17
3.2.4	Summary	19
3.3	ISaGRAF	19
3.4	NxtControl	19
3.5	FBench	19
4	The Eclipse Project	21
4.1	Eclipse	21
4.2	Eclipse Modeling Framework	22
4.3	Graphical Editing Framework and Draw2D	22
4.4	Graphical Modeling Framework	24
4.4.1	The Ecore Model	24
4.4.2	The Generator Model	25
4.4.3	The Graphical Definition Model	25
4.4.4	The Tooling Definition Model	26

Contents

4.4.5	The Mapping Definition Model	26
4.4.6	The Diagram Editor Generator Model	27
4.4.7	The generated Editor	27
4.5	Xtend	27
4.6	Xpand	29
4.7	Check	30
4.8	The KIELER Project	31
5	Concept	33
5.1	Modeling approaches	33
5.1.1	The XML-conformant Meta-Model	33
5.1.2	The custom Meta-Model	36
5.1.3	The hybrid Meta-Model	38
5.1.4	The remaining models	42
5.2	The Structure of the Editor	42
5.2.1	Hierarchy Levels	42
5.2.2	Separation of Concerns	42
5.2.3	Reuse of Elements	42
5.2.4	CAKeFEED and KIELER	44
5.3	Code Modifications	44
5.3.1	Attribute and Type Awareness	46
5.3.2	Port Layout	46
5.4	Simulation	46
5.4.1	The KIELER Execution Manager	46
5.5	Import and Export	47
6	Implementation and Results	49
6.1	Modeling	49
6.1.1	The Meta-Model	49
6.1.2	The Graphical Definition Models	52
6.1.3	The Tooling Definition Models	55
6.1.4	The Mapping Models	55
6.2	Code Modifications	56
6.2.1	Type and Attribute Awareness	56
6.2.2	Port Layout	62
6.3	Simulation	63
6.3.1	The Data Component	63
6.4	Import and Export	67
6.4.1	The Import Handler	67
6.4.2	The Export Handler	68
6.4.3	The Transformations	68
6.5	Evaluation	68
7	Conclusion	71

7.1	Summary	71
7.2	Future Work	71
8	Bibliography	73

Contents

List of Figures

2.1	A function block that conforms to the IEC 61499 standard.	3
2.2	A simple function block network.	4
2.3	A simple Execution Control Chart (ECC).	4
2.4	Execution of an ECC.	6
2.5	An Hierarchical Concurrent Execution Control Chart (HCECC).	7
2.6	Execution of an HCECC.	8
2.7	Execution of an HCECC.	9
2.8	A basic and a composite function block.	10
2.9	A service interface function block and a service sequence.	10
2.10	The basic function block that corresponds to the XML text in Listing 2.1.	11
3.1	The Function Block Development Kit (FBDK) workspace.	14
3.2	The user interface of Framework for Distributed Industrial Automation and Control (4DIAC).	16
3.3	Editing function block interfaces in 4DIAC.	17
3.4	In 4DIAC ECCs are edited in a different tab.	18
3.5	The workspace of FBench.	20
4.1	The roles of the different projects.	22
4.2	A simple ecore model.	23
4.3	The complete Graphical Modeling Framework (GMF) process.	24
4.4	A graphical definition model of the company.	25
4.5	A tooling definition model to modify a company diagram.	26
4.6	The mapping definition model for the company example.	26
4.7	The finished Company Diagram Editor.	28
5.1	Adding input events.	35
5.2	A custom ecore model for function blocks.	37
5.3	A library mechanism for the editor could work like this.	38
5.4	Hierarchy levels in the old and the new ecore model.	39
5.5	An extract of the hybrid meta-model.	41
5.6	Loading a resource.	43
6.1	The Eclipse Modeling Framework (EMF) tree editor.	50
6.2	The three main classes of the meta-model.	51
6.3	The different types of events and variables.	53
6.4	The different types of connections.	54

List of Figures

6.5	How events and variables may be connected.	55
6.6	How different ports appear in the diagram.	56
6.7	The workflow of the method <code>handleNotificationEvent</code>	57
6.8	The hierarchy of the command classes.	58
6.9	The relative sizes of the upper and lower parts change depending on the numbers of events and variables.	58
6.10	How edit parts react to changes in model elements.	59
6.11	The old and the new attribute awareness.	60
6.12	What happens when a change occurs.	60
6.13	Input ports point to the left while output ports point to the right.	61
6.14	The different port figures.	61
6.15	The default <code>BorderItemLocator</code> and the one employed by the function blocks editor.	62
6.16	The interface provided by the <i>KIEM</i>	63
6.17	The behavior of the data component.	64
6.18	The corresponding function block network.	64
6.19	The ECC of function block A.	65
6.20	Simulation of the function block network.	67
6.21	Simulation of the <i>ECC</i>	67

List of Acronyms

4DIAC	Framework for Distributed Industrial Automation and Control
BFB	Basic Function Block
CFB	Composite Function Block
CAKeFEED	Collaboration between Auckland and Kiel on effective Function Block Examination, Editing, and Depiction
ECC	Execution Control Chart
EMF	Eclipse Modeling Framework
FBDK	Function Block Development Kit
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HCECC	Hierarchical Concurrent Execution Control Chart
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KSBasE	KIELER Structure-Based Editing
OCL	Object Constraint Language
PLC	Programmable Logic Controller
QVTO	Query/View/Transformation Operational
SIFB	Service Interface Function Block
TMF	Textual Modeling Framework
UML	Unified Modeling Language
W3C	World Wide Web Consortium

List of Figures

XML	Extensible Markup Language
XSD	XML Schema Definition
XPath	XML Path Language

1 Introduction

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is a project that provides features to enhance graphical software development. It is discussed in detail in Section 4.8. One of its goals is to provide diagram and textual editors for a wide range of modeling languages and another one is to improve these with lots of generic enhancements that naturally work with all of its editors. With this in mind, developing a visual editor for the IEC 61499 standard suggests itself. This standard employs *function blocks* to describe complex software systems. That the editor is a part of the KIELER project brings several advantages. On the one hand, it means that it can benefit from the Graphical Modeling Framework (GMF). This framework provides a means to generate code for graphical editors from a handful of input models and is discussed in detail in Section 4.4. On the other hand, the editor also benefits from the wide range of features provided by KIELER itself.

Problem Description When designing a graphical editor a lot of questions have to be answered. One important decision that has to be made is what kinds of editing methods the editor should offer. Another one is how the information is to be displayed that is important for the edited model elements. Furthermore, while it is common to start from scratch it is also possible to take an existing editor and enhance it with additional features. When designing an editor from scratch, it is further to be decided what the underlying model of the data looks like. In the case of the GMF there are several possible means to provide these models: It can be provided by using an *ecore model*, an XML Schema Definition (XSD), or annotated Java code, for example. In addition, since the GMF generates a generic graphical editor, some parts of it might not be suitable for editing function blocks, which means that they have to be altered or removed. Sometimes even new features have to be implemented. In the case of the function blocks editor, which has been developed in the course of this thesis and is called Collaboration between Auckland and Kiel on effective Function Block Examination, Editing, and Depiction (CAKeFEED), the topics that have to be addressed are the implementation of attribute awareness, additional layouts, a means of simulation of counter examples, and a custom import and export mechanism.

Thesis Outline In this thesis the development of the CAKeFEED function blocks editor is discussed. The next chapter deals with the basics of the IEC 61499 function blocks language. More detailed information about the standard can be found in one of the books by Valeriy Vyatkin [24]. Chapter 3 introduces several existing function block editors. Chapter 4 introduces the *Eclipse Project* and related technologies that the CAKeFEED function blocks editor is based on. In Chapter 5 the ideas behind the

1 Introduction

project are discussed while Chapter 6 is about details of the implementation and its results. Chapter 7 concludes this thesis with a summary and an outlook on possible future work.

2 IEC 61499 Function Blocks

IEC 61499 [2] is a new standard for distributed control systems. It is an advancement of the IEC 61131 standard, which deals with Programmable Logic Controllers (PLCs). For more information on IEC 61131 and PLCs see the book by Karl-Heinz John and Michael Tiegelkamp [15] and the one by W. Bolton [8], respectively.

In general, function blocks are a means to describe several independent computational units and their relations with each other. IEC 61499 however, specifies the structure of a computational unit more precisely than its predecessor. Every function block is divided into two parts: The controlling part and the data part. Figure 2.1 shows this structure.

The controlling part is responsible for the behavior of the function block. It is influenced by the input events on the left side. Depending on the current state of the function block it may send out events that in turn affect the behavior of other function blocks. The results of a function block's computations depend on the incoming data that arrives through the input variables on the left side. The results of a computation are sent through the output variables on the right and may be reused by other function blocks.

A set of interconnected function blocks forms a function block network. An example network is shown in Figure 2.2.

IEC 61499 further distinguishes different types of function blocks. These are described in the following sections.

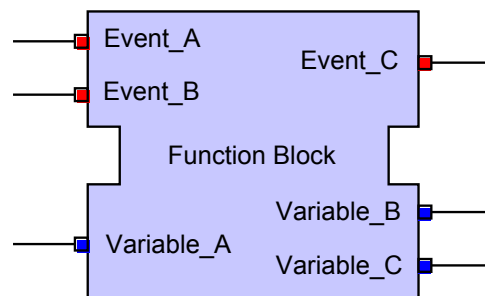


Figure 2.1: A function block that conforms to the IEC 61499 standard.

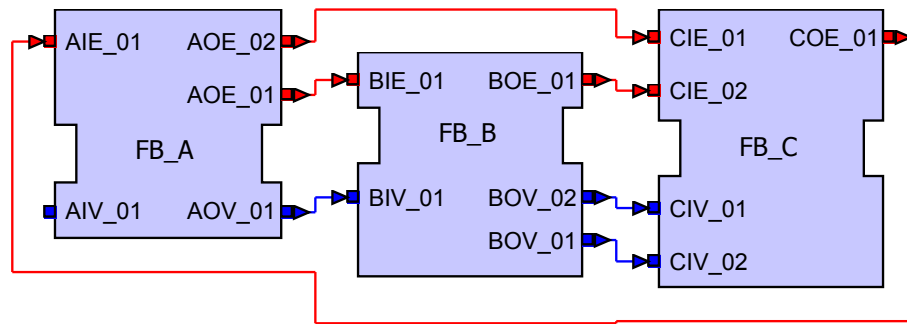


Figure 2.2: A simple function block network.

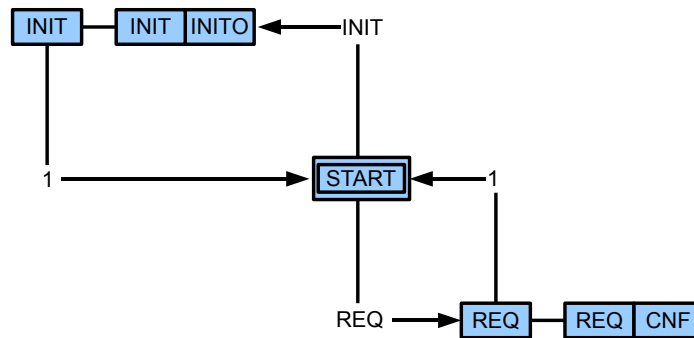


Figure 2.3: A simple ECC.

2.1 The Basic Function Block

Basic Function Blocks (BFBs) are the atomic units in a function block network. Their behavior is defined by an Execution Control Chart (ECC). An ECC is an event-driven *state machine*. Figure 2.3 depicts a simple example ECC. For more detailed information on state machines and ECCs see the books by Egon Börger and Robert Stark [9] and K. Thramboulidis [23].

2.1.1 The Execution Control Chart

As can be seen in the Figure 2.3, an ECC consists of a number of states, which are represented by rectangles. Only one of the states may be active at any time. In addition, states may be connected by transitions, which are shown as arrows. As soon as the trigger of one of the outgoing transitions of the active state is true, that

transition is taken. This means that its target state becomes the currently active one. A 1 indicates that the trigger is always true and thus the transition is always taken. The state with the label **START** is always active at first. States may also reference an action, which consists of a reference to an algorithm and an output event. Upon activation of the state, the referenced algorithm is executed and the output event is emitted. Note that these output events correspond to the aforementioned output events that control the behavior of other function blocks. Figure 2.4 depicts a sample execution of an ECC.

2.1.2 The Hierarchical Concurrent Execution Control Chart

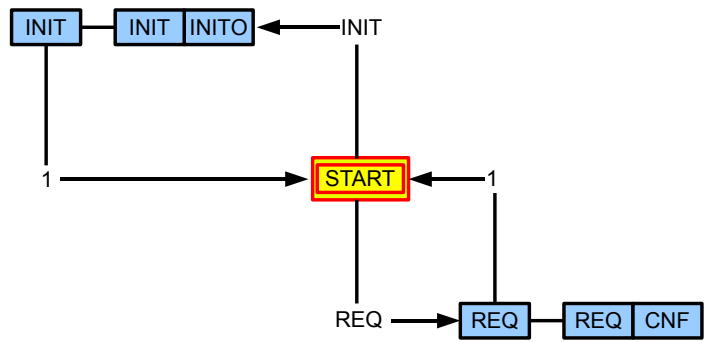
To simplify the creation and depiction of more complex ECCs, a new version of this language has been developed by Gareth Shaw [13] that supports hierarchy as well as parallelism. These charts are called Hierarchical Concurrent Execution Control Charts (HCECCs). The support of hierarchy has been realized by allowing any state in an ECC to be replaced by another ECC. Whenever the replaced state would usually become active, the **START** state of the replacing ECC is activated instead. In addition, the ECC may be deactivated as soon as one of its outgoing transitions is taken, just as a normal state. Parallelism is realized by allowing an ECC to be replaced by a number of other ECCs. When such a compound of ECCs is activated, every single **START** state is activated and any of their outgoing transitions may be taken simultaneously. The deactivation of an ECC compound is the same as for ECCs and states. Note that while HCECCs have a strong resemblance to Charles André's *SyncCharts* language [5], there are semantical ambiguities regarding their execution. One example of such an ambiguity is the question whether inner states or the **START** state of a deactivated ECC will be activated again once the ECC is reentered. Figure 2.5 depicts an HCECC with hierarchy and parallelism, and Figures 2.6 and 2.7 show a sample execution of that HCECC.

2.1.3 The Algorithms

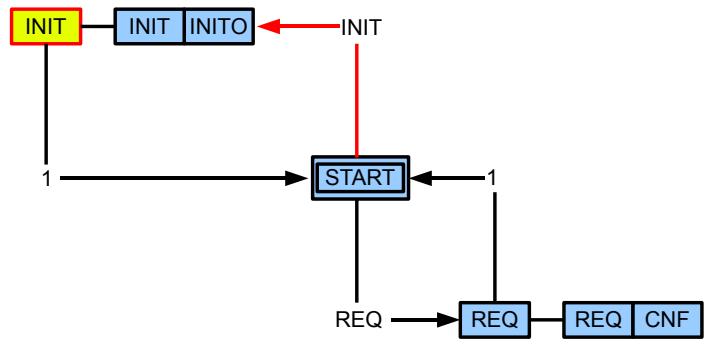
As has been mentioned previously, the activation of a state in an ECC or HCECC may invoke the execution of an algorithm. The algorithm in turn reads the values of the input variables and uses them to compute values for the output variables. Usually, an algorithm consists solely of source code written in a specific language, such as *Java* or *C++*. However, other forms such as *structured text* or *ladder diagrams* are also possible.

2.1.4 Association of events and variables

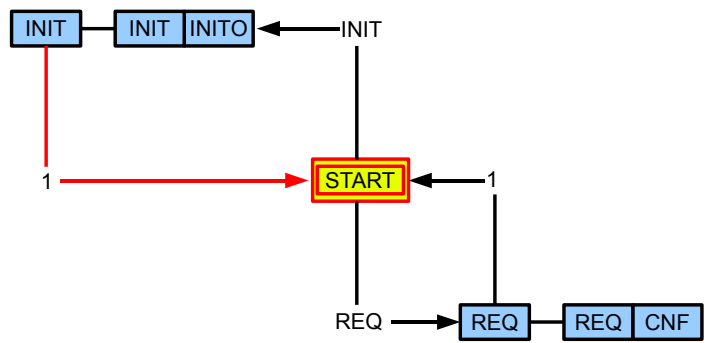
In Figure 2.8(a) a BFB is depicted. The final element that completes the description of a BFB are the associations between events and variables. An important question regarding variables is the point in time when they are updated to their new values. Since algorithms take some amount of time to execute and compute the values of variables in between, some of the output variables of a function block might already



(a) START state is active.



(b) Signal INIT is received.



(c) Return to START state.

Figure 2.4: Execution of an ECC.

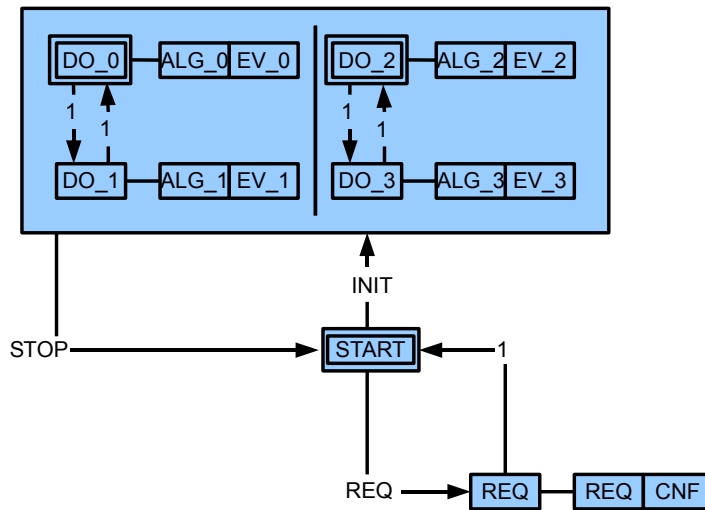


Figure 2.5: An HCECC.

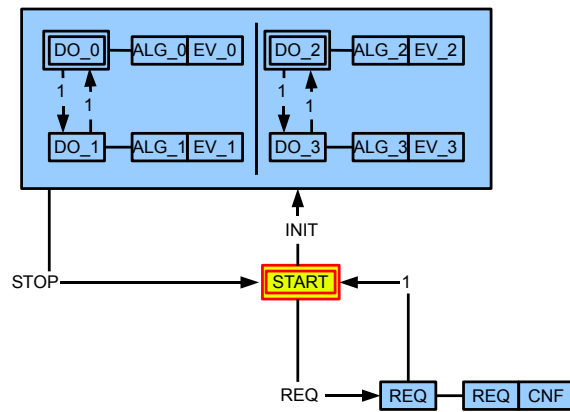
have changed while others have not. This might lead to inconsistent behavior. To remedy this, all variables may be connected to one or more events and are only updated when these events occur. This serves as a means of synchronization to function blocks. Associations are depicted by black lines as can be seen in Figure 2.8(a). In the example, variable `AIV_01` is updated when event `AIE_01` is active and variable `AOV_01` is updated when event `AOE_02` is active.

2.2 The Composite Function Block

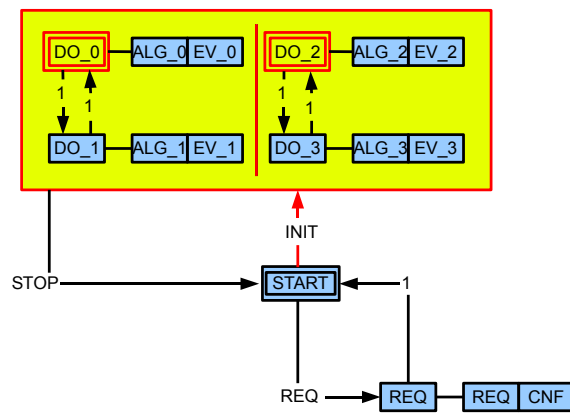
Like BFBs, Composite Function Blocks (CFBs) also contain a number of input and output events and variables that may be connected by associations. However, as shown in Figure 2.8(b), the interior of a CFB looks entirely different. It consists of a number of other function blocks of which the events and variables may be connected with each other. With this it is possible to express hierarchies, and function block networks of arbitrary complexity can be created. Note that unlike function block networks, composite function blocks contain additional interface events and variables that may be connected to inner function blocks.

2.3 The Service Interface Function Block

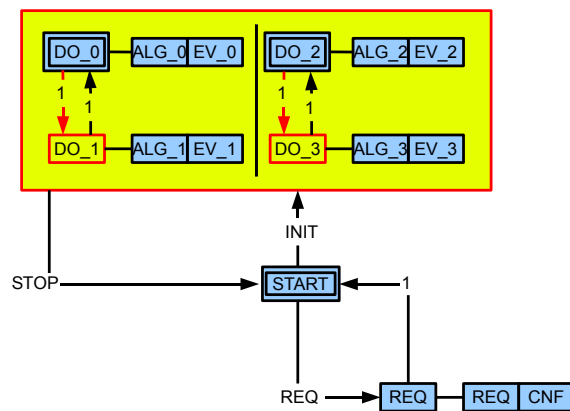
While the Service Interface Function Block (SIFB) has input and output events and variables and associations just like BFBs and CFBs, there is no such thing as an *interior* of SIFBs since this is hidden from the user. Instead, SIFBs act as interfaces



(a) START state is active.

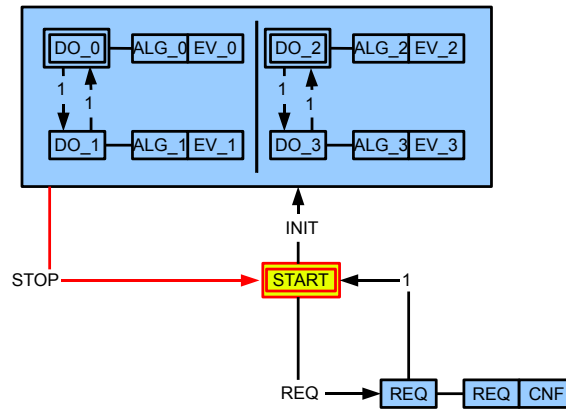


(b) Signal INIT is received.



(c) Transitions are taken simultaneously.

Figure 2.6: Execution of an HCECC.



(a) Signal STOP is received.

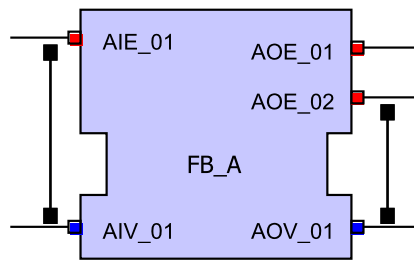
Figure 2.7: Execution of an HCECC.

to parts of the software system that are not part of the function block network, such as hardware controllers for example. The behavior of an SIFB is described by a number of *service sequences*, which define what kinds of outputs are generated by the occurrence of certain inputs. Figure 2.9(b) shows a simple service sequence for the successful handling of a request that could be one of many service sequences for the SIFB depicted in Figure 2.9(a). Whenever the event REQ is present, the request is handled. In the case of a successful execution the event CNF is returned.

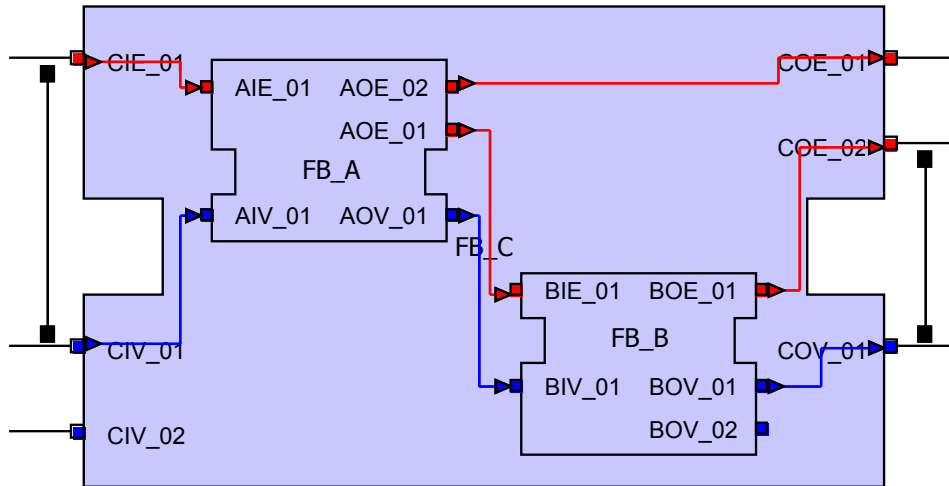
2.4 The Function Block XML Format

There is a commonly used Extensible Markup Language (XML) [14] Format for the description of function blocks and function block networks that is also defined in the standard [3]. For the remainder of this thesis, it is sufficient to understand the example shown in Listing 2.1, which corresponds to the BFB shown in Figure 2.10.

The example above showcases three important things to note about the XML format: It supports the specification of a lot more information than is necessary for the description and execution of function blocks. This includes version information, compiler information, and much more. Also, the format employs intermediate containers for child elements on many occasions. For example, instead of distinguishing input events and output events with two different tags, both types of events are represented by the same tag and then put into an intermediate container called `EventInputs` or `EventOutputs`. The Listings 2.2 and 2.3 illustrate the difference. In addition, all references to other function blocks, events, variables, etc. consist solely of a string which is the name of the referenced element. This implies that every name has to be unique. Why these three facts are important is explained in detail in Section 5.1.

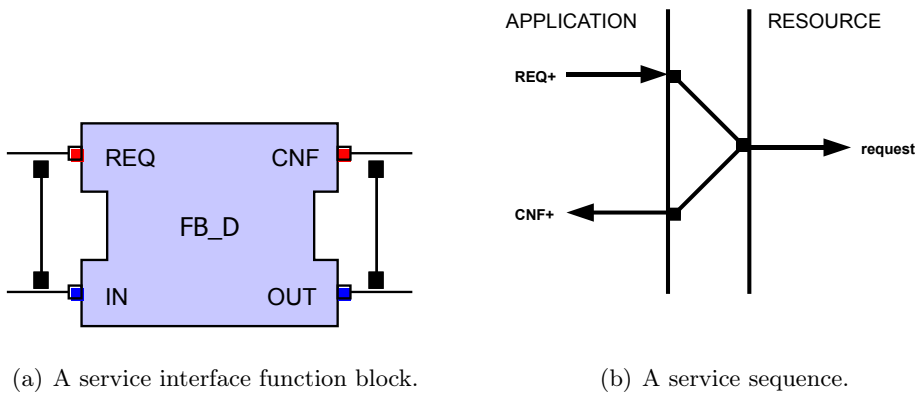


(a) A basic function block.



(b) A composite function block.

Figure 2.8: A basic and a composite function block.



(a) A service interface function block.

(b) A service sequence.

Figure 2.9: A service interface function block and a service sequence.

Listing 2.1: A basic function block in XML format.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
  <FBType Name="FBAND" Comment="Boolean AND" >
    <Identification Standard="61499-1-D.1" Classification="Boolean functions" />
    <VersionInfo Organization="RockwellAutomation" Version="0.1" Author="JHC" Date="01-08-05" />
6    <VersionInfo Organization="RockwellAutomation" Version="0.0" Author="JHC" Date="99-02-20" />
    <CompilerInfo header="package fb.rt.math;" classdef="class FB_AND extends FBFunction2" >
      <Compiler Language="Java" Vendor="IBM" Product="VisualAge" Version="3.0" />
    </CompilerInfo>
    <InterfaceList>
11      <EventInputs>
        <Event Name="REQ" >
          <With Var="IN1" />
          <With Var="IN2" />
        </Event>
16      </EventInputs>
        <EventOutputs>
          <Event Name="CNF" >
            <With Var="OUT" />
          </Event>
21      </EventOutputs>
        <InputVars>
          <VarDeclaration Name="IN1" Type="BOOL" />
          <VarDeclaration Name="IN2" Type="BOOL" />
        </InputVars>
26      <OutputVars>
          <VarDeclaration Name="OUT" Type="BOOL" Comment="Result" />
        </OutputVars>
    </InterfaceList>
    <BasicFB>
31      <Algorithm Name="REQ" >
        <Other Language="Java" Text="public void service_REQ(boolean qi)..." />
      </Algorithm>
    </BasicFB>
  </FBType>

```

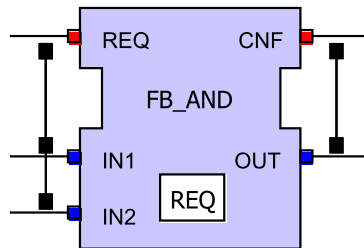


Figure 2.10: The basic function block that corresponds to the XML text in Listing 2.1.

Listing 2.2: A function block in XML with intermediate containers.

```
<FBType Name="FB AND" Comment=" Boolean AND" >
  <InterfaceList>
    <EventInputs>
      <Event Name="A" >
5        <With Var="C" />
      </Event>
    </EventInputs>
    <EventOutputs>
      <Event Name="B" >
10       <With Var="D" />
      </Event>
    </EventOutputs>
    <InputVars>
      <VarDeclaration Name="C" Type="BOOL" />
15    </InputVars>
    <OutputVars>
      <VarDeclaration Name="D" Type="BOOL" />
20    </OutputVars>
  </InterfaceList>
  <BasicFB>
    ...
  </BasicFB>
</FBType>
```

Listing 2.3: A function block in XML without intermediate containers.

```
<FBType Name="FB AND" Comment=" Boolean AND" >
2  <InputEvent Name="A" With="C" />
   <OutputEvent Name="B" With="D" />
   <InputVar Name="C" Type="BOOL" />
   <OutputVar Name="D" Type="BOOL" />
   <BasicFB>
7   ...
   </BasicFB>
</FBType>
```

3 Related Work

This section discusses some existing function block editors, their main features and editing methods, and why it was decided to stick with the KIELER project.


3.1 FBDK

The FBDK [10] is a free function blocks editor developed by Rockwell Automation¹ and managed by Holobloc, Inc.². It is widely used in applied research. This section discusses the key aspects that affect function block development with FBDK.

3.1.1 Views

A screenshot of FBDKs workspace is shown in Figure 3.1. It provides three different views on the currently edited function block. In the upper left there is a window that shows the outline of the function block. It is mainly used to select different parts that are to be modified, such as the interface, which specifies inputs and outputs, the ECC of a BFB, or the function block network of a CFB. The currently selected part is graphically depicted in the window on the upper right. Finally, there is the textual view at the bottom that shows either the corresponding source code [2] or the equivalent XML source text [3].

3.1.2 Graphical and textual development

The textual view is not only there to look at the XML source text or source code. It can also be used as a textual editor to modify the source text. Any changes made to the text are also reflected in the graphical view, albeit only after pressing the  PARSE button in the toolbar. Note that while this feature works with the XML source text, it is not available for the source code.

3.1.3 User interaction

The only other way to edit the properties of a function block is to right-click into the graphical view and choose an action from the context menu, which makes special editing windows appear. The connection of events and variables is also done with the help of the context menu since there is no drag-and-drop mechanism. Additionally, the possibilities to change the graphical appearance of the diagrams is limited. While

¹<http://www.rockwellautomation.com/>

²<http://www.holobloc.com/>

3 Related Work

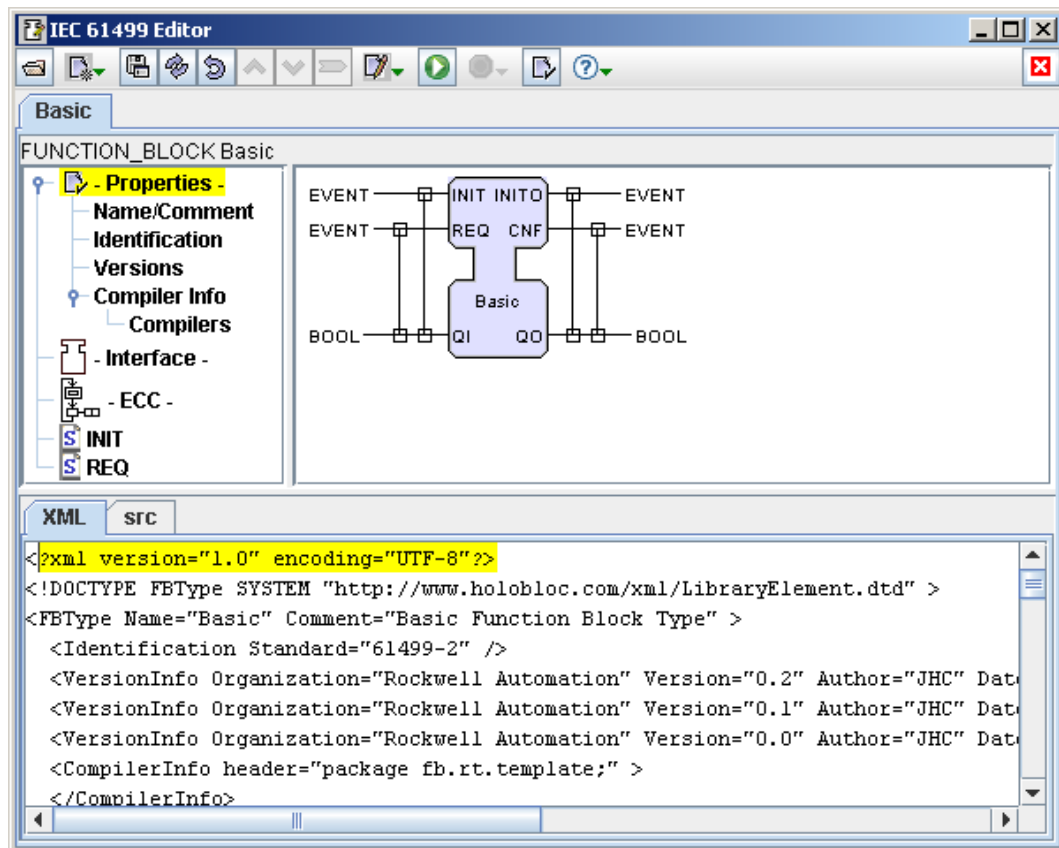


Figure 3.1: The FBDK workspace.

it is possible to select a single element like an *ECC State* and drag it around, there is no way of selecting multiple elements at once. Also, there are no automatic alignment or layout options available.

3.1.4 Library mechanism

When developing a CFB it is necessary to create and modify its function block network. New function blocks are inserted into the network by using the context menu and selecting **NEW** and **FB**. Then, a file wizard pops up from which the user can select a file that stores another function block and give a name to it. This mechanism is not as comfortable as dragging and dropping from a palette but still fast and easy to use.

3.1.5 Summary

The FBDK is a good free tool that has some nice features such as the graphical and textual view, and the possibility of editing function blocks textually. However, the user interface sometimes can be tedious as diagrams have to be laid out manually, one element at a time.

3.2 4DIAC

The 4DIAC project, developed by the PROFACOR GmbH, provides an open IEC 61499 compliant function block editor for Eclipse and uses the Graphical Editing Framework (GEF), a framework that allows to create graphical user interfaces. However, it was developed without the help of the GMF, which means that it cannot benefit from the features offered by the KIELER project.

3.2.1 Views

Figure 3.2 depicts the user interface of 4DIAC. It provides an overview and an outline on the left side, a diagram view where interfaces, networks and ECCs can be modified, a library view on the right with all available types of function blocks, and a properties view at the bottom where several properties of diagram elements can be adjusted.

3.2.2 User interaction

Instead of employing context menus, almost all modifications to function blocks are done with the help of the properties view at the bottom. Names, comments, and other attributes can be changed there. In addition, 4DIAC sometimes provides specialized views to manipulate certain attributes. For example, when editing the interface of a function block, a window with multiple tabs for adding and removing events and variables is displayed, as shown in Figure 3.3. Also, instead of selecting parts of function blocks like ECCs or function block networks in the overview, 4DIAC employs multiple tabs that are each responsible for one part of the function block.

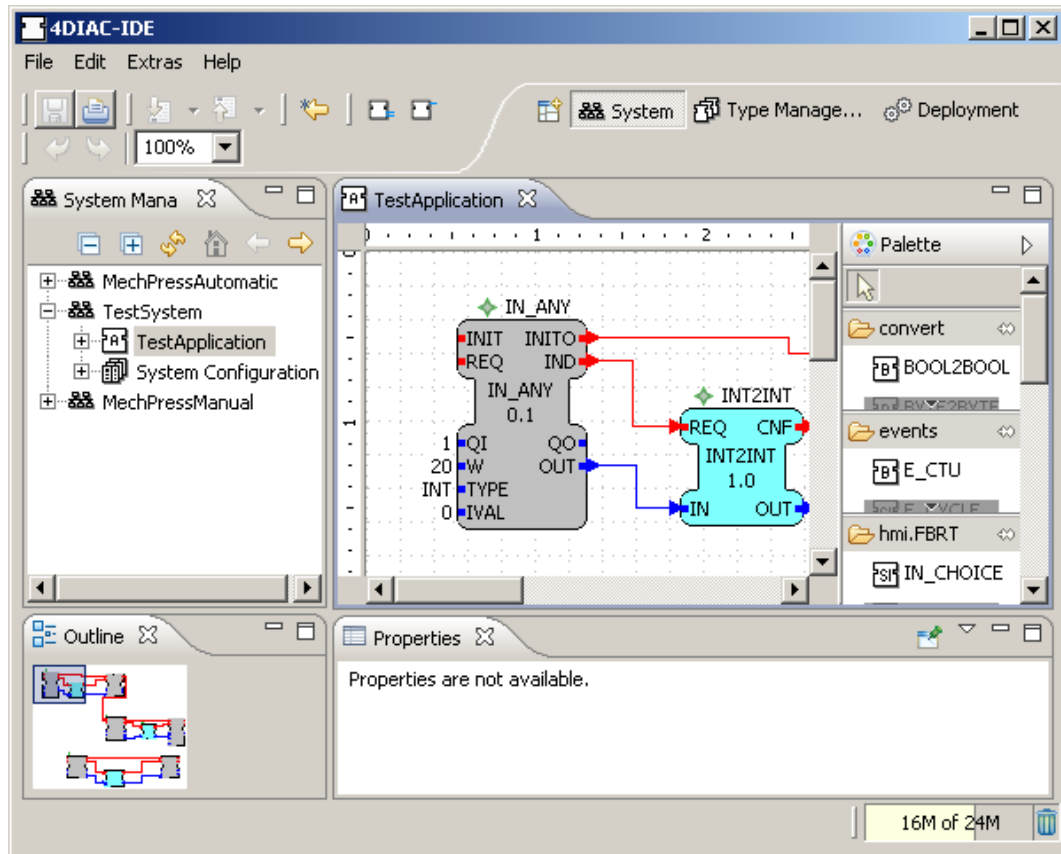


Figure 3.2: The user interface of 4DIAC.

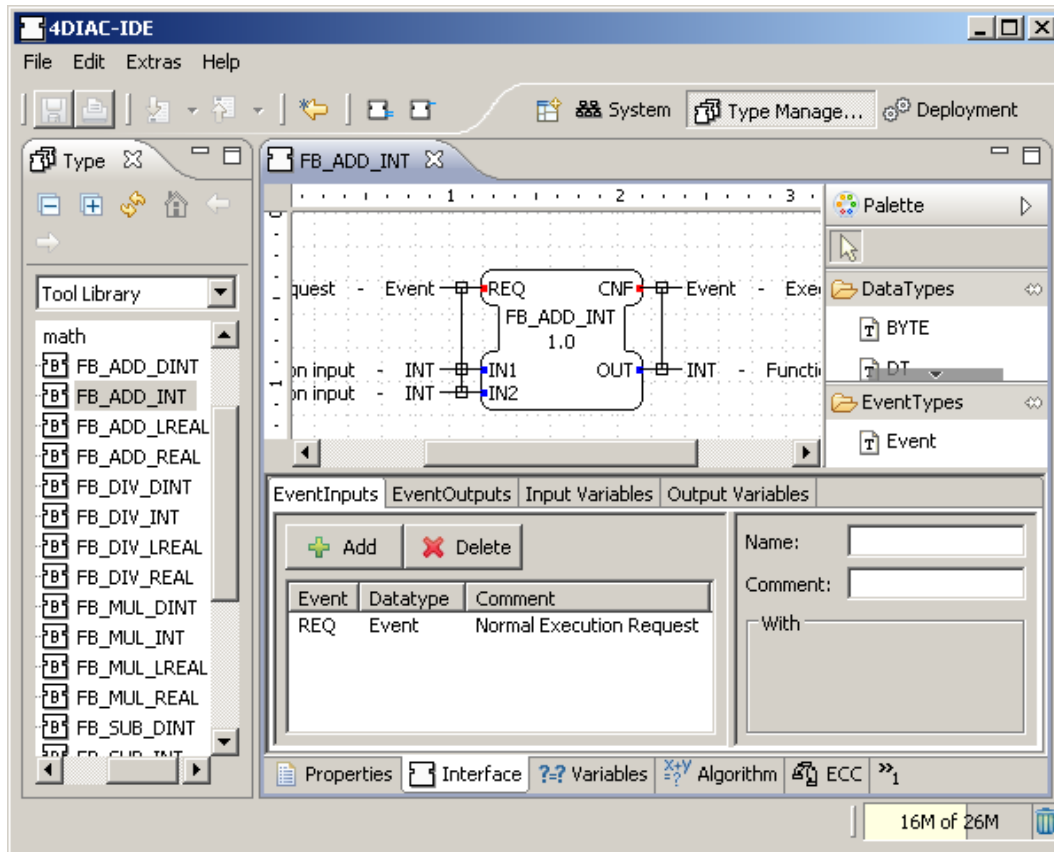


Figure 3.3: Editing function block interfaces in 4DIAC.

For example, there is a special tab that opens a new editor to modify the ECC of a function block. This is depicted in Figure 3.4.

Connecting function blocks in 4DIAC is more comfortable than in FBDK. It is done by simply clicking and holding the left mouse button over an event or a variable and releasing it over another one. However, one major shortcoming is that connections cannot be rerouted by the user. Instead, 4DIAC routes them automatically but always chooses the direct path.

3.2.3 Library mechanism

The insertion of function blocks into a network is quick and comfortable in 4DIAC. Predefined or self-defined types of function blocks can be dragged from the palette on the right and dropped onto the canvas in the middle of the workspace. In addition, it is possible to create groups of types in the palette and to delete them. Also, the appearance of the palette can be modified for example by laying out the types in lists or columns.

3 Related Work

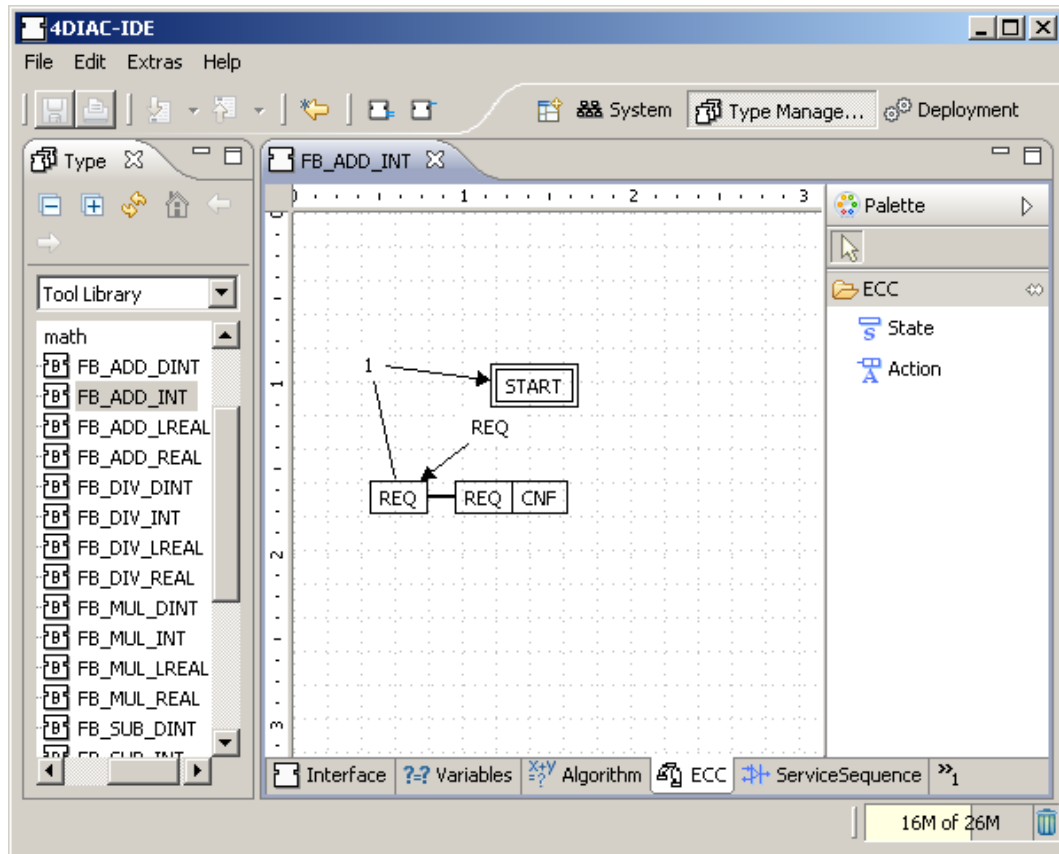


Figure 3.4: In 4DIAC ECCs are edited in a different tab.

3.2.4 Summary

As a GEF project, 4DIAC comes with drag-and-drop editing and a wide range of customizable views. It also offers some additional features such as the function block type palette that can be extended by the user. It also separates concerns by providing different tabs for the function block interface, the ECC, the network and so on. However, the fact that the connections are not routable by the user is a major shortcoming, especially in large diagrams.

3.3 ISaGRAF

*ISaGRAF*³ is a commercial function block tool developed by ICS Triplex ISaGRAF Inc. that offers a wide range of functionality in all aspects of function block development. However, since it is not publicly available it could not be tested.

3.4 NxtControl

Another commercially available tool is NxtControl⁴ by the nxtControl GmbH. It has a similarly rich set of functionality like *ISaGRAF*, but unfortunately could not be tested either.

3.5 FBench

The free *FBench* tool⁵ was developed by Cheng Pang at the University of Auckland. Figure 3.5 shows its workspace. It is very similar to FBDK as it provides an overview for selecting elements, a graphical view, and a textual view of the edited function block. Also, many modifications are done by using a context menu, and the possibilities to manipulate the graphical representation are exactly the same. The advantages of FBench in comparison to FBDK are that it provides a properties view to edit attributes and is able to compile created function blocks to executable Java code.

³<http://www.isagraf.com/index.htm>

⁴<http://www.nxtcontrol.com/>

⁵<http://www.ele.auckland.ac.nz/~vyatkin/fbench/index.html>

3 Related Work

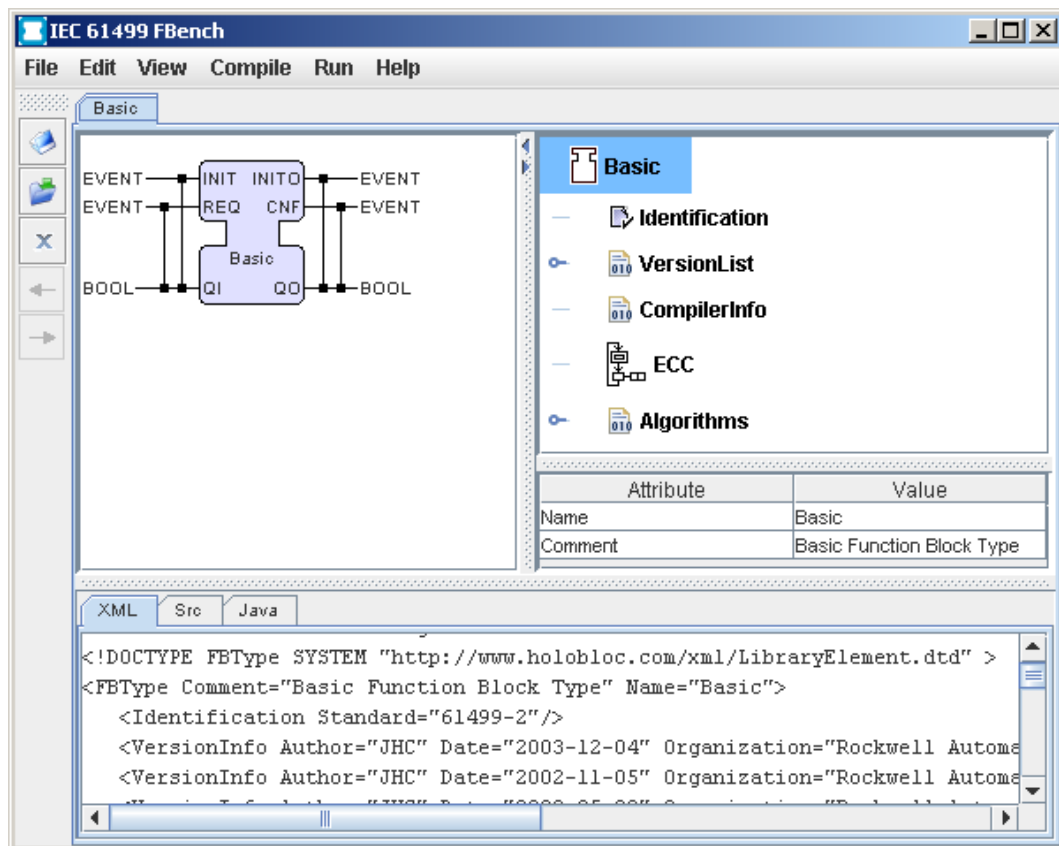


Figure 3.5: The workspace of FBench.

4 The Eclipse Project

The CAKeFEED function blocks editor is based on many different tools. The roles of all the projects are illustrated in Figure 4.1. Eclipse is an Integrated Development Environment (IDE) that is easily extensible and thus supports the development of additional plug-ins that further enhance its capabilities. One way of developing such plug-ins is to use the GMF. This is a framework that makes use of the EMF and the GEF. The EMF provides a means to specify models to develop software in a model-driven way. The GMF is a framework that encapsulates the *Draw2D* library and allows to comfortably implement graphical applications. The GMF employs models of the EMF, which describe systems like function blocks to generate Java code for a graphical diagram editor that allows to create and edit the systems described in the EMF models. *Xtend*, *Xpand*, and *Check* are languages developed by the Eclipse Modeling Project¹ and are used within the GMF to write model transformations and code templates and to check models for inconsistencies. The KIELER project eventually comprises the CAKeFEED function blocks editor together with several other editors and provides them with a number of enhancements that improve their capabilities.

The following sections give a short overview of the employed tools. For more detailed information see the study thesis [19] that precedes this diploma thesis.

4.1 Eclipse

Eclipse is not only the name of an IDE but also that of the open source community, which was founded by IBM in 2001. Since 2004 it is supported by the Eclipse Foundation, a not-for-profit organization that employs staff members who provide services to the open source developers of the Eclipse Project.

The main advantage that makes Eclipse so suitable for additional enhancements is its extensibility. Simply put, it consists only of a small kernel that can load and start other plug-ins. All additional functionality, such as the possibility to edit Java code in a special editor or to draw Unified Modeling Language (UML) [18] sequence diagrams, is added by loading plug-ins that provide the corresponding functionality. Additionally, any plug-in may define its own *extension points* that can be used by other plug-ins to employ services they cannot provide themselves. This way, it is possible to create custom IDEs for special purposes or even an all-round development platform that can cater to any need.

¹<http://www.eclipse.org/modeling/>

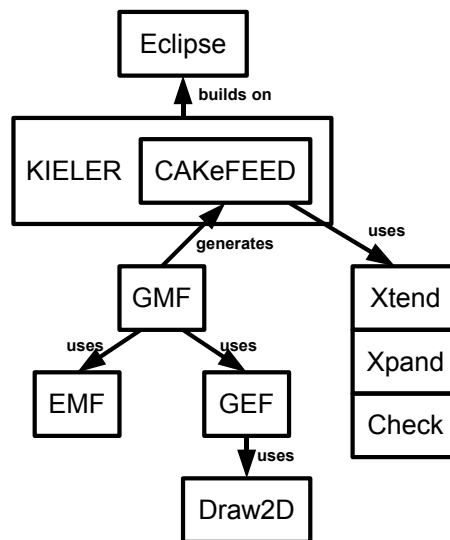


Figure 4.1: The roles of the different projects.

4.2 Eclipse Modeling Framework

The EMF² [22] allows its users to specify models of software systems that are to be implemented and is able to automatically generate source code based on those models. The so-called ecore models are very similar to UML class diagrams as they contain classes, aggregations and associations. These meta-models as well as instances of the meta-models are stored in XML files. Figure 4.2 depicts a small ecore model and Listing 4.1 shows its textual representation. For more information on the EMF refer to the study thesis [19].

4.3 Graphical Editing Framework and Draw2D

The GEF³ is a framework that provides a means to create graphical environments for Eclipse applications. It encapsulates Draw2Ds drawing capabilities in its more abstract *edit parts*, which serve as the interfaces between the figures in Draw2D, the EMF model, and the user. In the GEF all diagram elements drawn with Draw2D, which are also called *figures*, are represented by their own edit parts. Since figures in Draw2D might themselves contain other figures, it is also possible, and very common, for edit parts to contain other edit parts. In addition, edit parts offer a lot of functionality that figures do not: They support user interaction like clicking and dragging and their reactions on those user triggered events can be modified by using

²<http://www.eclipse.org/emf/>

³<http://www.eclipse.org/gef/>

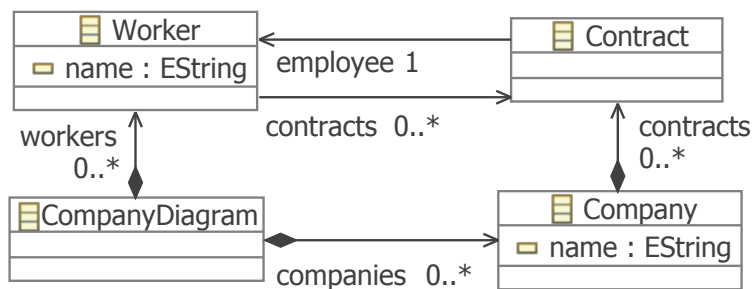


Figure 4.2: A simple.ecore model.

Listing 4.1: The textual representation of the.ecore model.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="company" nsURI="company" nsPrefix="company">
6   <eClassifiers xsi:type="ecore:EClass" name="CompanyDiagram">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="companies" upperBound="-1"
      eType="#//Company" containment="true"
      eOpposite="#//Company/diagram"/>
11   <eStructuralFeatures xsi:type="ecore:EReference"
      name="workers" upperBound="-1"
      eType="#//Worker" containment="true"
      eOpposite="#//Worker/diagram"/>
16  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Company">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="diagram" lowerBound="1"
      eType="#//CompanyDiagram"
      eOpposite="#//CompanyDiagram/companies"/>
21   <eStructuralFeatures xsi:type="ecore:EReference"
      name="contracts" upperBound="-1"
      eType="#//Contract" containment="true"
      eOpposite="#//Contract/employer"/>
26   <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name" lowerBound="1" eType="ecore:EDatatype
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
31  <eClassifiers xsi:type="ecore:EClass" name="Contract">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="employer" lowerBound="1"
      eType="#//Company" eOpposite="#//Company/contracts"/>
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="employee" lowerBound="1"
      eType="#//Worker" eOpposite="#//Worker/contracts"/>
36  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Worker">
    <eStructuralFeatures xsi:type="ecore:EReference"
      name="diagram" lowerBound="1"
      eType="#//CompanyDiagram"
      eOpposite="#//CompanyDiagram/workers"/>
41   <eStructuralFeatures xsi:type="ecore:EReference"
      name="contracts" upperBound="-1"
      eType="#//Contract" eOpposite="#//Contract/employee"/>
46   <eStructuralFeatures xsi:type="ecore:EAttribute"
      name="name" lowerBound="1" eType="ecore:EDatatype
      http://www.eclipse.org/emf/2002/Ecore#//EString"/>
  </eClassifiers>
</ecore:EPackage>

```

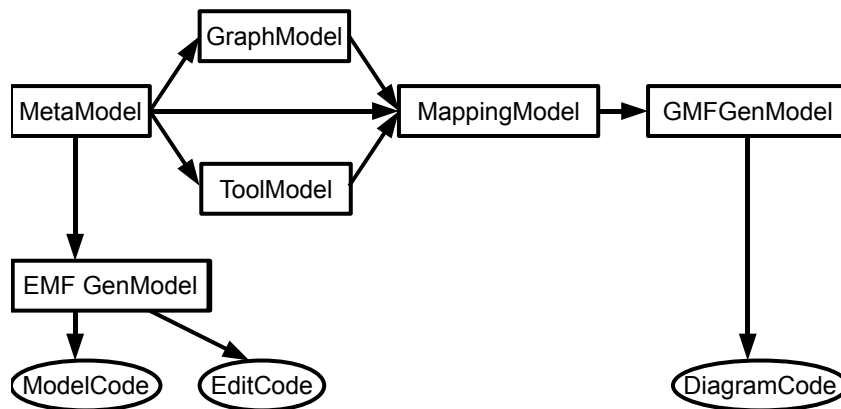


Figure 4.3: The complete GMF process.

edit policies. For more detailed information on the GEF and *Draw2D* see the study thesis [19].

4.4 Graphical Modeling Framework

Both the EMF and the GEF are employed by the GMF to generate graphical editors for specific visual languages. The EMF is used to create a meta-model of the elements that are to be described by the language. After the language and several other models have been specified, the GMF generates Java code based on the GEF and GMF runtime libraries that implements the graphical editor for the specified language. However, before the code can be generated there are some models that have to be created. These are discussed in the following sections. The complete process is depicted in Figure 4.3.

4.4.1 The Ecore Model

As has been mentioned above, the ecore model defines the entities that are to be created with the generated diagram editor. Simply put, it is like a UML class diagram with classes, associations, and aggregations.

The example shown in Figure 4.2 represents a company diagram in which the user can add companies and workers. Additionally, companies can employ workers with the help of contracts. It is possible for a company to have contracts with several workers and workers can also have contracts with more than one company. It is even possible for a pair of company and worker to have several different contracts. Note that the company diagram is the root whereas companies and workers are supposed to be represented as notes in the diagram and can be connected by contracts.

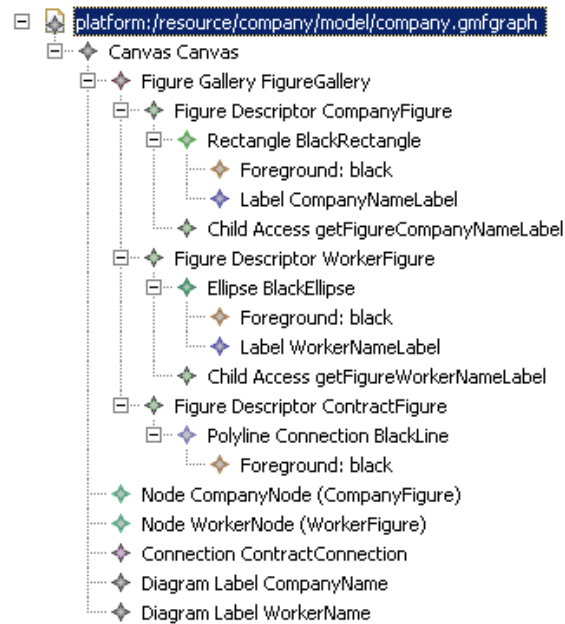


Figure 4.4: A graphical definition model of the company.

4.4.2 The Generator Model

The *generator model* looks very much like the ecore model. The difference between the two is that the generator model contains further information on the code generation, which may be altered by the user. For example it is possible to specify a different name for the generated project or another path for the source code. Once that is done, the generator model is used to generate the Java Code for the entities defined in the ecore model, which comprises the classes themselves along with getter and setter methods, a *package* and a *factory* to create instances of the objects. Additionally, the so-called *edit code* is created that encompasses *providers* for the modification of the model elements. It is also possible to automatically generate *test code* as well as a *tree editor* that is able to create and modify instances of the model elements.

4.4.3 The Graphical Definition Model

The *graphical definition model* is responsible for the visual representation of model elements in the diagram editor. The outline of an example that fits to the ecore model in the preceding section is depicted in Figure 4.4.

Here, the company is represented by a rectangle with a black outline while workers are depicted as ellipses with a black outline. Contracts connect companies and workers and are represented by black lines. The figures in the figure gallery are tied to nodes or connections, depending on whether they are supposed to be depicted as the former or the latter. Additionally, there are diagram labels that show the names

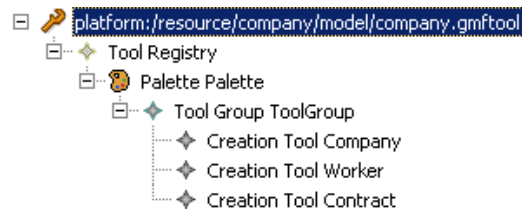


Figure 4.5: A tooling definition model to modify a company diagram.

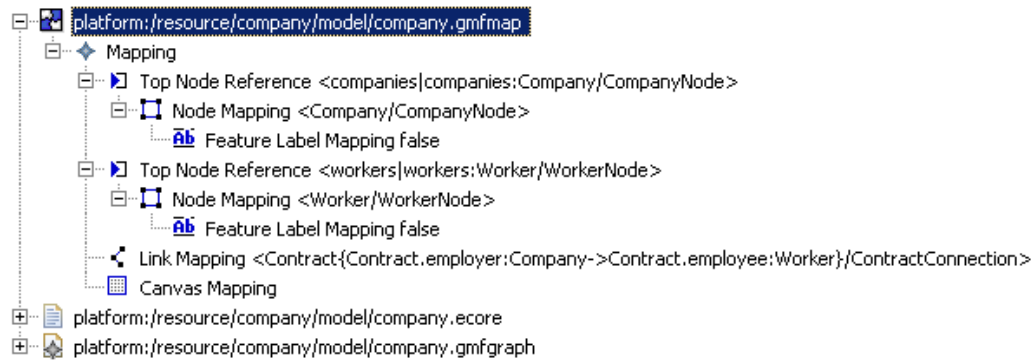


Figure 4.6: The mapping definition model for the company example.

of companies and workers. These are tied to the labels within the company rectangle and the worker ellipse, respectively. To access these inner labels it is necessary to define child accesses as seen in the figure.

4.4.4 The Tooling Definition Model

The tools that are provided to the user are defined in the *tooling definition model*. In Figure 4.5 the outline of a tooling definition model for the manipulation of the company diagram is depicted.

This model is very simple as it consists merely of a tool palette with one tool group. It contains three tools; one to create a new company, another one to create a new worker, and a third one to create a contract between a company and a worker.

4.4.5 The Mapping Definition Model

The *mapping definition model* is the one that ties the previous ones together. It assigns the graphical elements and tools defined in the graphical and tooling definition models to the model elements in the.ecore model. Also, it defines which model elements are the top level elements in the diagram, which are their children, and where those are located. The outline of the mapping definition model for the company example is shown in Figure 4.6.

Note that the canvas of the editor is mapped to the class `CompanyDiagram` and not

the class `Company` itself. This is done so that it is possible to depict the company as a top level node that is drawn onto the canvas. Workers also act as top level nodes. The class `Contract` is mapped to the corresponding polyline connection and the contract tool with the help of a *link mapping*. In addition, the company and worker top level node mappings contain feature label mappings that let them display their names.

4.4.6 The Diagram Editor Generator Model

When the mapping definition model is complete the *diagram editor generator model* can be generated. It augments the information from the former with additional information about the code that is to be generated. For example it is possible to change the name of the plug-in, and the path of the generated code, to enable model validation, and much more. However, the diagram editor generator model is much more complex than the previous ones and should only be modified by advanced users. Note that if one of the underlying models has to be changed, the diagram editor generator model has to be changed, too. Unfortunately, it is not yet possible to transfer the changes directly. Instead, the model has to be regenerated and all previous modifications have to be applied again. To remove this intermediate step, it is possible to define a Query/View/Transformation Operational (QVTO) [4] transformation that is automatically executed after the diagram editor generator model has been generated.

4.4.7 The generated Editor

Finally, the diagram editor generator model can be used to generate the code of the diagram editor. Figure 4.7 shows what the result of the example looks like.

The editor provides a canvas and tools to draw companies, workers, and contracts between them. It is also possible to give each company and worker a name when they are created. Additionally, there are many features which the GMF provides for free: drag-and-drop editing, context menus, collapsible compartments and tool groups, an outline, undo and redo, saving and loading wizards, a menu bar, and much more. Note that the worker names look a little out of place. This is due to the fact that no layout for the ellipse figures has been specified. In a more elaborate editor this is very well possible as well as defining insets for the labels, drawing dashed lines, coloring figures, and much more.

4.5 Xtend

Xtend is one of several languages provided by the Eclipse Modeling Project. It is a functional language with which it is possible to write transformations that can be understood and executed by an `XtendFacade` to transform model elements. For the remainder of this thesis it is only necessary to understand the commands `import` and `create`. The former is used to import an ecore model by using its namespace prefix. For example if there is an ecore model with the namespace prefix `example`, the line

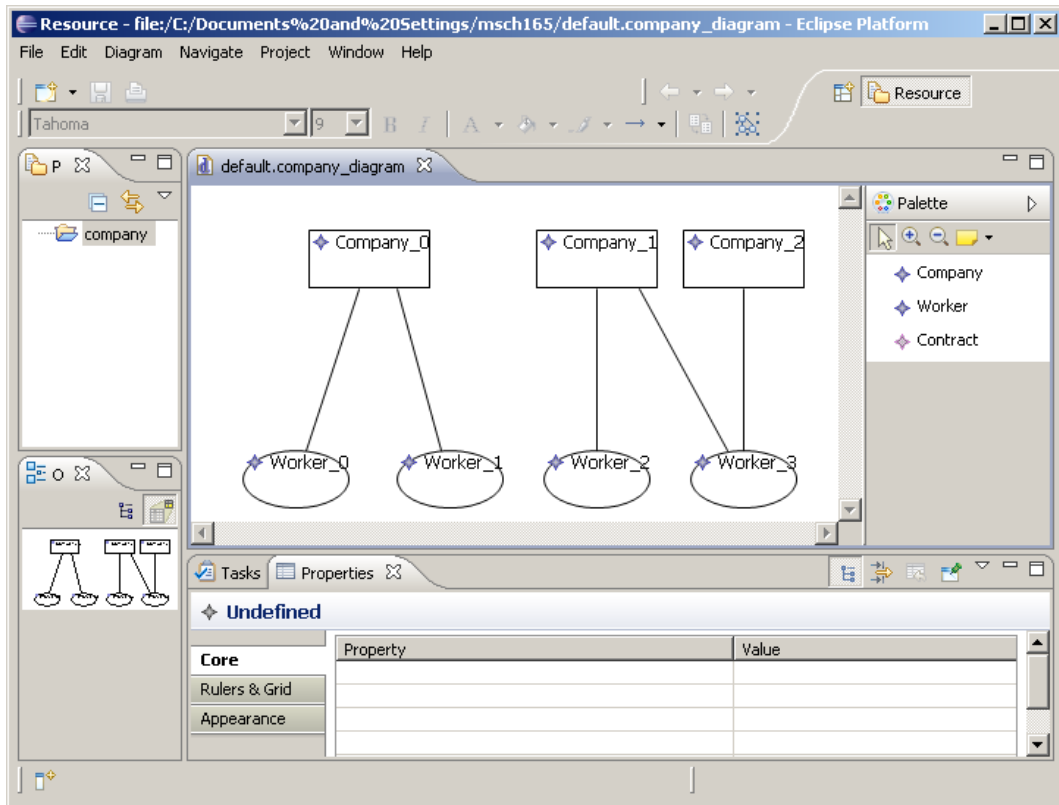


Figure 4.7: The finished Company Diagram Editor.

`import example;` imports that ecore model. The `create` command is a convenience operation to create new model elements. Listing 4.2 shows a simple example.

Listing 4.2: A simple transformation.

```

import company;
import enterprise ;

5 create enterprise :: EnterpriseDiagram this companyDiagramToEnterpriseDiagram
   (company::CompanyDiagram input)
   this . enterprises .addAll(input . companies .CompanyToEnterprise()) ->
   this . employees .addAll(input . workers .WorkerToEmployee())
;

10 create enterprise :: Enterprise this companyToEnterprise
   (company::Company input):
   this . setName(input . Name) ->
   this . contracts .addAll(input . contracts .ContractToContract())
;

15 create enterprise :: Employee this workerToEmployee(company::Worker input)
   this . setName(input . Name)
;

20 create enterprise :: Contract this contractToContract(company::Contract input)
   this . setEmployee(input . employee .WorkerToEmployee())
;

```

The first step is to import all needed ecore models. Suppose there are two models with the namespaces `company` and `enterprise`. The first one is the model from the example above and the second one is like the first with the only exceptions that the class `CompanyDiagram` is replaced by the class `EnterpriseDiagram`, the class `Company` is replaced by the class `Enterprise`, and the class `Worker` is replaced by the class `Employee`. The first transformation, also called *extension* in *Xtend* terms, takes a company diagram as an argument and creates an enterprise diagram, which is returned upon completion. Additionally, the lists of companies and workers are replaced by lists of enterprises and employees, which result from calling the extensions `companyToEnterprise` and `workerToEmployee`, respectively. Note that the extension is called for every company and every worker in the company diagram, since the two lists are implicitly being iterated through. As can be seen, there is a compact syntax for doing this because it is a very common case. The extension `companyToEnterprise` creates an enterprise from a company by replacing its contracts with new contracts from the enterprise model. Note that although both classes have the same name and attributes, they are entirely different. The extension `workerToEmployee` in turn takes a worker as argument and creates a new employee with the same name. Finally, a company contract is translated into an enterprise contract by replacing its worker with an employee. Thus, it is possible to transform a whole company to an enterprise by simply executing the extension `companyToEnterprise` with a company as its argument.

4.6 Xpand

Another language developed by the Eclipse Modeling Project is *Xpand*. It is used to define code templates which the GMF uses to generate the diagram editor code. On some occasions it is necessary to alter the code templates in order to augment the

diagram editor with additional features. The basic keywords are explained with the help of the example depicted in Listing 4.3.

Listing 4.3: A simple code template

```

«IMPORT 'http://www.eclipse.org/gmf/2009/GenModel' »
3 «DEFINE additions FOR company::Company—»
  I am the Company named «self.Name»
  «FOREACH getContracts(self) AS contract»
    I have a contract with «contract.getEmployee().getName()»
  «ENDFOREACH»
8 «ENDDDEFINE»

```

The `IMPORT` statement is used to import *data models*, which are derived from ecore models. With the `EXTENSION` statement it is possible to import QVTO transformations and reuse them in the template. The `DEFINE . . . FOR` statement is used to define a new code template rule for the specified node in the data model. In order to invoke other template rules the `EXPAND` statement is used. Finally, there are other statements like `IF`, `ELSEIF`, `FOREACH` and others of which the meaning is intuitive.

In the example above, there is a template rule, which specifies that for every company the code 'I am the Company named <CompanyName>' should be generated. In addition, for every contract of the company the code 'I have a contract with <WorkerName>' is generated.

4.7 Check

The final language provided by the Eclipse Modeling Project that is important for this thesis is the *Check* language. With Check it is possible to check models for inconsistencies. The language is very compact and can be explained with the short example shown in Listing 4.4.

Listing 4.4: A check file for the company example.

```

import company;
2
context Company WARNING "This company has no name!" :
  ( this . name != null ) && ( this.name.length > 0 )
;

7
context Worker WARNING "This worker has no name!" :
  ( this . name != null ) && ( this.name.length > 0 )
;

12
context Company WARNING "This company has no contracts!"
  this . contracts . size > 0
;

```

Again, the import statement is used to import ecore models by using their namespace prefixes. A check rule begins with the keyword `context`. The name of the class that follows this keyword defines which element in the diagram is to be marked as inconsistent in the case of that rule being active. It might be followed by an additional `if` that further restricts the scope of the rule. Then the `ERROR` statements initiates the error message that is to be displayed if the rule is active. Alternatively, it might be replaced by a `WARNING` statement that indicates that the inconsistency produces a warning rather than an error. Finally, the check rule is concluded by an

expression that is to be true for the rule to not become active. In the example above all companies and all workers are required to have a name. Also, every company has to have signed at least one contract, otherwise a warning message is shown. How these rules can be checked in the diagram editor or be invoked programmatically is discussed in the study thesis [19].

4.8 The KIELER Project

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)⁴ is a project that combines all of the aforementioned tools to provide graphical and textual editors that have been augmented with a number of additional features to further improve their usability. As has been pointed out by Hauke Fuhrmann and Reinhard von Hanxleden [12], graphical editors can be greatly improved in terms of usability and efficiency when techniques like automatic layout and structure-based editing are employed.

The original idea of the KIELER project was, as described by Spönemann *et al.* [21], to enhance model-based design by providing automatic layout mechanisms and thus to improve the efficiency of the development process. Over time, the KIELER project has grown into a wide collection of subprojects, which all have the goal to make model-based design better. Some of these projects were also used in this thesis to improve the CAKeFEED function blocks editor. One of those is the KIELER Structure-Based Editing (KSBasE) [16] project by Michael Matzen, which adds structure-based editing mechanisms to any editor created with the GMF. The other one is the KIELER Execution Manager (KIEM) [17] by Christian Motika that provides a means to simulate models created with GMF editors. In addition, there are other projects such as the KIELER *View Management* [7] project by Nils Beckel and the KIELER *Textual Editing Framework* [6] project by Özgün Bayramoğlu. The View Management project allows to automatically invoke functions that improve the readability of diagrams, such as zoom or fade in and fade out effects. The Textual Editing Framework uses *Xtext* [11] from the Textual Modeling Framework (TMF) to provide an exemplary textual editor for *SyncCharts*.

Like the other projects mentioned before, CAKeFEED is a part of the *KIELER* project and as such benefits from its other subprojects.

⁴<https://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/WikiStart>

4 The Eclipse Project

5 Concept

This chapter presents the problems that had to be addressed during the development of the CAKeFEED function blocks editor and one or several possible solutions for each. Section 5.1 discusses three different approaches on how to model function blocks. In Section 5.2 the structure of the editor is explained that results from the decisions documented in the preceding section. Section 5.3 discusses how attribute and type awareness and the layout of ports have been dealt with, while Section 5.4 is about the simulation of function blocks. Section 5.5 presents the translation from the XML Function Block format to the XML format that the editor uses and vice versa.

5.1 Modeling approaches

Three different approaches to modeling function blocks have been identified in the course of this thesis. They all take the existence of the XML Format into account, which is specified in the IEC 61499 function blocks standard [3] as it is a requirement for the editor to be able to read and write that format. Since any editor created with the GMF saves documents in XML by default, the first possible approach discussed in Section 5.1.1 is to make the editor use exactly the same format as is specified in the standard. Another possible approach is to provide an import and export mechanism to translate the XML format of the editor into the one specified in the standard and vice versa. This one is discussed in Section 5.1.2. The approach that has finally been employed is explained in Section 5.1.3. It is a compromise between the two aforementioned approaches.

5.1.1 The XML-conformant Meta-Model

As has been mentioned above, there is an XML format for function blocks defined in the function blocks standard [3]. It is possible to translate that format to an ecore model. The advantage is that instances of the resulting model are saved in exactly the same format as specified in the XSD, which means there is no need to implement a mechanism to import and export function blocks written in the XML format. However, there are some shortcomings to this approach:

Intermediate Containers On the one hand, the XML format often uses intermediate containers to distinguish model elements that are of the same class but have different purposes. Listing 5.1 illustrates this.

5 Concept

Listing 5.1: Intermediate containers in the function blocks XML format.

```
2 <?xml version="1.0" encoding="UTF-8"?>
  <!DOCTYPE FBType SYSTEM "http://www.holobloc.com/xml/LibraryElement.dtd" >
  <FBType Name="FBAND" Comment="Boolean AND" >
    ...
    <InterfaceList >
      <EventInputs >
7        <Event Name="REQ" >
          <With Var="IN1" />
          <With Var="IN2" />
        </Event >
      </EventInputs >
12     <EventOutputs >
        <Event Name="CNF" >
          <With Var="OUT" />
        </Event >
      </EventOutputs >
17     <InputVars >
        <VarDeclaration Name="IN1" Type="BOOL" />
        <VarDeclaration Name="IN2" Type="BOOL" />
      </InputVars >
      <OutputVars >
22     <VarDeclaration Name="OUT" Type="BOOL" Comment="Result" />
      </OutputVars >
    </InterfaceList >
    <BasicFB >
      ...
27 </BasicFB >
  </FBType >
```

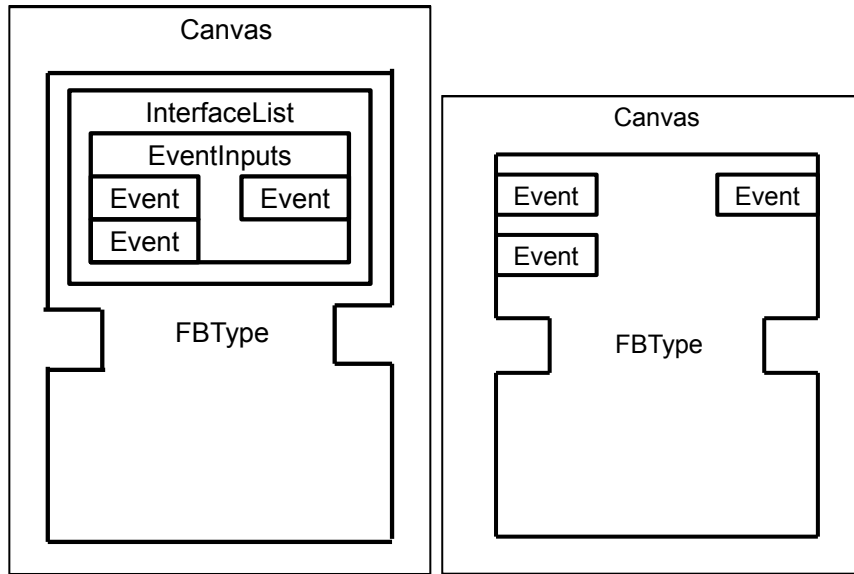
As shown in the figure, input events and output events are both of the same class called `Event`. To distinguish both kinds of events, they are put into intermediate containers, which are either called `EventInputs` or `EventOutputs`. In addition, these two intermediate containers are contained in another intermediate container called `InterfaceList`, which, while making the purpose of the following elements clearer to the reader, serve no semantical purpose at all and could as well be left out. The same function block could be defined by using the two different classes of events `InputEvent` and `OutputEvent`, which are direct children of the `FBType` element. This is shown in Listing 5.2.

Listing 5.2: A function blocks XML format without intermediate containers.

```
2 <FBType Name="FBAND" Comment="Boolean AND" >
  <InputEvent Name="REQ" >
    <With Var="IN1">
    <With Var="IN2">
  </InputEvent >
7  <OutputEvent Name="CNF" >
    <With Var="OUT">
  </OutputEvent >
  <InputVar Name="IN1" Type="BOOL" />
  <InputVar Name="IN2" Type="BOOL" />
  <OutputVar Name="OUT" Type="BOOL" />
12 <BasicFB >
  ...
  </BasicFB >
</FBType >
```

The intermediate containers have implications on the diagram editor. Since child references in mapping models are not allowed to leave out intermediate containers, each one would have to be represented in the diagram, which means that the user has to create them manually. So, instead of simply adding an `InputEvent` to a function block, the user has to add an `InterfaceList` and an `EventInputs` container before finally adding the `Event`.

Fortunately, this usability issue can be remedied by employing the `KSBasE` [16] project in the editor. This project allows to define transformations for any model



(a) Adding an input event to a function block with intermediate containers. (b) Adding an input event to a function block without intermediate containers.

Figure 5.1: Adding input events.

element depicted in a diagram. For example, it is possible to define a transformation `AddInputSignal` that automatically adds an `InterfaceList` and an `EventInputs` container when needed before adding the `Event`. Such a transformation may be invoked either by using the context menu, the appropriate entry in the menu bar, or a keyboard shortcut. The last problem to address is that elements added by using KSBasE would have to be laid out, as they are usually placed in an arbitrary position inside their parent container. However, this is possible with automatic layouts as demonstrated in [20].

String References On the other hand, all references in the resulting ecore model, like events referenced in the `With` elements or function block types referenced in `FB` elements, are of the type `String`, where the string contains the name of the referenced element. Note that all function block types are stored in a single file. So this method of referencing only works because all function block types are supposed to have a unique name and to be stored in a file of the same name. In the EMF references are represented by an XML Path Language (XPath) [1] expression as shown in Listing 5.3.

While references to elements in the same file may be changed to an arbitrary format, this is not possible for references that refer to elements in a different file. In that case the reference has to have the format `<resource>#<fragment>` where `<resource>` and `<fragment>` can be freely changed but the `#` has to remain in any case. Thus, it is not possible to replace the XPath reference by a simple name. However, there is one last option to modify the XML format as needed that has been

Listing 5.3: A string reference compared to an XPath expression.

```

5 <fBs name="controller">
  <type xsi:type="cakefeed:BasicFunctionBlockType" href="BFBType"/>
  ...
</fBs>

<fBs name="controller">
  <type xsi:type="cakefeed:BasicFunctionBlockType" href="default.bfbtype#//@bFBType"/>
  ...
</fBs>

```

used in the *KIELER Infrastructure for Textual Modeling* project. There, the textual format of *SyncCharts* is replaced by a new grammar written in *Xtext* [11]. With *Xtext* it is possible to specify a grammar that imitates the XML syntax and provides keywords for all needed elements like `FBType`, `Event`, and so on. However, writing a new *Xtext* grammar is a lot of work and probably more than implementing an import and export mechanism with an *Xtend* transformation that transforms XPath references to strings. As the goal of this approach was to save the additional work of writing a transformation for the import and export of function blocks, replacing this by the work of writing an *Xtext* grammar is not a desirable solution.

5.1.2 The custom Meta-Model

The previous section showed that it is not possible to simply use the ecore model that results from the XSD without further work like creating an *Xtext* grammar or an *Xtend* transformation. Due to these circumstances, another approach comes to mind that involves creating a custom ecore model that is tailored to the needs of the GMF and providing an *Xtend* transformation to translate instances of this model to the function blocks XML format and vice versa. In addition, this model could be created in such a way that it has minimal complexity to make it as accessible as possible. What such a model could look like is depicted in Figure 5.2.

As can be seen, a lot of classes have been left out so that only the ones are included that are important for function block semantics. Also, since events, variables, and connections are represented by only one class each, it is possible in the resulting diagram editor to connect input events with output variables for example and to do other things that make the diagram invalid. However, these violations can easily be checked against and marked as errors with the Check language. This has been discussed in the study thesis [19]. Another way of preventing these violations would be to use Object Constraint Language (OCL) constraints. The main difference between the original and the new ecore model is hidden in the class `FunctionBlock`: The model does not discern function blocks and function block types like the old one. Instead, there is only one class `FunctionBlock` that formally acts as both as every function block is its own function block type. This means that whenever there are two equal function blocks they have two different function block types. More importantly, it is not possible to define one type and assign it to several function blocks as is the case in the original model. To remedy this, a library mechanism has to be implemented

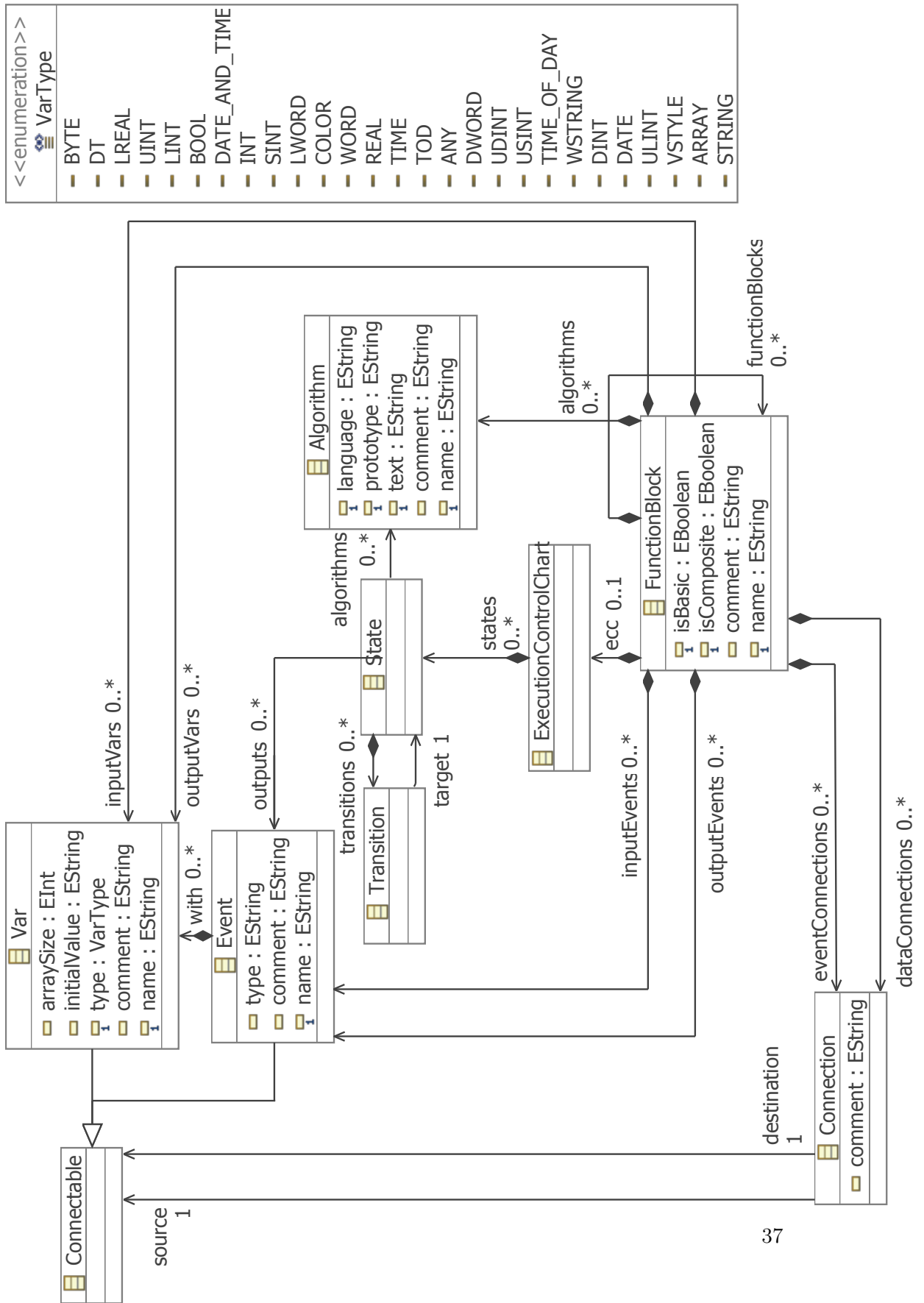


Figure 5.2: A custom ecore model for function blocks.

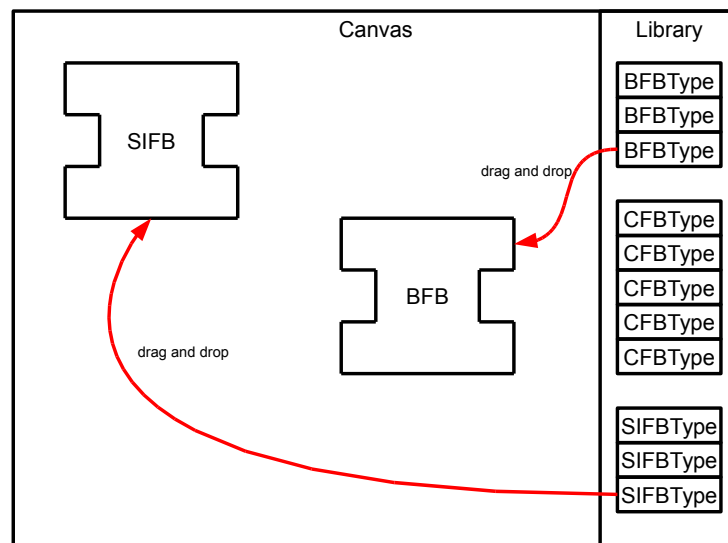


Figure 5.3: A library mechanism for the editor could work like this.

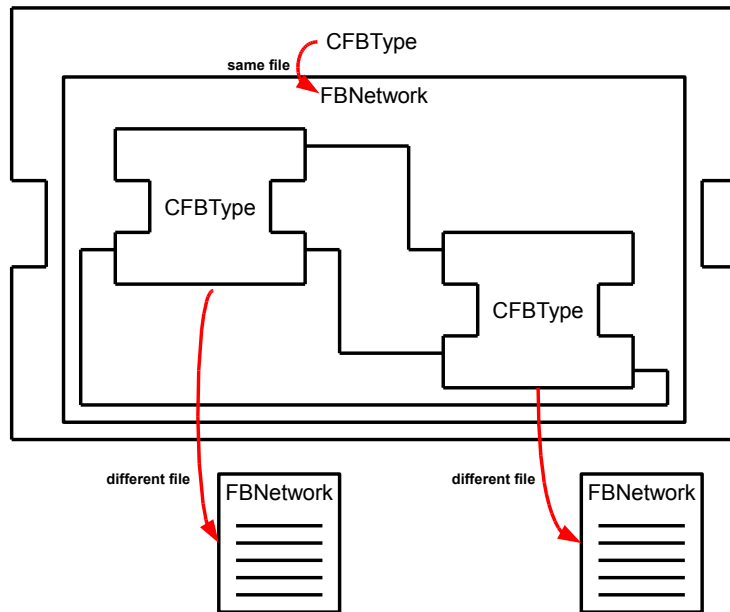
that allows to store already created function blocks in a collection that resembles a library of function block types and to drag any type to the diagram canvas to create a new instance of the type. As can be imagined, this is not a trivial task. Figure 5.3 depicts an example for a library mechanism.

Another disadvantage of the new model is that the transformation that transforms its instances to the XML format has to be much more complex. The main difficulties result from the fact that a function block may contain an arbitrary number of other function blocks. This implies that there might be an arbitrary number of hierarchy levels within one function block diagram whereas in the XML format every function block may only contain one further level of function blocks, namely inside of its function block network. All of those further function blocks are black boxes that refer to their type, which is stored in another file, instead of showing their inner structure directly. Figure 5.4 illustrates this.

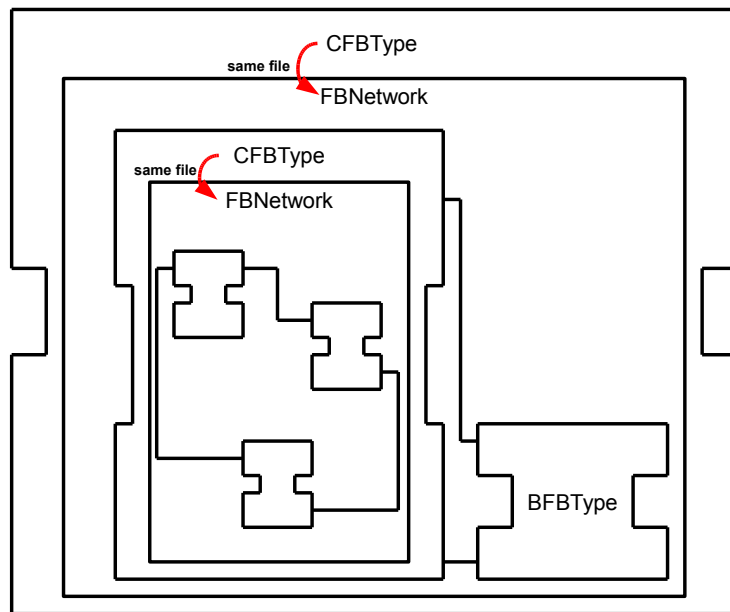
This implies that the transformation has to get rid of the exceeding hierarchy levels by storing them in function blocks contained in different files. Also, hierarchy levels with the same structure are not supposed to be stored to different files as they represent the same function block types. As function blocks may have an arbitrary number of hierarchy levels it is a costly task to determine whether a hierarchy level is already included in a separate function block or not.

5.1.3 The hybrid Meta-Model

During the course of this thesis it was decided that an editor is desired where there is not an arbitrary number of hierarchy levels, but where function blocks merely



(a) Only one hierarchy level in the old ecore model.



(b) Several hierarchy levels in the custom ecore model.

Figure 5.4: Hierarchy levels in the old and the new ecore model.

point to function block types that specify their inner structures. This is already the case in the ecore model resulting from the XSD. Also, many of the classes that are defined in that meta-model but have been omitted in the custom meta-model have been found to be useful for the specification of function blocks. Some of those classes are `Identification` and `VersionInfo` for example. However, as has been discussed before, that meta-model is not suitable for an editor created with the GMF: If no custom Xtext grammar is employed then there has to be an Xtend transformation in any case. This leads to a third possible approach, which uses a more elaborate Xtend transformation and a meta-model that originates from the one resulting from the XSD but that has been modified to better suit the needs of the GMF. The changes that have been made to the meta-model are discussed in the following paragraphs.

Replace String Attributes with Object References In the GMF references are required to be of an object type so that only proper editing possibilities are provided to the user, such as adding an input event to a function block or connecting an output event to an input event. If those references are mere string attributes in the ecore model, the GMF can neither provide graphical editing mechanisms such as drawing a connection from one point to the other, nor check if a user action is invalid like connecting an event to a variable. This means that every string attribute, which is used as a reference in the original ecore model such as the type attribute of a function block, has to be replaced by a reference with the proper type, for example a type reference with the type `FunctionBlockType`. Of course, these references have to be changed back by the Xtend transition when exporting a model.

Remove intermediate Containers Where possible, intermediate containers like the classes `InterfaceList` or `EventInputs` have been removed to make editing more comfortable. These containers can easily be inserted by the *Xtend* transformation.

Add Subclasses On some occasions, subclasses were added to improve the usability of the editor. One example, as shown in Figure 5.5, is the addition of the classes `InputEvent` and `OutputEvent` that are subclasses of `Event`. These two classes serve to distinguish input events from output events. The same is done for variables. Further, the classes `InputWith` and `OutputWith` are added that extend the class `With`, which models associations. `InputEvents` may only aggregate `InputWiths` while `OutputEvents` may only aggregate `OutputWiths`. By giving the referenced variable in the class `InputWith` the type `InputVariable` and the referenced variable in the class `OutputWith` the type `OutputVariable`, it is no longer possible for the user to associate an input event with an output variable and vice versa. Although this could also be done by employing the Check language, it is more comfortable because this way it is not possible for the diagram to enter an invalid state whereas with Check invalid states can be entered but are then marked as erroneous.

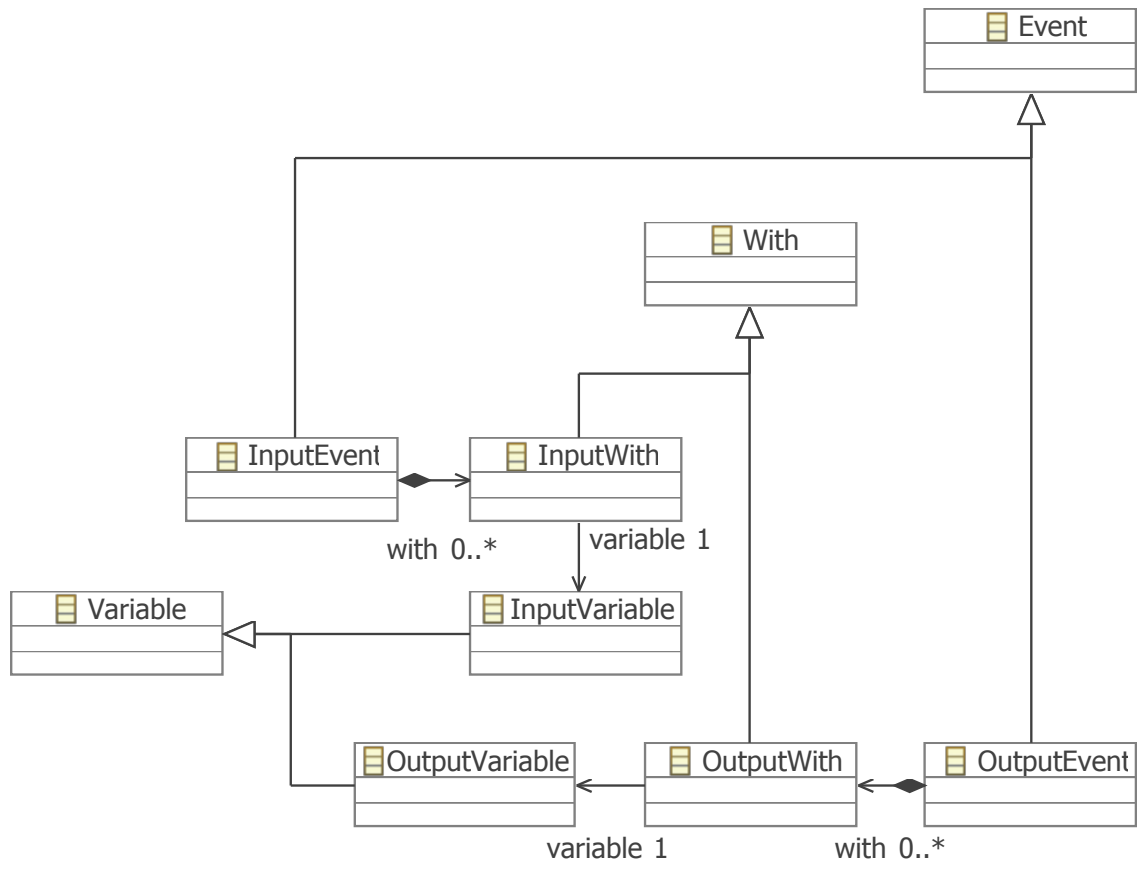


Figure 5.5: An extract of the hybrid meta-model.

5.1.4 The remaining models

The remaining models including the graphical definition model, the tooling definition model, and the mapping definition model are modeled in the usual way as described in the study thesis [19]. The only specialty is that the graphical definition model contains custom figures that implement attribute awareness. These are discussed in detail in Section 5.3.1

5.2 The Structure of the Editor

The decision to use the hybrid meta-model had several implications for the nature of the editor. The four most important aspects are discussed in the following sections.

5.2.1 Hierarchy Levels

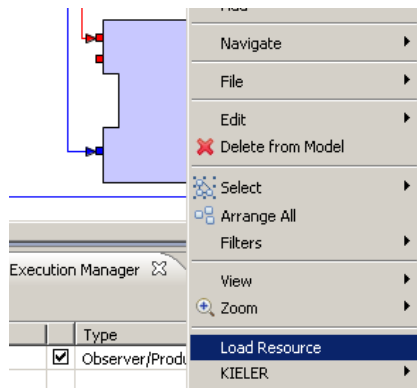
As has been mentioned before, the inner structure of function blocks is defined by their types. This means that they do not reveal their inner structure but rather are black boxes while their inner structure is defined in the diagram of the function block type. Thus, every diagram has always only one hierarchy level. Although the use of an arbitrary number of hierarchy levels also has its advantages, the restriction to one level is necessary to ensure compatibility to the IEC 61499 XML format.

5.2.2 Separation of Concerns

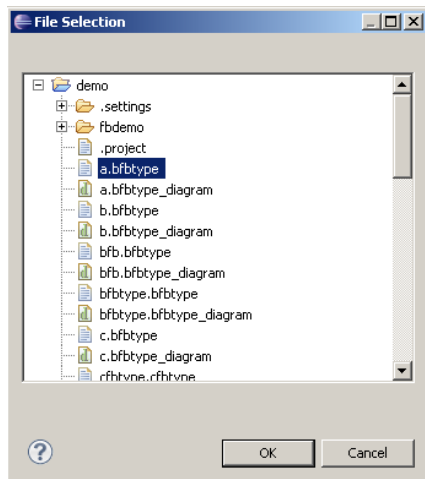
Since every function block is edited in its own diagram, other function blocks that are not relevant to the one currently being edited cannot obstruct the view. Thus, the focus always remains on the relevant parts of the system being created. However, as with the previous point, the *View Management for Visual Modeling* [7] project would be useful to improve usability when not every function block has its own diagram as is the case in the custom meta-model.

5.2.3 Reuse of Elements

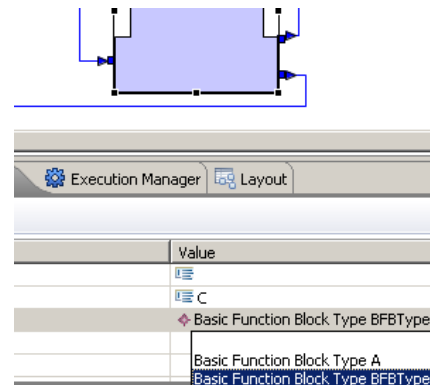
By employing the hybrid meta-model the need for a newly implemented library mechanism is reduced. While it would enhance usability, it is sufficient to use a mechanism that is built-in in every GMF editor: As depicted in Figure 5.6, it is possible to load other function block types into the one currently edited by right-clicking on the diagram canvas, selecting **Load Resource** from the context menu and selecting any files that are to be loaded. While they are not visible in the diagram editor, the instances defined in the loaded files can then be selected as the type of any function block in the diagram. This mechanism is very similar to the one employed by the FBDK with the only exception that types are chosen after the creation of a function block instead of before.



(a) Select Load Resource.



(b) Choose the file to load.



(c) Set the type attribute.

Figure 5.6: Loading a resource.

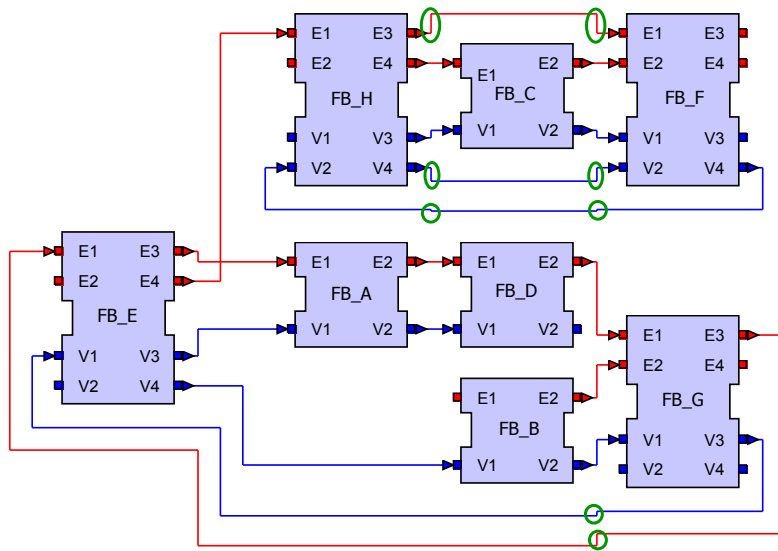
5.2.4 CAKeFEED and KIELER

The nature of the meta-model that was used also determines how much the editor can benefit from the features provided by the KIELER project. The hybrid meta-model is designed in such a way that on the one hand it does not rely on any feature offered by the KIELER project but on the other hand can still benefit from any of them. The use of only one hierarchy level and the separation of concerns reduce the need for automatic layout and view management which otherwise would have been direly needed. Still, the employment of those two features can greatly improve the usability of the editor as single levels of hierarchy also benefit from automatic layout and large function block networks can make good use of view management. If the XML-conformant meta-model was employed, the use of structure-based editing would have been compulsory. However, the editor resulting from the hybrid meta-model can also benefit from it since especially the addition of events and variables with the help of drag-and-drop editing can be time consuming. To sum it up, it might be worth to try and create a function blocks editor based on another of the proposed meta-models in the future, especially if closer ties to the KIELER project are desired.

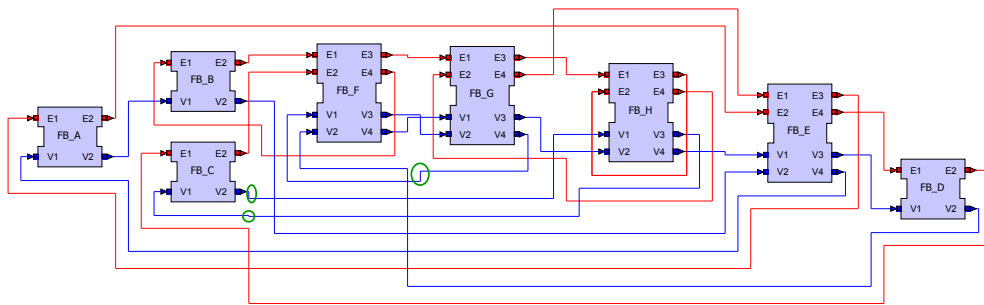
Automatic Layout While view management and structure-based editing are not used in the CAKeFEED editor yet, automatic layout is already employed. Since function block diagrams are dataflow diagrams, the best results can be achieved with Miro Spönemann's hierarchical dataflow layout [20]. As can be seen in Figure 5.7(a), it provides very clean layouts in the CAKeFEED editor. Even diagrams with many crossovers of connections like the one shown in Figure 5.7(b) yield good results. Since it is not uncommon for function blocks to have many ports, also a diagram with lots of such function blocks has been tested. Again, the result looks very clean, as can be seen in Figure 5.7(c). However, sometimes the layouts have small flaws as some connections have more bend points than would be needed. These incidents are marked by green circles in the figures. Also, sometimes the routing of connections is not optimal and higher complexity often comes with larger unused areas.

5.3 Code Modifications

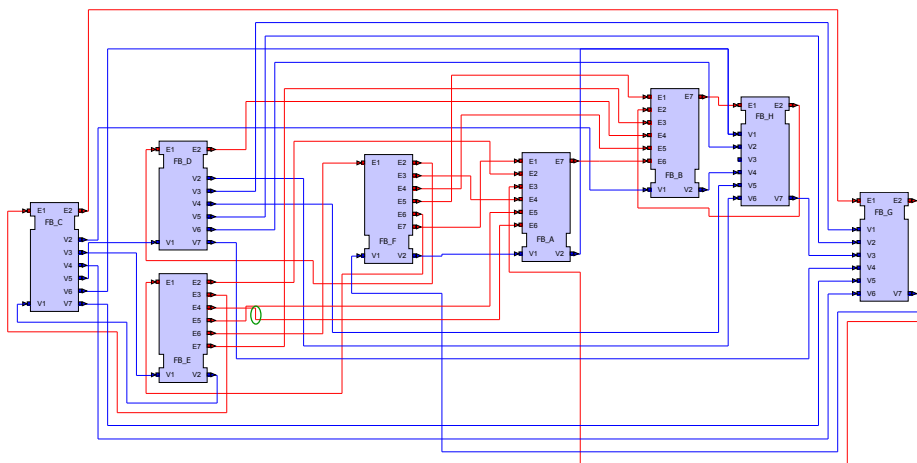
Although the function blocks editor that is generated from the hybrid meta-model and its accompanying other models already provides a great deal of the needed functionality, there is still one important part missing along with some features that improve its usability. These have to be implemented by manually modifying the generated code. While these manual modifications are sufficient to enhance the editor, the Xpand language was used to embed them in the code generation templates so that they are applied automatically every time the code is generated. How this is done is explained in the study thesis [19]. The following sections describe the manually implemented attribute and type awareness of function blocks and the layout of event and variable ports.



(a) A function block diagram with hierarchical dataflow layout.



(b) A layouted diagram with many connection crossovers.



(c) A layouted diagram with many function blocks that have lots of ports.

5.3.1 Attribute and Type Awareness

Due to the built-in Load Resource mechanism that comes with any GMF editor it is possible to set the type attribute of a function block to any function block type defined in another file. However, the change of the attribute alone has no effect on the figure of the function block depicted in the diagram. The reason for this is that the structure of edit parts as well as figures is supposed to be static. This means that the events and variables of a function block are supposed to be permanent and not to be removed or substituted. This problem is addressed by enhancing the behavior of the edit parts, which form the interfaces between the user and the figures. They are modified so that they send notifications to the figures whenever the user has changed the type of a function block.

5.3.2 Port Layout

The function block standard [2] specifies certain positions for different ports. While events have to be located at the upper part of the function block, variables are supposed to be located at the lower part. This issue has also been addressed in the editor. Usually, the layout of a figure's children is determined by the layout of the parent figure. This is also how the children of state figures are laid out in the study thesis [19]. However, with ports it is slightly different. Although the port figures formally count as children of the function block figures, they are not located inside that figure like usual children. Instead, they appear at the border on the outside of the figure. Due to these circumstances, layouts do not affect them and their locations are determined by *locators*. Whenever the user tries to change the position of a port figure, the assigned locator returns a set of valid locations in which the figure might be placed. In the CAKeFEED editor, the set of valid locations calculated by the default locator is restricted by another specialized locator to ensure that all ports are located in the right places. The advantage of locators compared to usual layouts is that the user can still change the position of events and variables within the bounds that the locators provide while layouts always assign one fixed location to an element that cannot be changed by the user.

5.4 Simulation

In the course of the thesis it was decided that the function blocks editor should be augmented with a feature that provides a means to simulate counter examples of a function block network. Such a counter example is simply a sequence of event and ECC state activations that leads to an erroneous state. The KIEM [17] is used to accomplish this goal.

5.4.1 The KIELER Execution Manager

The KIEM provides an infrastructure for the simulation of domain specific models. With its help it is possible to simulate function blocks behavior by implementing pro-

ducers and observers that produce or observe simulation data, respectively. Hereby, the KIEM merely plays the role of a data bus that connects the producers and observers, which are also called *data components*. The data components use the KIEM to communicate with each other. If a data component produces data that can be reused by other components, it is called a producer, while a data component that only reads data from other data components is called an observer. In addition, it is possible for data components to be both a producer and an observer or even neither of the two. The latter is the case if the data component only performs some tasks in its initialization, but makes no use of the data bus during the simulation. In the case of the function blocks editor only one data component is needed that is neither a producer nor an observer as it merely loads the counter example during initialization and displays active events and transitions during simulation.

5.5 Import and Export

As discussed before, the function blocks editor needs to be augmented with an import and export mechanism that allows to translate function blocks created with the editor into the XML format defined in the standard [3] and vice versa. This is done with the help of the transformation framework, which is a product of the KSBasE [16] project. Several transformations have been developed, which can translate function blocks in the IEC 61499 function blocks format into the XML format that is used by the GMF editor, and also transform function blocks in the GMF XML format into the IEC 61499 format. This enables the user to import function blocks into the GMF editor that have been created with other tools such as FBDK or FBench and to export those created with the GMF editor for further processing in other tools that understand the IEC 61499 function blocks format.

5 *Concept*

6 Implementation and Results

The implementations of the concepts discussed in Chapter 5 are explained in detail in this chapter. Section 6.1 deals with the models that are used to generate the editor with the GMF. Section 6.2 explains how the attribute and type awareness and the layout of ports were implemented. The implementation of the simulation feature is discussed in Section 6.3 while the implementation of the import and export feature is explained in Section 6.4. Finally, Section 6.5 gives a short evaluation of the employed approaches.

6.1 Modeling

In this section, the different models are explained that are used to generate the function block editor. These are the meta-model, the graphical definition model, the tooling definition model, and the mapping definition model. The generator model as well as the diagram editor generator model are omitted as they are automatically generated and the few changes made to them are of a trivial nature.

6.1.1 The Meta-Model

The hybrid meta-model is derived from the XSD and as such contains a wide range of classes that help define function blocks. Note that these classes are intended to be used by the generated EMF tree editor, which is used as an overview of the currently edited function block. In the overview it is possible to add all those elements including compiler information, version information, and more. In addition, it is possible to define whole *systems* with *devices* and *resources* in which function block networks can be embedded that have been developed with the GMF diagram editor. Figure 6.1 illustrates this.

Since the meta-model is very large, it is not explained in its entirety and only the parts that need a deeper understanding are discussed. The three main classes that are important for the diagram editor are `BasicFunctionBlockTypeDiagram`, `CompositeFunctionBlockTypeDiagram`, and `FunctionBlockNetwork`. These three each make up the canvas of either the basic function block type editor, the composite function block type editor, or the function block network editor. Figure 6.2 shows the relations between those classes. Note that some aspects have been simplified or omitted to keep the diagram simple.

BasicFunctionBlockTypeDiagram The class `BasicFunctionBlockTypeDiagram` contains the `BasicFunctionBlockType`, which inherits several attributes from its super

6 Implementation and Results

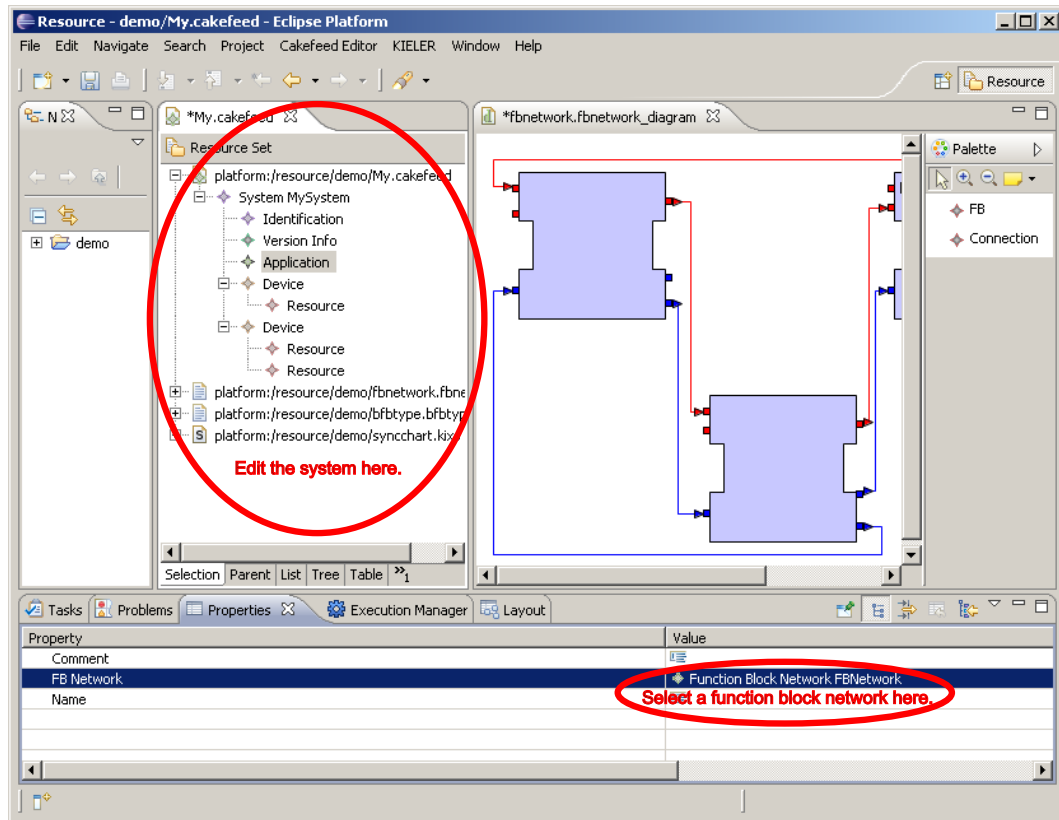


Figure 6.1: The EMF tree editor.

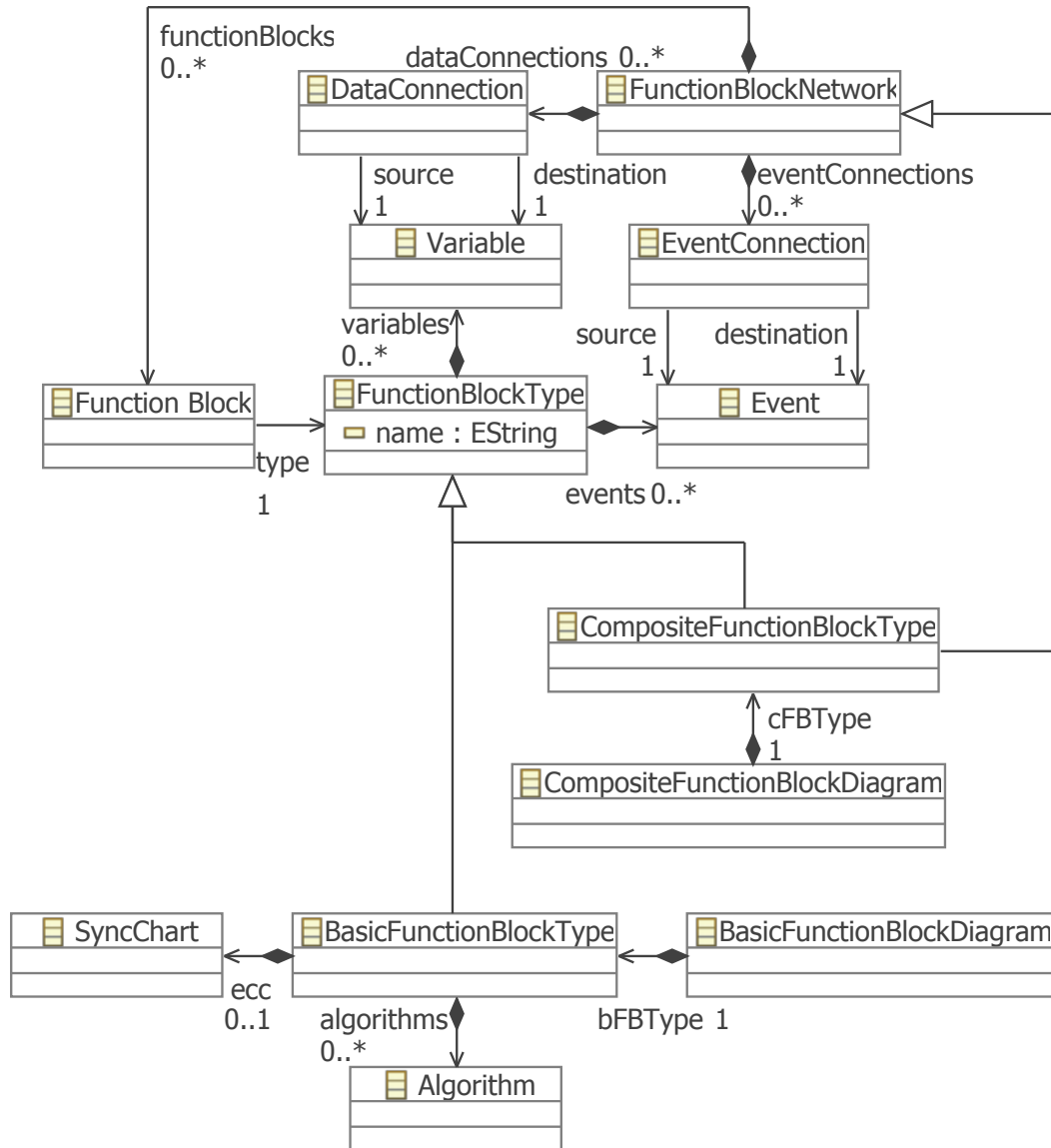


Figure 6.2: The three main classes of the meta-model.

6 Implementation and Results

classes such as a name, events, and variables. In addition, it may contain a SyncChart that serves as its ECC and a range of different algorithms that encompasses *function block diagrams*, *ladder diagrams*, *structured text*, and *source code algorithms*. For the sake of brevity, these have been merged into a single class `Algorithm` in the diagram. The meta-model of SyncCharts is explained in the study thesis [19].

CompositeFunctionBlockTypeDiagram The `CompositeFunctionBlockTypeDiagram` is the container of the `CompositeFunctionBlockType`. Like the `BasicFunctionBlockType`, this class inherits a name attribute as well as input and output events and variables. Also, it inherits the attributes of a function block network that lets it contain other function blocks and connections.

FunctionBlockNetwork The `FunctionBlockNetwork` simply contains other function blocks and the connections between them.

Events and Variables Note that there are several different subclasses of events and variables. These are depicted in Figure 6.3. On the one hand, they are divided into input and output classes. These are used to distinguish inputs and outputs so that it is not possible to connect an input event in the interface of a function block to another input event in the same interface for example. On the other hand, there are classes prefixed with either IF or FB. This is used to distinguish between events and variables that are contained in the interface of a basic or composite function block type and those that are contained in a function block. This is required since sometimes it is necessary to connect an input event in an interface to an input event in a function block, for example. Figure 6.5 summarizes how the different kinds of events and variables may be connected to each other.

Connections To ensure that events and variables are connected correctly, it is also necessary to have a number of different classes for connections. They are depicted in Figure 6.4. For brevity, connections for internal variables have been left out. Every one of them has a source and a destination attribute with different types. There are those connections that may only connect two events or variables in the interface of a function block type, those that may only connect two that are contained in function blocks, and those that may connect interface and function block events or variables. Note that the function blocks language is a dataflow language, which means that connections may only start at either interface input ports or function block output ports and end at either interface output ports or function block input ports. The relations between events, variables and connections are also summarized in Figure 6.5.

6.1.2 The Graphical Definition Models

The graphical definition models of the CAKeFEED editor define the appearance of diagram elements. As this type of model is explained in the study thesis [19], only

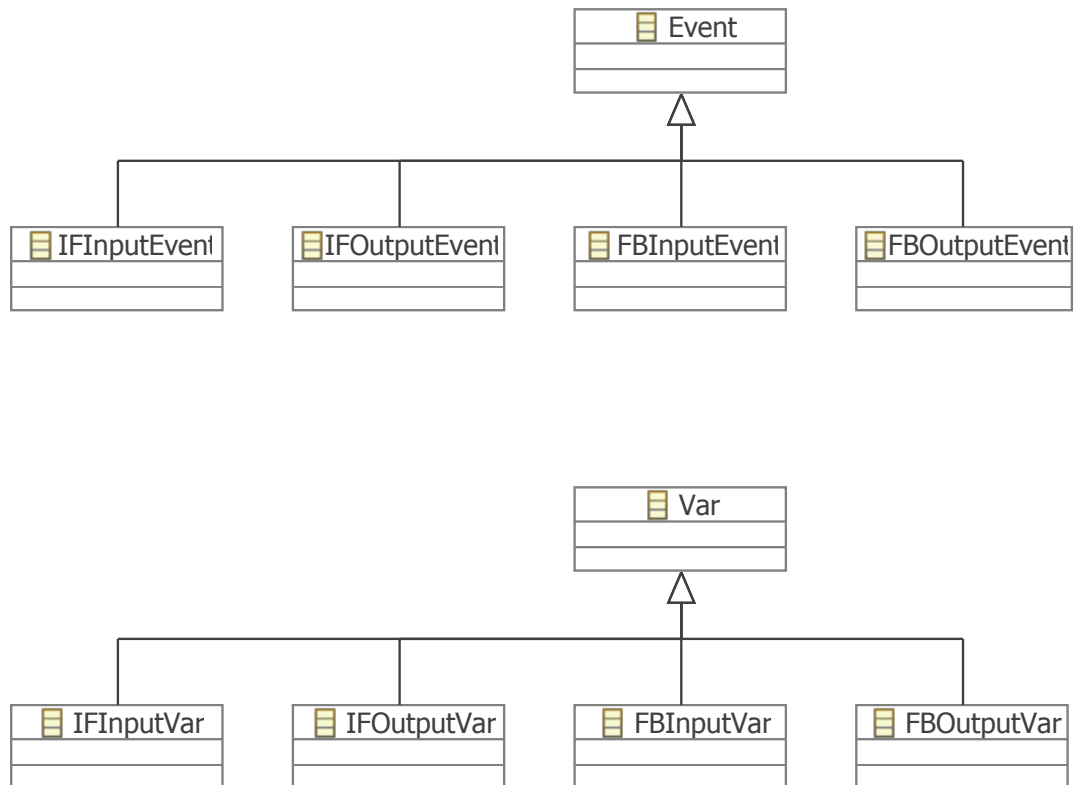


Figure 6.3: The different types of events and variables.

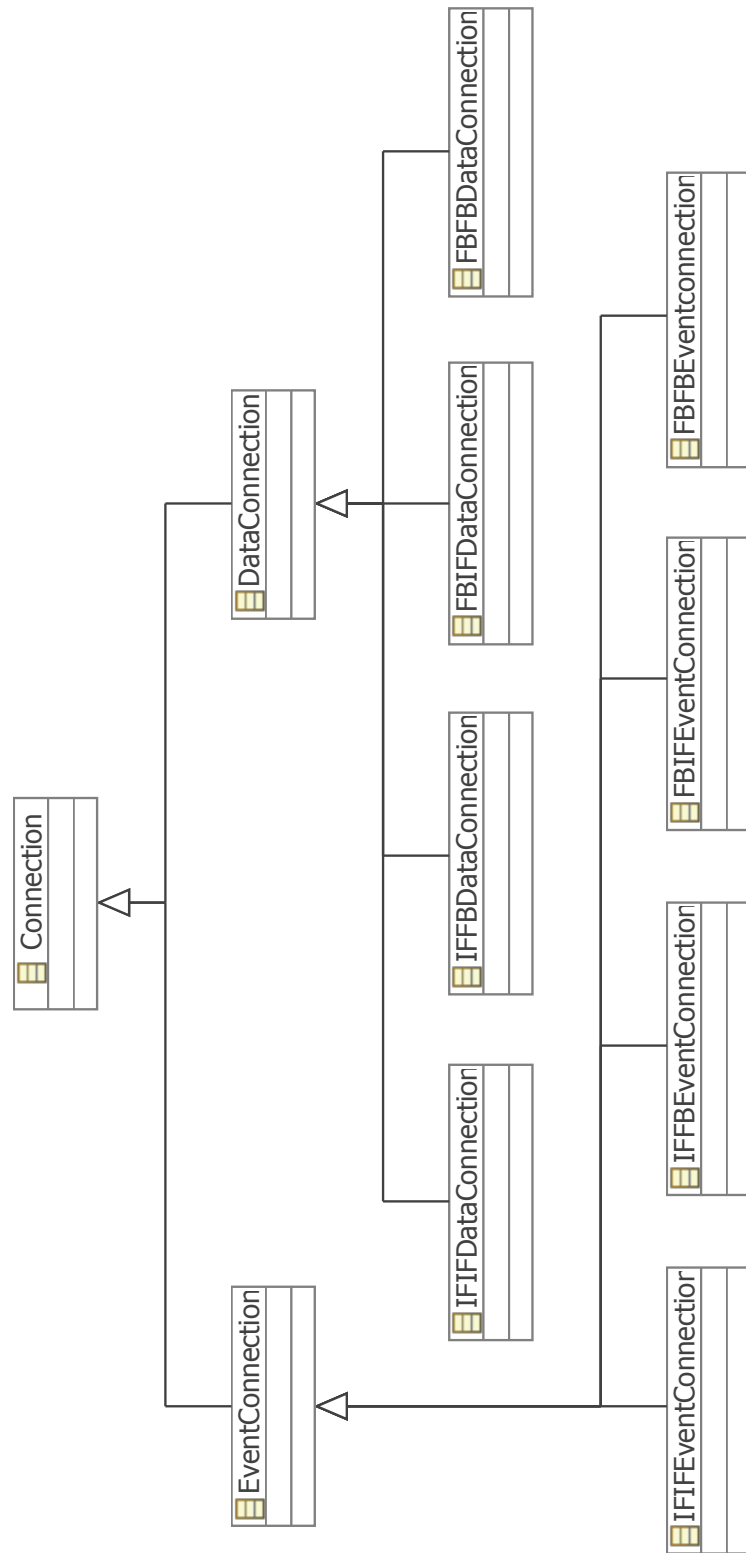


Figure 6.4: The different types of connections.

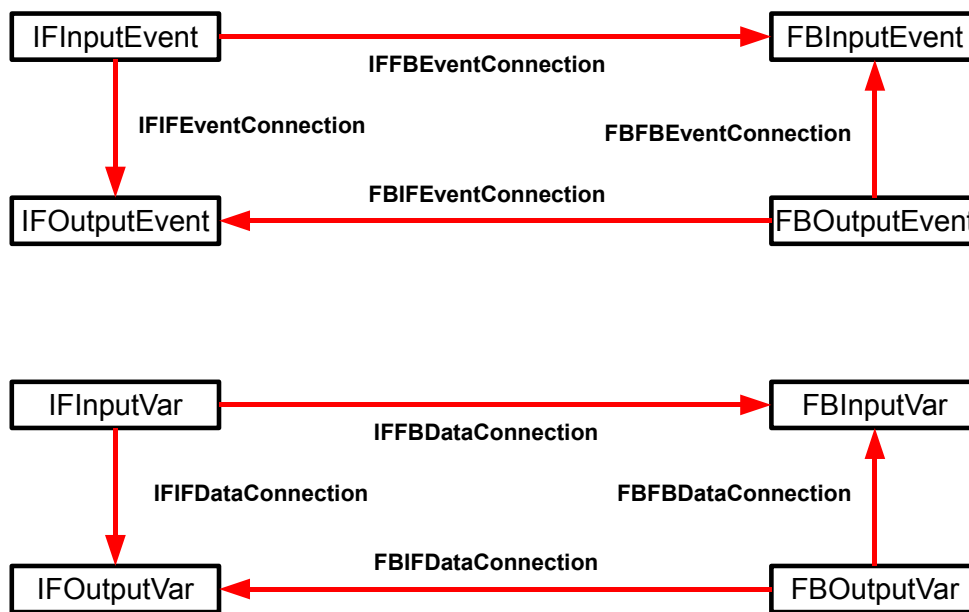


Figure 6.5: How events and variables may be connected.

the notable aspects of the particular models for the CAKeFEED editor are explained here. Important to note is that the figures that represent function blocks are custom figures as well as those that represent ports in interfaces. There has to be a number of different figures for events and variables in interfaces since events and variables feature a short line at the side, which is on the left in the case of inputs and on the right in the case of outputs. Ports in function blocks do not have these lines, thus there is no need for a distinction between inputs and outputs. Take a look at Figure 6.6 for a summary of port appearances.

6.1.3 The Tooling Definition Models

The tooling definition models are straight-forward. They simply contain creation tools for all different elements that need to be created including basic function block types, events, variables, connections, and others.

6.1.4 The Mapping Models

Although the mapping models might seem complicated, their overall structure is rather easy to understand. In the case of basic and composite function block type diagrams the top level elements are mapped to the appropriate types, in the case of function block networks the top level elements are mapped to function blocks. Also, each type or function block contains its corresponding event and variable mappings

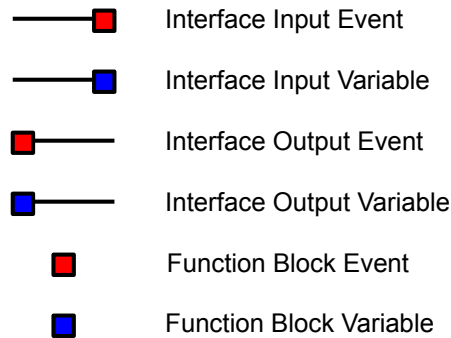


Figure 6.6: How different ports appear in the diagram.

and sometimes additional child mappings for algorithms or other elements. Finally, each model contains appropriate link mappings for the connections.

6.2 Code Modifications

This section discusses the implementation of the type and attribute awareness of function blocks as well as the special appearance of the event and variable ports and how they are laid out.

6.2.1 Type and Attribute Awareness

The desired behavior of a function block figure is to display a number of events and variables that correspond to the ones defined in its function block type. Also, these events and variables have to be able to react to user input like clicking and dragging. This is not possible for simple figures. Instead, there also have to be edit parts that implement this behavior. The easiest way to bring edit parts into the editor is to add event and variable references to the `FunctionBlock` class in the ecore model. This leads to both function blocks and function block types containing events as well as variables although it would be sufficient if only the function block type contained them. The next step is to make the edit part of the function block react to a change of the function block's type by removing old events and variables and adding new ones that correspond to the newly selected type. This is done by overriding the method `handleNotificationEvent` in all function block edit parts with the help of a custom code template.

The template is constructed in such a way that whenever the code for a function block edit part is to be generated, the new definition of the method `handleNotificationEvent` is appended. The workflow of the method is depicted in Figure 6.7. In this method the edit part checks whether the type of the function block has been changed.

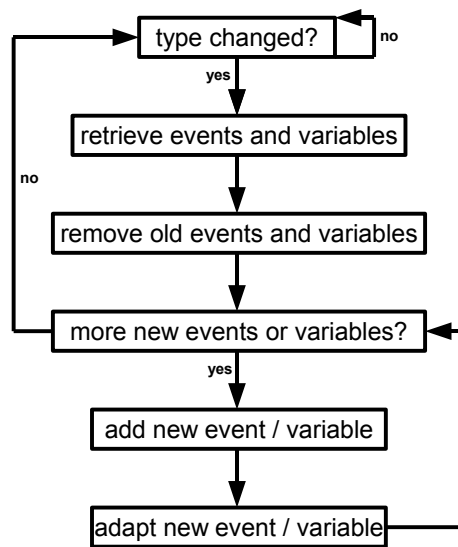


Figure 6.7: The workflow of the method `handleNotificationEvent`.

If that is the case, it retrieves the new events and variables from the newly selected type. Then it removes all events, variables, and connections from the function block. After that, it iterates over all events and variables, puts a command on the command stack for every element to be added, and executes them. Finally the same is done with commands that adapt the attributes of the newly created elements to those of the elements in the type. The commands for removing events and variables, adapting their attributes, and removing connections are subclasses of the class `FBCommand`, which is a subclass of `Command` and features an additional attribute `functionBlock` that denotes the function block to be changed by the command. Figure 6.8 illustrates this.

The Function Block Figure As defined in the standard [2], function blocks have a specific shape that identify them as such. To give the function block figures in the diagram editor the same appearance they are represented by custom figures with special drawing methods that create the required shape. An additional feature that improves the usability of the editor is the fact that the function block figures adjust their appearance based on the number of events and variables: As depicted in Figure 6.9, the figures automatically resize their upper and lower parts depending on the numbers of input and output events and variables.

This is possible by making the figure aware of its corresponding model element's attributes as has been done before in the study thesis [19]. Although the mechanism has been improved since then, the basic approach is the same: The figure is registered as a listener to the model element, which makes the model element call the

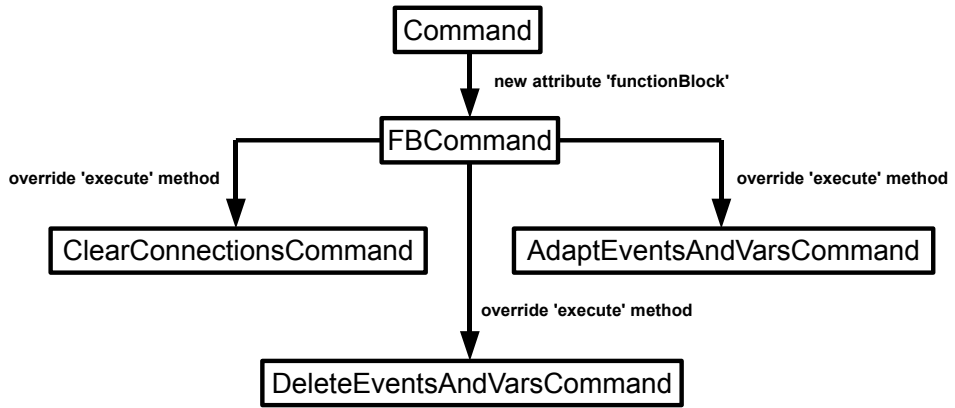


Figure 6.8: The hierarchy of the command classes.

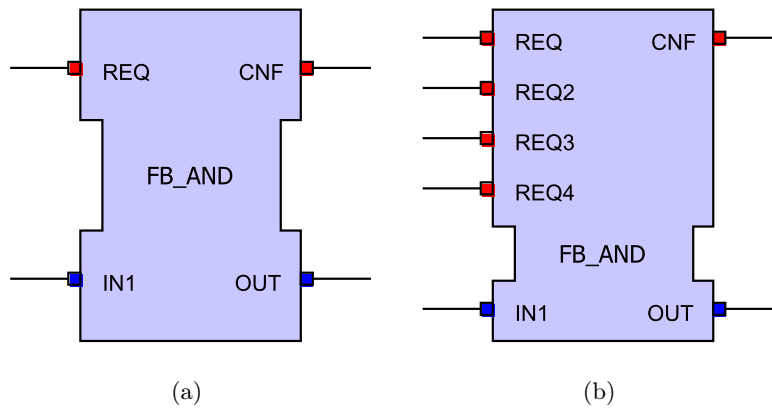


Figure 6.9: The relative sizes of the upper and lower parts change depending on the numbers of events and variables.

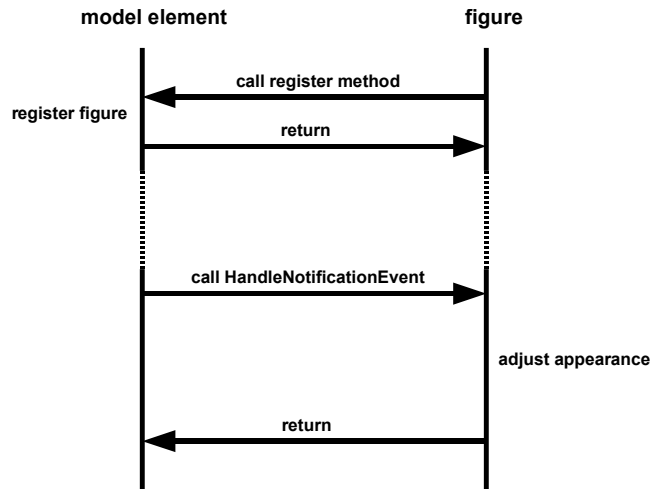


Figure 6.10: How edit parts react to changes in model elements.

`handleNotification` method of the edit part. In this method, the edit part reacts to the changes. Figure 6.10 illustrates this.

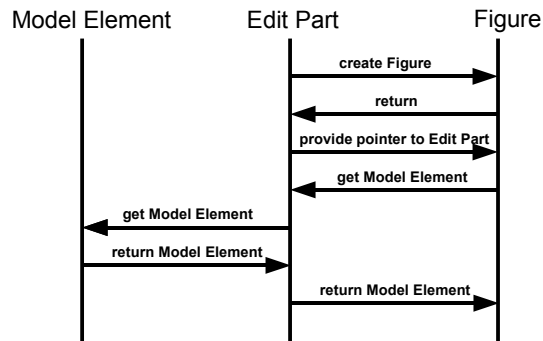
The difference between the old and the new approach is rather subtle: Whereas in the study thesis [19] an attribute aware figure had a reference to its corresponding edit part to get a link to its model element, it now only has a reference to its model element that is set by the edit part itself. Figure 6.11 illustrates this.

By registering the figure as a listener to the model element, its `handleNotificationEvent` method is called every time a change is made to the model element. Thus, the figure can recalculate the new number of events and variables and adapt its template points. The drawing methods have been changed so that they draw an outlined polygon along those template points. The whole process is depicted in Figure 6.12.

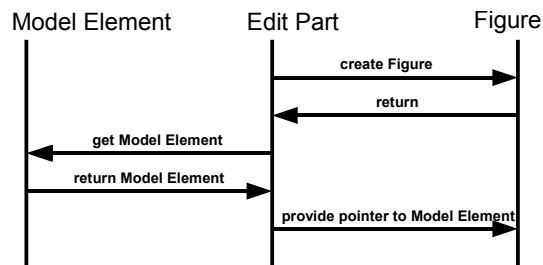
The Port Figures Like the function block figures, the figures of events and variables also have custom drawing methods. Depending on whether they are input or output events/variables they draw themselves pointing in different directions. This is shown in Figure 6.13.

Every port has a boolean attribute `reverse` that indicates whether the line is to be drawn to the left or to the right of the port's square. In addition, there are methods that compute the location of the square and the start point as well as the end point of the line depending on the `reverse` attribute. The figures of interface input and output events and variables are derived from the super class `PortFigure` and have a constructor that sets their `reverse` attribute and their fill color. Figure 6.14 summarizes this.

6 Implementation and Results



(a) The figure needs a pointer to its edit part.



(b) No additional pointer is needed.

Figure 6.11: The old and the new attribute awareness.

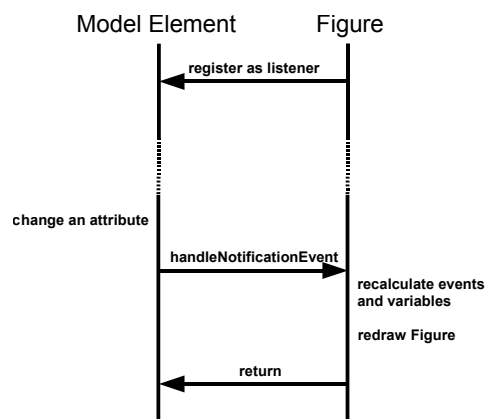


Figure 6.12: What happens when a change occurs.

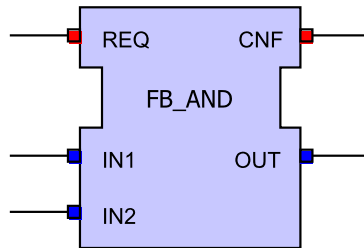


Figure 6.13: Input ports point to the left while output ports point to the right.

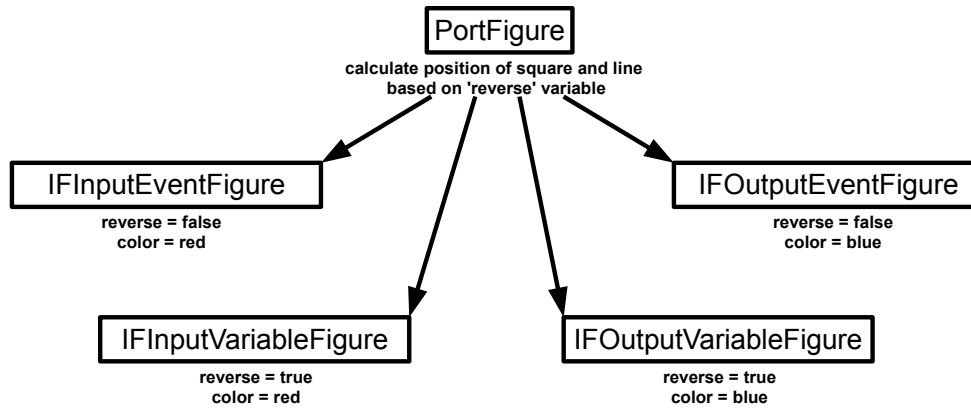


Figure 6.14: The different port figures.

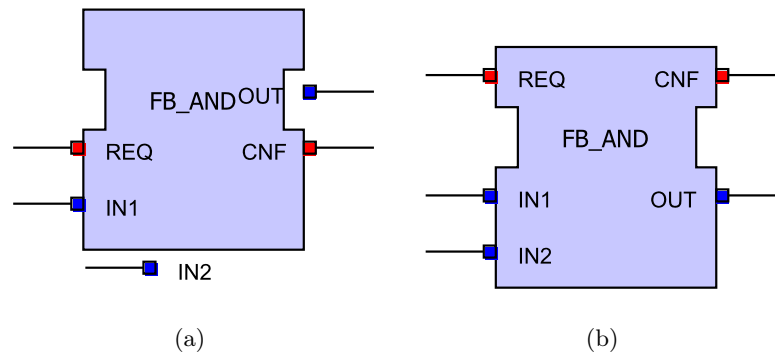
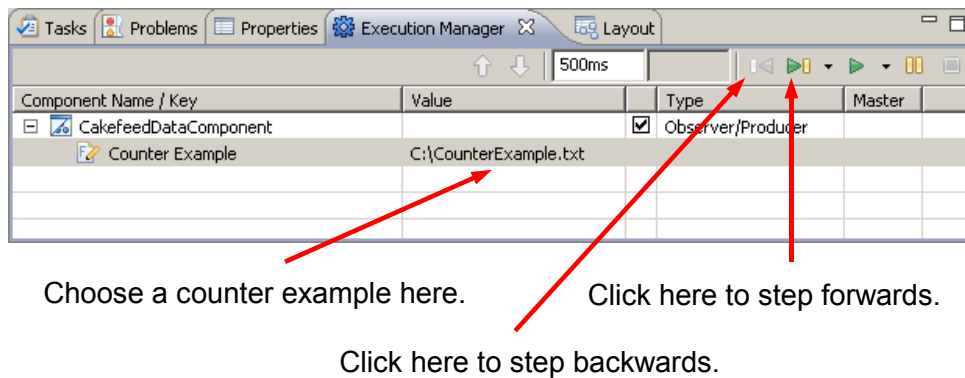


Figure 6.15: The default `BorderItemLocator` and the one employed by the function blocks editor.

6.2.2 Port Layout

In the IEC 61499 function blocks standard, it is defined that all ports have to be in a certain location: Input events are supposed to be located on the upper left side, while output events are supposed to be located on the upper right, input variables on the lower left, and output variables on the lower right side. To adhere to these restrictions, locators have been employed in the CAKeFEED editor.

Use of Locators Locators determine the relative position of side affixed children compared to their parent figure. The `BorderItemLocator`, which is used as default for side affixed children, returns as possible positions those on the border of the parent figure. In the function blocks editor, the possible positions are restricted as the `CakefeedBorderItemlocator` limits the position of input events and variables to the left border and output events and variables to the right border of the function block. In addition, the possible positions of events are restricted to the upper part and the ones of variables to the lower part of the function block. To realize this, the `CakefeedBorderItemLocator` contains a reference to the function block figure it is assigned to. Thus, it can retrieve the bounds of that figure's upper and lower part. In addition, it has two boolean attributes `input` and `ctrl` that define whether it is supposed to restrict the location of inputs or outputs and events or variables, respectively. When asked to return a valid location, the locator first lets its super class calculate all locations along the border of the function block. Then it restricts them based on the upper and lower regions of the function block figure and the `input` and `ctrl` attributes. Figure 6.15 compares the default `BorderItemLocator` and the `CakefeedBorderItemlocator`.

Figure 6.16: The interface provided by the *KIEM*.

6.3 Simulation

The simulation of counter examples has been implemented with the help of the KIEM [17]. It provides a user interface that consists of a property field used to select a file that contains a counter example, a **step forward** button, and a **step backward** button as depicted in Figure 6.16.

6.3.1 The Data Component

The data component simulates the counter example. Figure 6.17 illustrates its behavior. When the step forward button is pressed for the first time, the `initialize` method is called that parses the file containing the counter example and produces a *W3C document object*. In any following activation of the step forward or step backward button the number of the current tick is updated and the events and ECC transitions that have to be highlighted are extracted from the W3C document object. Finally, these elements are highlighted by drawing them in a light green color. Note that whether an event or a transition has to be highlighted is determined solely by the name of the event or the source and target states, as this is the only information available in a counter example. This means that the simulation only works correctly when all states in an ECC and all function blocks have a unique name. Event names only need to be unique within their own function block. The Listings 6.1 and 6.2 show a simple counter example and Figures 6.18 and 6.19 depict the corresponding function block network and the ECC of function block A.

As can be seen, the function block network represents a ring of three interconnected function blocks. The counter example states that in every tick the output event of one of the function blocks is active along with the input event of the following function block, which means that the connection that connects both events has to be

6 Implementation and Results

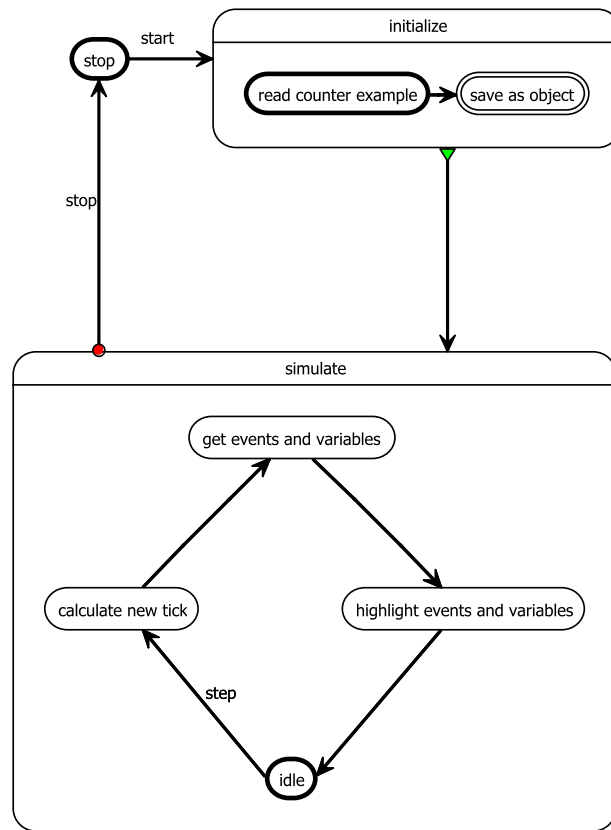


Figure 6.17: The behavior of the data component.

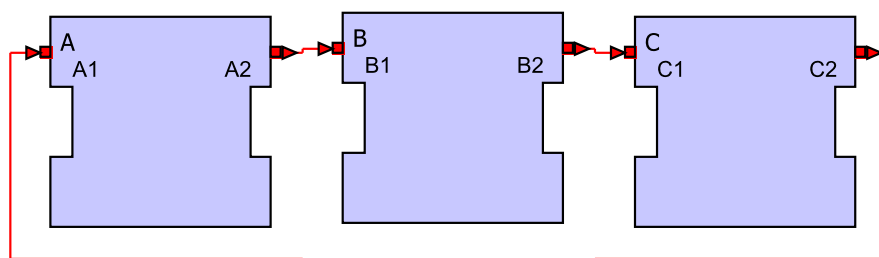


Figure 6.18: The corresponding function block network.

Listing 6.1: A simple counter example.

```

1 <?xml version="1.0" encoding="UTF-8"?>
  <FB Type="input\rmc\example\example" Name="input\rmc\example\example">
    <Tick Id="1">
      <Component Name="A">
        <InputSignals>
6         </InputSignals>
        <OutputSignals>
        </OutputSignals>
        <SourceState>S0</SourceState>
11        <DestinationState>S1</DestinationState>
      </Component>
      <Component Name="B">
        <InputSignals>
16        </InputSignals>
        <OutputSignals>
        </OutputSignals>
      </Component>
      <Component Name="C">
        <InputSignals>
21        </InputSignals>
        <OutputSignals>
        </OutputSignals>
      </Component>
    </Tick>
26 </Tick>
    <Tick Id="2">
      <Component Name="A">
        <InputSignals>
31        </InputSignals>
        <OutputSignals>
        <Signal>A2</Signal>
        </OutputSignals>
        <SourceState>S1</SourceState>
36        <DestinationState>S2</DestinationState>
      </Component>
      <Component Name="B">
        <InputSignals>
        <Signal>B1</Signal>
        </InputSignals>
41        <OutputSignals>
        </OutputSignals>
      </Component>
      <Component Name="C">
        <InputSignals>
46        </InputSignals>
        <OutputSignals>
        </OutputSignals>
      </Component>
    </Tick>
  </FB>

```

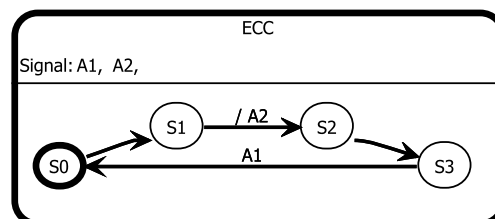


Figure 6.19: The ECC of function block A.

Listing 6.2: A simple counter example (continued).

```

5  <Tick Id="3">
    <Component Name="A">
      <InputSignals>
      </InputSignals>
      <OutputSignals>
      </OutputSignals>
      <SourceState>S2</SourceState>
      <DestinationState>S3</DestinationState>
    </Component>
10 <Component Name="B">
    <InputSignals>
    </InputSignals>
    <OutputSignals>
      <Signal>B2</Signal>
    </OutputSignals>
15 </Component>
    <Component Name="C">
      <InputSignals>
      <Signal>C1</Signal>
      </InputSignals>
      <OutputSignals>
      </OutputSignals>
20 </Component>
    </Component>
  </Tick>
25 <Tick Id="4">
    <Component Name="A">
      <InputSignals>
      <Signal>A1</Signal>
      </InputSignals>
      <OutputSignals>
      </OutputSignals>
30 <SourceState>S3</SourceState>
      <DestinationState>S0</DestinationState>
    </Component>
35 <Component Name="B">
    <InputSignals>
    </InputSignals>
    <OutputSignals>
    </OutputSignals>
40 </Component>
    <Component Name="C">
      <InputSignals>
      </InputSignals>
      <OutputSignals>
      <Signal>C2</Signal>
      </OutputSignals>
45 </Component>
  </Tick>
50 </FB>

```

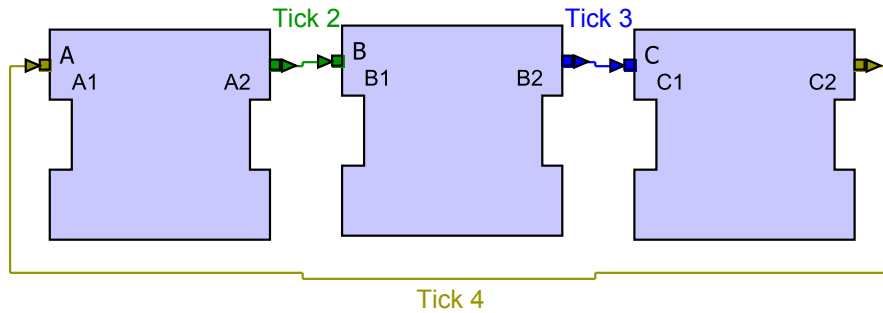
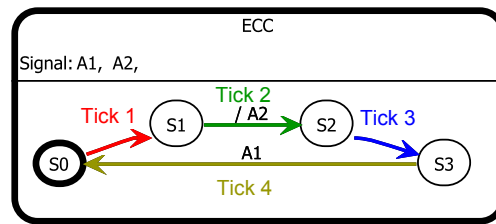


Figure 6.20: Simulation of the function block network.

Figure 6.21: Simulation of the *ECC*.

highlighted. Also, in every tick, the source and destination states of function block A change. This indicates that certain transitions in the ECC of function block A are taken and have to be highlighted. Figure 6.20 shows the simulation of the function block network and Figure 6.21 that of the ECC.

6.4 Import and Export

The import and export of function blocks is realized with the help of two *handlers*, the *import handler* and the *export handler*. These implement `execute` methods that are invoked when the appropriate entry in the context menu is selected. However, the work on this feature is not finished at this moment.

6.4.1 The Import Handler

The supposed workflow of the import handler is the following:

1. Open file wizard
2. Read file
3. Transform model
4. Store model in new file

6 Implementation and Results

5. Open file in new editor

First, it opens a dialog that lets the user select a file that contains a function block stored in the XML format defined in the standard [3]. Then a new resource is created for a new function block. After that, an Xtend transformation is employed to translate the function block that is defined in the file selected by the user into a function block that conforms to the XML format used by the editor. The transformed function block is then stored in the newly created resource, which is opened in a new editor window for the user to edit.

6.4.2 The Export Handler

The planned behavior of the export handler is as follows:

1. Get currently edited model
2. Transform model
3. Store model in new file

First, it extracts the currently edited element from the opened diagram. Then it chooses the appropriate transformation based on the class of the element. After that, the chosen transformation is executed with the element as argument. Then, a new resource is created that can contain the resulting element in the IEC 61499 function blocks format and the transformed element is stored in it.

6.4.3 The Transformations

Since the hybrid meta-model has been derived from the meta-model that results from the XSD, the transformations needed to translate IEC 61499 function blocks and CAKeFEED function blocks into one another are quite simple. The four tasks these transformations have to fulfill are to eliminate and insert intermediate containers, to transform object references into strings, and to transform strings into object references. The first three tasks are trivial, since intermediate containers can easily be omitted or inserted by changing some pointers and object references can easily be replaced by the names of their objects. However, to transform strings into object references, it is necessary to search the workspace for files that contain objects with the name defined in the string. This is not possible with Xtend alone. Fortunately, it is possible to write a *Java extension* in Xtend, which can execute a method written in *Java*. At this moment, the transformations are not complete, since the Java extensions still have to be written.

6.5 Evaluation

The approach of using the hybrid meta-model presented in this thesis has several advantages. On the one hand, its similarity to the original meta-model derived from

the XSD results in a separation of concerns as basic function block types, composite function block types, and function block networks are all created and edited in different windows, and also in a better clarity because there is only one level of hierarchy in every diagram. It also makes the development of transformations for import and export a lot easier.

The use of Eclipse and the GMF together with the EMF and the GEF is advantageous because many useful features are automatically provided, which would otherwise have to be implemented manually. These features include drag-and-drop editing, outlines, overviews, menu and tool bars, the properties view, and many others. In addition, there are many extension points, which provide a comfortable means to extend existing features or add new ones. A disadvantage is the XML format used by GMF editors. Although it is already very flexible, the need for a `#` between resource and fragment makes the approach of the XML-conformant meta-model infeasible.

The KIELER project provides excellent support for the CAKeFEED editor since the KIEM alleviates the implementation of counter example simulation and the automatic layout saves users of the editor a lot of work that would otherwise be spent on tweaking the layout of function block diagrams manually.

6 *Implementation and Results*

7 Conclusion

This chapter concludes the thesis with a summary in Section 7.1 and an outlook on possible future work in Section 7.2.

7.1 Summary

In this thesis, a new GMF diagram editor for the function blocks language has been presented that conforms to the IEC 61499 function blocks standard. Several different approaches to the creation process have been discussed, such as the XML-conformant meta-model, the custom meta-model, and the hybrid meta-model along with their advantages and disadvantages. One of these approaches has been employed to implement the editor and has been explained in detail. While the chosen meta-model was the most advantageous for this project, the other ones have other interesting qualities and it may prove worthwhile to employ them, too. The KIELER project has been used to show how graphical editors created with the GMF can benefit from the features of KIELER. It was shown how the KIEM can be used to implement a simulator in a GMF editor and how attribute aware figures can easily be implemented with the classes provided by the KIELER. Also, the versatility of the GMF was utilized to make function blocks react to user inputs by adapting their events and variables to the chosen type. In addition, the layout of events and variables was realized by employing locators.

7.2 Future Work

The CAKeFEED function blocks editor provides a comfortable environment for the creation of function blocks. However, some of the features have potential to be extended in the future. The import and export mechanism could be completed and the simulation feature could be extended by a simulation mode where the user specifies the active inputs. Also, it might be worth a try to employ the custom meta-model instead of the hybrid meta-model so that the features of the KIELER can be used more effectively. In the course of that approach it would also be an interesting endeavor to develop a generic library mechanism for GMF editors and to explore the use of multiple hierarchy levels.

7 Conclusion

8 Bibliography

- [1] XML Path Language (XPath) Version 1.0, 1999. <http://www.w3.org/TR/xpath/>.
- [2] IEC 61499-1: Function blocks - part 1 architecture, 2005. http://webstore.iec.ch/preview/info_iec61499-1{ed1.0}en.pdf.
- [3] IEC 61499-2: Function blocks - part 2 software tool requirements, 2005. http://webstore.iec.ch/preview/info_iec61499-2{ed1.0}en.pdf.
- [4] Meta object facility (mof) 2.0 query/view/transformation specification. Technical report, OMG Object Management Group, 2007. <http://www.omg.org/mof/>.
- [5] C. André. Semantics of SyncCharts. Technical Report ISRN I3S/RR-2003-24-FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
- [6] Ö. Bayramoglu. The KIELER textual editing framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/oba-dt.pdf>.
- [7] N. Beckel. View Management for Visual Modeling. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Oct. 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nbe-dt.pdf>.
- [8] W. Bolton. *Programmable Logic Controllers*. Newnes, 2009.
- [9] E. Börger and R. Stark. *Abstract State Machines*. Springer, 2003.
- [10] J. Chouinard and R. W. Brennan. Software for next generation automation and control. In *Industrial Informatics, 2006 IEEE International Conference on*, pages 886–891, 2006.
- [11] S. Efttinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006.
- [12] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. In *Proceedings of the 15th International Monterey Workshop on Foundations of Computer Software, Future Trends and Techniques for Development (2008)*, LNCS (to appear), Budapest, 2010. Also available as Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

- [13] P. Z. S. Gareth D. Shaw, Dr. Partha S. Roop, editor. *A hierarchical and concurrent approach for IEC 61499 function blocks*, IEEE International Conference on Emerging Technologies and Factory Automation, 2009.
- [14] C. F. Goldfarb and P. Prescod. *XML Handbook, Third Edition*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.
- [15] K.-H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer, 2001.
- [16] M. Matzen. Structure-Based Editing in Eclipse. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Mar. 2010. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mim-dt.pdf>.
- [17] C. Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf>.
- [18] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.
- [19] M. Schmeling. An Eclipse-Editor for Safe State Machines. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/schm-st.pdf>.
- [20] M. Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Mar. 2009. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf>.
- [21] M. Spönemann, H. Fuhrmann, R. von Hanxleden, and P. Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010.
- [22] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.
- [23] K. Thramboulidis. *Advances in Computer, Information, and Systems Sciences, and Engineering*. Springer, 2006.
- [24] V. Vyatkin. *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*. ISA, 2007.