Masterarbeit

Design and implementation of a domain specific architecture for programmable logic controllers

Andreas Haselsberger, Bakk.techn.

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß

Begutachter und Betreuer: Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, im Jänner 2010

Abstract

The gap between software development on computers and programmable logic controllers (PLC) is wide. PLC-programming is a growing discipline, done by technicians whereas software development is more an academic discipline. There are many projects that try to improve the quality and reuse of PLC software and make it more systematic. Some of these projects for example apply existing software development paradigms like object oriented software development, with little outcome however.
An important attempt to reach systematic PLC code design was a standardised architecture and programming scheme (DIN/EN 61131) which is unfortunately not explicit.
Another influential issue is the development speed (time to market) and flexibility of PLC compositions which is moderate.
Software product lines (SPL) are state of the art approach in systematic software reuse. The three main processes are *domain engineering* to extract reusable parts, *application engineering* for creating custom applications and the *management* to deal with economic aspects.
The goal of this master thesis is to create a domain specific architecture and an implementation to support systematic software reuse for a PLC controlled inventory system model. A synergy between the software development paradigms SPL and PLC programming is analysed, developed and implemented. A suitable implementation of an SPL for the logistics domain is selected and a tool evaluation is done. An architecture model with abstract logistics objects and a corresponding platform model (PLC) for the implementation is developed. The transformation between the models is done with generators which use function-blocks as the basic concept. These generators extract the information of the modeled logistics system to create the target sources: function-blocks, functions, data blocks and organisational blocks. The implemented process includes the graphical system architecture modeling which can be transformed to the full and complete PLC source code together with consistent documentation and other helpful files (i.e. a pin binding diagram, recommended tests or a sample hardware configuration).
Finally a technical- and business evaluation is done to compare this work with existing methods and the literature.

Kurzfassung

In der Praxis besteht ein wesentlicher Unterschied zwischen der Software Entwicklung für herkömmliche Rechner und der für speicherprogrammierbare Steuerungen (SPS). Während die SPS Programmierung meist von Technikern durchgeführt wird, gibt es die Software Entwicklung als akademische Disziplin, dies zeigt bereits die Unterschiede dieser beiden Techniken. Viele Projekte versuchen die Entwurfsmethodik von SPS-Programmen zu verbessern und zu systematisieren. Es gibt Ansätze, bereits existierende Methoden, wie beispielsweise objektorientiertes Programmieren, auf die SPS-Programm-Entwicklung abzubilden, deren Erfolg aber nur mäßig ist.
Ein wichtiger Schritt in Richtung systematischer Programmierung war die Einführung einer Norm (DIN / EN 61131), die die Programmierung von SPS Systemen vereinheitlichen sollte, was aufgrund vieler Ungereihmtheiten nur dürftig gelang.
Ein wesentliches Maß der derzeitigen Marktlage ist die Geschwindigkeit mit der neue Produkte auf den Markt gebracht werden (time to market) und die Flexibilität von SPS-Systemen, die nicht ausreichend effizient ist.
Software Produkt Linien (SPL) sind derzeit *state of the art* in Sachen effizienter Wiederverwendung von Code und damit auch effizienter Produktion. Die drei wesentlichen Teilprozesse sind *Domain-Engineering*, um wiederverwendbare Blöcke zu identifizieren und zu erstellen, *Application-Engineering*, um Anwendungen aus der Produkt Linie abzuleiten und das *Management*, das durchgehend involviert ist und firmenpolitische sowie finanzielle Aspekte berücksichtigt.
Das Ziel dieser Arbeit ist der Entwurf und die Entwicklung einer domänenspezifischen Architektur für ein SPS-System, die ein hohes Maß an Wiederverwendung und Entwicklungsbeschleunigung bietet. Dabei handelt es sich um Labormodell eines Logistiksystems. Es wird versucht, eine Synergie zwischen der Entwicklung von Software Produkt Linien und der SPS Programmierung zu finden und zu realisieren. Dazu wird eine entsprechende Implementation einer SPL ausgewählt und eine Tool-Evaluierung für diese Logistik Anwendung durchgeführt. Ein Architektur-Modell mit abstrakten Objekten der Logistik und ein zugehöriges Plattform Modell (PLC) für diese Implementierung wird entwickelt. Die Transformation zwischen den Objekten wird mit Generatoren realisiert, welche Funktionsblöcke als Grundbausteine nutzen. Generatoren extrahieren Information aus dem modelierten System, um den fertigen und kompletten Quellcode zu generieren (Funktionsbausteine, Funktionen, Daten- und Organisationsbausteine). Der entwickelte Prozess beinhaltet eine graphische Modellierung des Systems, das ein fertiges PLC Programm mit einer persistenten Dokumentation und weitere hilfreiche Dokumente (z.B. Verdrahtungsplan, Tests oder eine mögliche Hardware-Konfiguration) erstellt.
Abschließend wird eine Evaluierung durchgeführt, die technische als auch wirtschaftliche Aspekte beinhaltet um diese Arbeit mit anderen Ansätzen vergleichen zu können.

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....	....
date	(signature)

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
AWL	Anweisungsliste
BsC	Bachelor of Science
CAFE	Concepts to Application in System-Family Engineering
CAD	Computer Aided Design
COM	Component Object Model
CSS	Cascading Style Sheet
DDD	Detailed Design Document
DIN	Deutsches Institut für Normung
DOME	Domain Modeling Environment
DOPLER	Decision-Oriented Product Line Engineering for effective Reuse
DSL	Domain Specific Languages
DSM	Domain Specific Modeling
ECU	Engine Control Unit
EN	Europäischen Normen
ESAPS	Engineering Software Architectures, Processes, and Platforms for System Families
FBD	Function Block Diagram
FODA	Feature-Oriented Domain Analysis
FUP	Funktionsplan
GEMS	Generic Eclipse Modeling System
GME	Generic Modeling Environment
GMF	Graphical Modeling Framework

GMT	Generative Modeling Technologies
GOPPRR	Graph Object Property Port Role Relationship
GP	Generative Programming
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IL	Instruction List
ISIS	Institute for Software integrated Systems
ITEA	Information Technology of European Advancement
LAN	Local Area Network
LD	Ladder Diagram
LGPL	Lesser Gnu Public License
MEDEIA	Model-Driven Embedded Systems Design Environment for the Industrial Automation Sector
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MERL	MetaEdit Reporting Language
oAW	openArchitectureWare
OCL	Object Constraint Language
OLE	Object Linking and Embedding
OMG	Object Management Group
OPC	OLE for Process Control
PIG	PLC Implementation Generator
PL	Product Line
PLC	Programmable Logic Controller
POU	Program organistation Unit
RAM	Random Access Memory
RFID	Radio Frequency Identification

ROM	Read Only Memory
SCL	Structured Control Language
SEI	Software Engineering Institute
SFC	Sequential Function Chart
SIPN	Signal Interpreted Petri Nets
SOAP	Simple Object Access Protocol
SPL	Software Product Line
SPLC	Software Product Line conference
SPLE	Software Product Line engineering
SQL	Structured Query Language
ST	Structured Text
UML	Unified Modeling Language
WSDL	Web Service Description Language
XML	Extensible Markup Language

Credits

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.
I would like to thank my supervisor Christian Kreiner for his encouragement and many research discussions. Further, I would like to thank all involved staff of the institute, especially the IT-administrators who provided a great workplace and Michael Thonhauser. Finally would like to say thanks to my family and Verena. They kept me grounded at all times.

January 2010 Andreas Haselsberger

Chapter 1

Introduction

This thesis describes a novel approach of a domain specific architecture for programmable logic controllers (PLC). The current chapter gives a short motivation, introduces the goal and the structure of this work.

1.1 Motivation and goal

The basic idea of this thesis arose from a project which was done at the Institute of Technical Informatics at Graz University of Technology. The project dealt with the automation of a conveying system in combination with a high bay racking and other hardware for a logistics system. The development of an application for this domain resulted in an inflexible and fixed software, although modern techniques (function-blocks) for programming programmable logic controllers were used.
Such applications should be adaptable and flexible. Each customer has a special structure or limitations in space, so every application is different although they are similar. Perhaps the high bay racking is smaller, loading is done in another way, or additional hardware like RFID readers are used. These different requirements should not lead to a complete redesign and a new implementation of the software. Novel functionality for the system should be covered too, to include it in future applications.

So the specification for a flexible domain specific architecture arose. Defined domain and business specific changes in the system should be covered. This modifications should be realised very simply without a redesign just with strategic and systematic reuse of code.

Generally the software development for computers can not be compared to the development of applications for programmable logic controllers. Software development is an academic discipline which has been explored for more than 60 years whereas PLC programming is done by technician. The programming of PLC has not changed significantly over the years, but many paradigms for conventional software development have come into existence. Currently, researches try to apply known software development methods like object oriented approaches to programmable logic controllers with little outcome. The reason for missing methods is the history of PLC. Many vendors came into existence in the PLC market and all of them had different approaches of programming and different architectures which made it hard to develop uniform paradigms. The hardware and software of todays programmable logic controllers is also not fully compatible. PLC are state of the art for real time control systems so standards were created. The two important standards are IEC 61131 and the newer IEC 61499.

Software product lines are currently the state of the art in strategic reuse of software. Figure 1.1 shows a short history of reuse in software development.

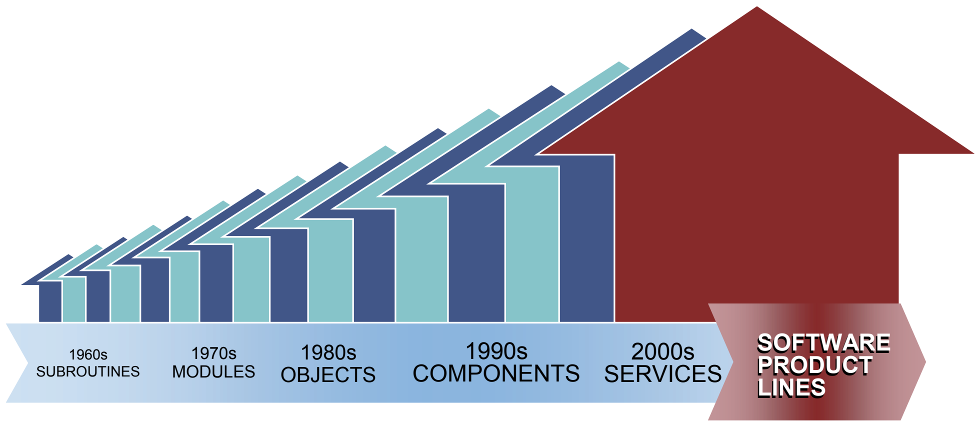


Figure 1.1: Reuse History [Nor07]

In the 1960s, reuse started with subroutines, followed by modules in 1970s and objects in 1980s. In 1995, components appeared followed by services in 2000. Today, software product lines are seen as the most effective way to implement strategic reuse. The key idea of product lines is old and based on Henry Ford's mass customisation to provide an effective way for cheap, individual cars. Today many different approaches exist to implement a software product line.

The main goal of this thesis is the design and implementation of a domain specific architecture for programmable logic controllers. This architecture should provide a simple design and implementation of applications for a logistics system: a straight and flexible architecture from design to implementation and finally deployment to the controller. The design of the structure should be done in a graphical way without textual programming by reusing existing objects. A common protocol between the individual parts should be worked out too.

1.2 Structure of the thesis

Chapter 2 covers the related work of this thesis, starting with Section 2.1 which deals with software product lines. After a short introduction, the history, the terminology and the development process is covered. The three main processes are discussed in detail. After this, the variability which is a major part of the product line is examined. At the end of Section 2.1, current projects and approaches to implement a software product line are shown. Approaches like domain specific languages, generative programming and model driven engineering are discussed.

Section 2.2 discusses the topic programmable logic controllers. Starting with a historical review and the definition the platform and the workflow of a PLC are illustrated. Afterwards, the standards IEC 61131 and IEC 61499 are discussed as well as the harmonisation between them. Finally, new approaches to programming PLC are shown, which are adopted from conventional software development: for example, applying the object oriented paradigm to a programmable logic controller.

Chapter 3 discusses the design and implementation. After defining the requirements, both architectures are shown. The SPL architecture and the PLC architecture. Section 3.4 gives an idea how the generators produce the code and after illustrating the user interface, an evaluation is done.

General trends and ideas for future work are given in chapter 4.

Chapter 5 includes concluding remarks.

The appendix A contains a short evaluation of the used software called MetaEdit+® as well as an installation guide of the implemented software packages. Additionally, a list of criteria for choosing a software product line tool is shown. An example workflow of the implementation and several screenshots are also shown in appendix A.

Chapter 2

Related work

This chapter covers the research which was done during the master thesis. It is divided into two main Sections. Section 2.1 discusses software product lines, Section 2.2 deals with programmable logic controllers.

2.1 Software product line engineering

This Section will deal with Software Product Line Engineering (SPLE). A short historical review and the motivation to use this paradigm is given. Afterwards the terminology and the development process are discussed. This is split into three sub tasks called domain engineering, application engineering and management. Tools to handle SPLE are mentioned, as well as current and honoured projects. The final parts of this Section show current approaches and corresponding tools of software product lines.

2.1.1 Introduction to SPLE

Henry Ford developed the production line in the automotive Section, which enabled the production of goods for the mass market. This production was much cheaper than creating individual products, so the procedure for producing goods changed significantly. An important aspect of the product line is the diversification, because with the original product line, the production of different products was limited.
At the beginning this was not a big problem but not everybody wants the same kind of a product. The industry was confronted with the demand for individualised products. This is called mass customisation [PBvdL05; Sch07].

The definition of mass customisation is the large scale production of goods which are tailored to individual needs [Dav87]. This was the beginning of platforms, because individual products are too expensive and so the profit is too small. Common platforms increase profit because they allow the reuse of technology. The use of this methodology in the software industry is called *software product line engineering* (SPLE).

Software product line engineering is a paradigm to develop software applications (software - intensive systems and software products) using platforms and mass customization [PBvdL05].

Some of the key ideas of SPLE are 25-35 years old. Dijkstra discussed the idea of SPLE in the late 1960s. Parnas and others resolved the idea in the mid 1970s and Jim Neighbors invented domain analysis in the early 1980s. Between 1980 and 1990 a systematisation of SPLE took place and the first applications appeared. The first *Software product line conference* was in the year 2000 [Wei05].

In the United States, the Software Engineering Institute (SEI) leads in dealing with SPL [Ins08].
In Europe there were some projects to improve the overall system knowledge of SPLE. They were organised by the Information Technology for European Advancement (ITEA). ITEA was an industry driven strategic research and development program. The projects were *Engineering Software Architectures, Processes, and Platforms for System Families* (ESAPS 1999-2001), *Concepts to Application in System-Family Engineering* (CAFÉ 2001-2003) and FAMILIES (2003-2005) [vdLBK+04; vdL02a]. ITEA2 is the follow up project to ITEA [fEA08].

2.1.2 Motivation for SPLE

There are several key motivations additional to mass customisation for using software product line engineering [PBvdL05]. The most important are discussed in the following description.

Reduce development costs: An essential reason to apply a new engineering practice is the economical justification which means cost reduction. At the beginning of a SPLE, the costs are higher compared to a single system, because of the common platform development and the reusable parts. Subsequent products can be made more economically because of the commonalities in the product line. This means a company has to make an investment to create the platform before it can reduce the costs per product. Figure 2.1 shows this behaviour. At the beginning, the accumulated costs are higher, but after the break-even point of approximately three different products, the software product line is the better strategy.

The use of a common platform and the reuse of artefacts in numerous products lead to the next point.

Increase quality: The platform is reviewed many times and tested in several products, so finding errors in a product increases quality in all products of the product line.

Reduce time to market: This aspect is important in many business areas because reducing the time to market may be a key motivation. Figure 2.2 shows the different traces of time to market with both strategies. Similar to the costs (Figure 2.1) the product line shows the advantage after a certain number of implementations.

Reduce maintenance effort: As a result of the architecture, the maintenance effort is reduced, because there is only one common platform and the same artefacts for all kinds of products.

Coping with evolution and complexity: Implementing one new artefact for the platform gives the opportunity to put it in all other products of the SPL to set trends.

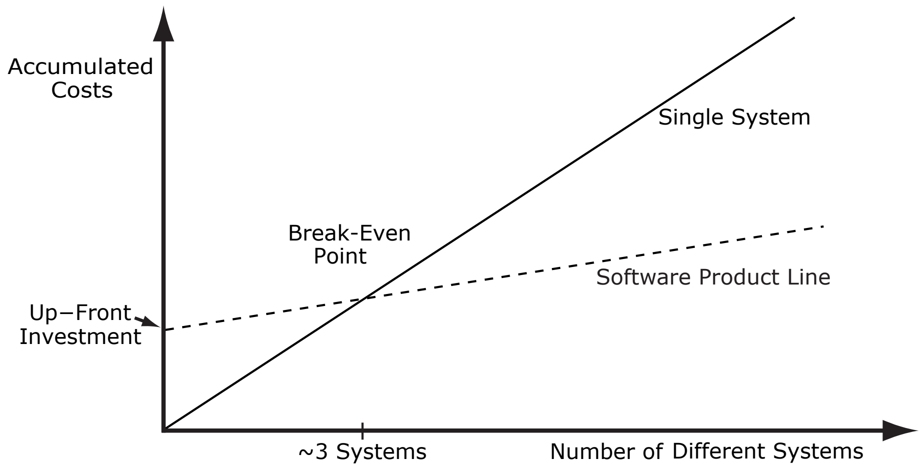


Figure 2.1: Costs for developing single products compared to SPLE [PBvdL05]

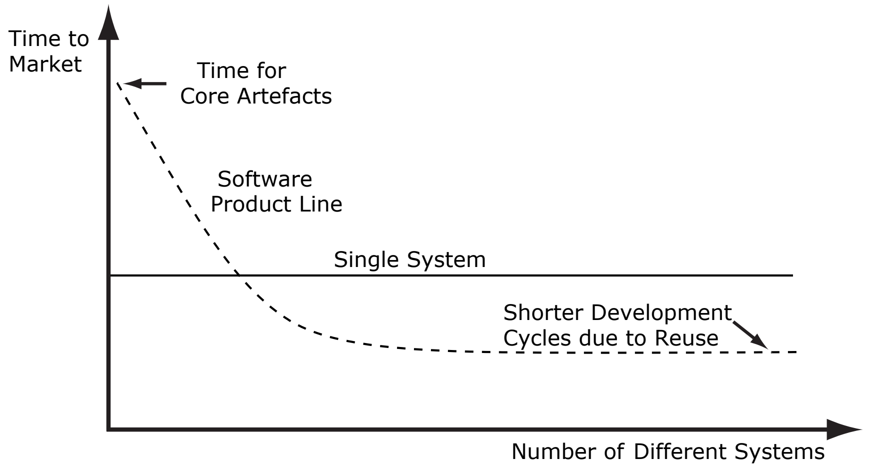


Figure 2.2: Time to market with and without SPLE [PBvdL05]

The reuse of parts reduces the complexity, because the development with higher abstraction of already implemented parts is easier.

Benefits for the customer: Last but not least, there are benefits for the customers who get a product which is adapted to their needs with an adjusted price.

Companies dealing with software product line engineering tell about all these motivating factors (see 2.1.6).

2.1.3 Terminology

Before dealing with the development processes, a uniform terminology has to be defined, because the literature uses two notations for the same engineering discipline. In Europe the term *software product family* is used more often than the American notation *software product line engineering* (SPLE). The reason for this is the independent start of the conference series in Europe and America. Today these conferences are merged and named *Software product line conference* (SPLC) so the american notation is adopted [PBvdL05].

Additionally, there are some other naming conflicts like *core asset development* which is the same as *domain engineering* and *product development* as a synonym for *application engineering*. In this master thesis, the terms *software product line engineering*, *domain engineering* and *application engineering* are used. Only the illustrations used from the Software Engineering Institute have their original American naming scheme.

Table 2.1 lists all essential differences in the terminology.

Product Line	Product Family
Product Development	Application Engineering
Core Asset Development	Domain Engineering
Product	Customisation
Business Unit	Product Line
Software Core Assets	Platform

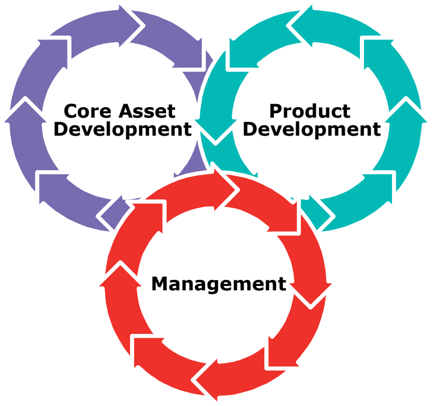
Table 2.1: SPLE - Synonymous terminology [LCP+00]

2.1.4 SPL development processes

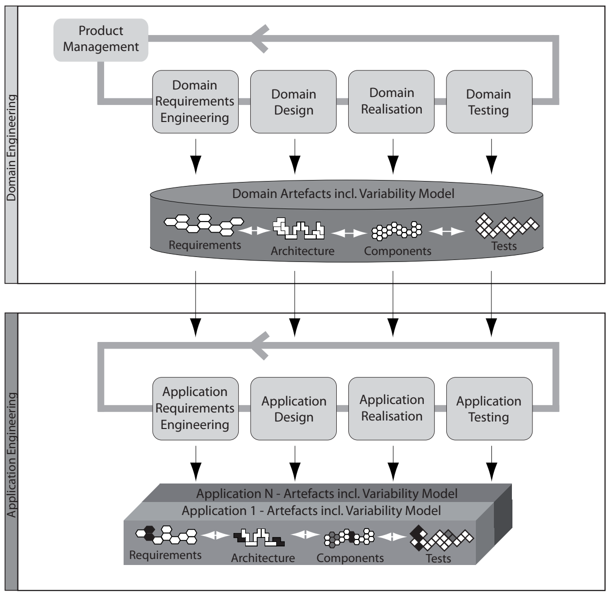
The paradigm of software product line engineering differentiates between two respectively three processes[PBvdL05; Ins08; Sch07]. These processes are all iterative and partially parallel.
Domain engineering is the process which is responsible for creating the platform, defining and realising all the commonalities as well as variabilities of the software product line. Section 2.1.4.2 deals with this process. The second process is called *application engineering* where all the applications are built by using the domain artefacts and exploiting the variability of the software product line. Detailed information is given in Section 2.1.4.3 *Management* is the third process and deals with the economic aspects of the software product line. This part of the development is handled as own process in the SEI framework (Figure 2.3(a)) whereas it is a part of domain engineering in the other framework. Detailed information is given in Section 2.1.4.1. Figure 2.3 shows two different graphics for the development process, called the framework for software product line engineering. They differ in few points, but both can be found in the literature and both introduce the main processes.

Another two interesting graphics illustrating the goal of SPLE are shown in figure 2.4. Figure 2.4(a) gives an idea how strategic reuse is realised. The business strategy is as important as the technical strategy to reach an efficient reuse level.
Figure 2.4(b) shows again an illustration of software product lines with all three processes.

The business goals in the application domain lead to an architecture representing the base for creating products with components. As the graphic shows, the business strategy is on the top. Every process follows the strategy.

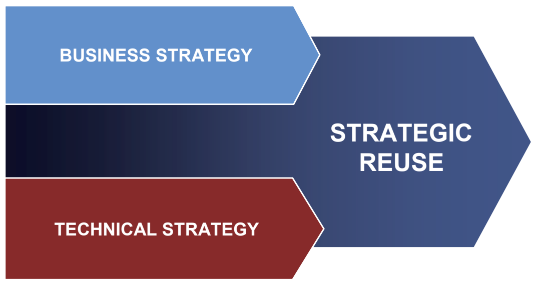


(a) SPLE framework (SEI [Nor02])

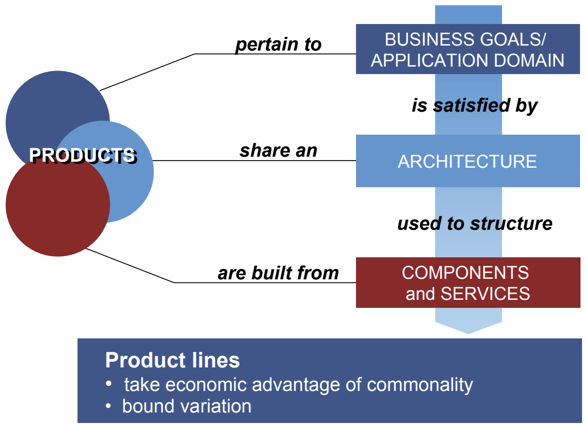


(b) SPLE framework [PBvdL05]

Figure 2.3: Development process for SPLE



(a) Strategic reuse [Nor07]



(b) SPLE process[Nor07]

Figure 2.4: Development process for SPLE

2.1.4.1 Management

As already mentioned, the management process involves economic aspects of the software product line. The business strategy and the product portfolio are the main parts. The company goals are the base for the so called *product roadmap* determining the ongoing and future set of product types, the commonalities and the variabilities. The roadmap also defines a schedule of market introduction. For software product line success, the management must be closely linked to the other processes [PBvdL05; Nor02; CJNM05; Sch07]. This behaviour is illustrated in Figure 2.3(a).

The management also controls the iteration between application and domain engineering to stay on the roadmap. In traditional software engineering, the management is responsible for creating a single result in a defined amount of time, so product management in product lines differs in some points from single systems:

- The goal is to generate a complete product portfolio (product variants / roadmap)
- The product variants are similar
- The platform for the product variants is crucial because it affects all products

An often used definition of product management in software product line engineering is:

Product management is the sub-process of domain engineering for controlling the development, production, and marketing of the software product line and its applications [PBvdL05].

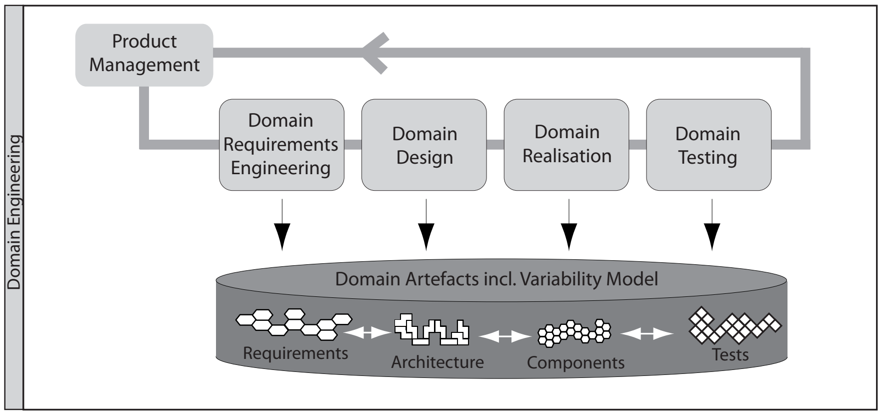
The products must make the best use of the domain artefacts and variabilities and the domain artefacts must be feasible for the products of the roadmap. Therefore, the management is involved in the whole process of software product line engineering [CJNM05] (See Figure 2.3(a)).

2.1.4.2 Domain engineering

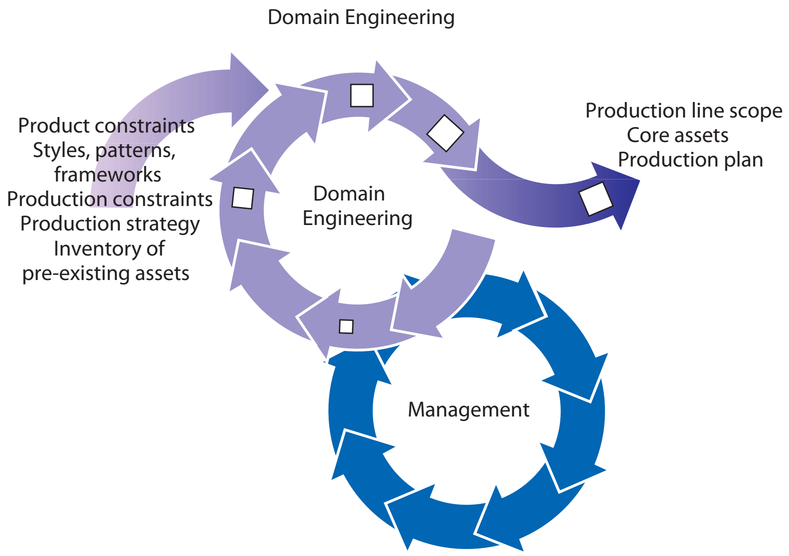
The process *domain engineering* defines the commonality and the variability of the software product line. In cooperation with the *management* (Section 2.1.4.1), the set of applications will be selected. Now the reusable artefacts can be constructed to determine the variability. Both figures 2.5 illustrate domain engineering [PBvdL05; Nor02; Sch07]. Figure 2.5(a) shows subprocesses which will be discussed now. The input for *domain requirements engineering* is the roadmap from the management which causes reusable requirements as output. The format of these requirements is not specified and may be textual or anything else. These requirements are not for special applications, but for all possible applications of the product line. In this process, the chosen approach to start a product line must put into practice. Further information about the starting-approaches can be found in Section 2.1.4.4.

The *domain design* subprocess takes all requirements as input and creates a reference architecture. During this stage, technical reasons may influence the internal variability.

After defining the reference architecture, *domain realisation* is done. Reusable components can now be designed and implemented. It is possible to test the implemented components against their specification to reduce the later testing of the whole application. This subprocess is called *domain testing*. All the individual subprocesses discussed are iterative. Figure 2.5(b) shows the same processes in another way. The output is again reusable artefacts (core assets), which the company uses to implement all products of the roadmap [CJNM05]. Production constraints influencing the development may be external or company specific standards which must be applied to all products [Nor02].



(a) Domain engineering [PBvdL05]

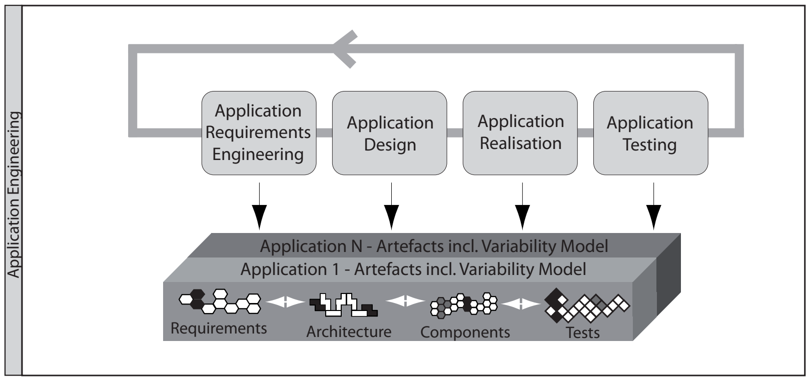


(b) Core asset development [Nor02]

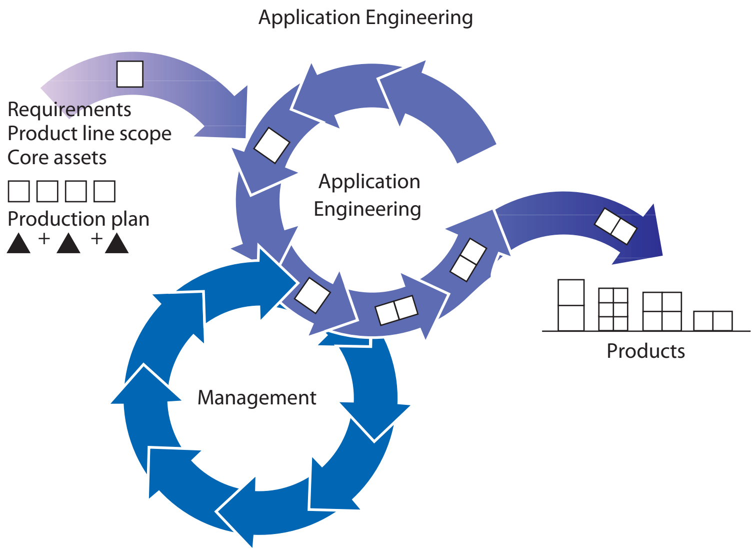
Figure 2.5: Domain engineering

2.1.4.3 Application engineering

The *application engineering* process tries to achieve a high reuse of the domain artefacts during the creation of a new application [PBvdL05; Nor02; Sch07]. Exploring the variabilities of the product line should lead to many resulting applications. Figure 2.6 shows two views on application engineering.


Application Engineering
Application Requirements Engineering
Application Design
Application Realisation
Application Testing
Application N - Artefacts incl. Variability Model
Application 1 - Artefacts incl. Variability Model
Requirements
Architecture
Components
Tests

(a) Application engineering[PBvdL05]



(b) Core asset development [Nor02]

Figure 2.6: Application engineering

This process is also divided into subprocesses as illustrated in Figure 2.6(a).

Application requirements engineering is the first step in developing the specification of the application requirements. Differences between the reusable domain artefacts provided and the necessary requirements can be found, because the domain artefacts are the only

input for this subprocess with the product roadmap kept in mind. The result of this subprocess is a requirement specification for an application.
Modifying the reference architecture to fit the needs of a specific application is done in *application design*. The reference architecture and the specification are the input parameters and the output is an adjusted architecture for a single application.
Now the application is created in the subprocess *application realisation*. Reusable components are selected and configured. It is also possible to implement application specific parts. The result is a running application which will be tested in the last subprocess called *application testing*.
Figure 2.6(a) illustrates the set of applications resulting from application engineering in the software product line. The graphic 2.6(b) gives another view on application engineering by taking again the domain artefacts in respect to the roadmap to create customised products. Here an explicit connection to the management is shown.

2.1.4.4 Approaches for software product line development

A company decides to develop a software product line, but how to get it started? Basically, there are three different approaches starting with a software product line [Nor07; KAG+07].
These approaches depend on the strategy of the user or company and there is no formula to select the right one.

Proactive
In this case, the Domain Engineering Process is the first step. The development process must take into account all feasible products of the Software Product Line. The complete set of all artefacts is developed from scratch which is, of course, a risk for the company. This needs predictive knowledge and a clear strategy. After the SPL has been constructed, the new products will come to market quickly with a minimum of coding effort by exploring the variability.

Reactive
The reactive approach starts with just one or a few more products. These are used for domain engineering and for further products. This leads to lower costs at the beginning of a new project compared to the proactive approach, but the architecture and the core artefacts must be robust, extensible and appropriate to future needs.

Incremental / Extractive
This approach starts with existing software products or systems. Then reusable artefacts are extracted to create a first version of a software product line. Further products extend the artefacts incrementally.
Companies dealing with risky products may develop the products this way.

2.1.5 Variability

[PBvdL05] defines variability this way:

Documenting and managing variability is one of the two key properties characterising software product line engineering. The explicit definition and management of variability distinguishes software product line engineering from both singles-system development and software reuse.

Variability is defined during the domain engineering process where it is refined in all subprocesses (see Figure 2.5(a)). Variability describes the ability of domain artefacts to be used in different applications of the product line roadmap [BC05; BFG+01; PM06]. Two important terms are variability subject which is a variable item in the real world and variability object, as a specific instance of the subject. For example, a feed system would be the subject and a two meter long and one meter high conveyor band would be the object. Further abstractions which are interesting related to variability in software product line engineering are variation points and a variant. This is the representation in a context of the variability subject (variation points) and the variability object (variation). A simple example makes it more clearly. Let us take again the variation subject feed system with objects like conveyor band and a rotary table. The context in the demonstration is a logistics system so the variation point is feed system and a variation could be a rotary table for the logistics system.

The variation points and the variants are used to define the variability of a software product line. When creating a SPL, first, all variation subjects must be declared, then the variation points have to be worked out and finally the according variations. To document the variability, the following questions have to be answered:

What varies? The mapping between the real world and the variation points should be documented.

Why does it vary? The reason can be internal like technical constraints or external like laws, standards or needs of stakeholders.

How does it vary? All possible variants should be documented and linked to the domain model elements.

From whom is it documented? There is a difference between extern and intern. Some documents are only for internal use and some are relevant for the stakeholder.

Figure 2.7 shows a graphical notation to model variability. It defines the variation points, the variation and the dependencies between them. Dependencies are divided into variability and constraint dependencies.

A short example in the context of this master thesis is given in Figure 2.8. Sample variations like rotary tables and conveyor bands of the feed system and dependencies between single variations can be seen. Dependencies between variations are, for example, the rack servicing unit which is needed if a crane should be part of the system. There is also a dependency between a variation (crane) and a variation point (control) which implies, that a crane can only be realised with a control. The control itself may be either the basic or the advanced implementation (alternative).

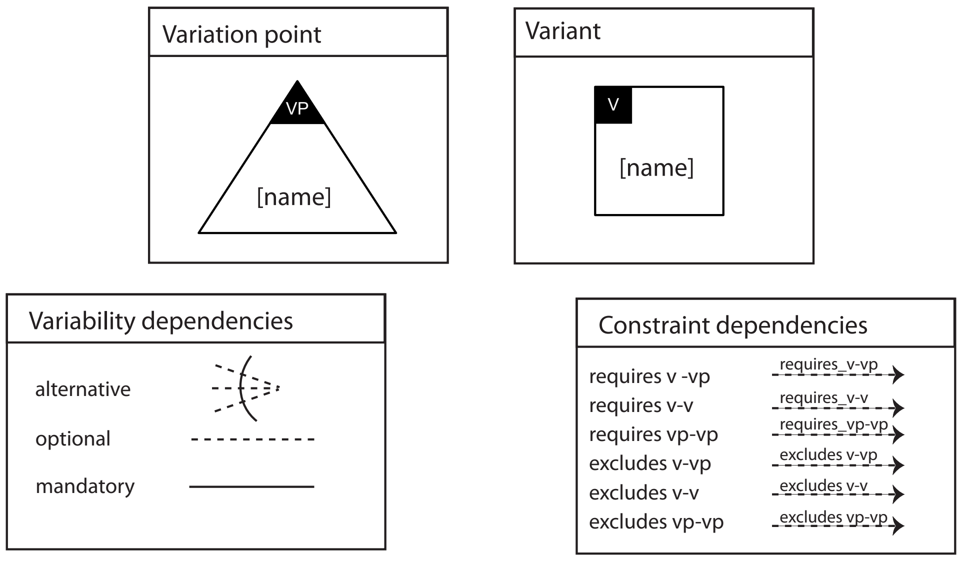


Figure 2.7: Graphical notation for variability [PBvdL05]

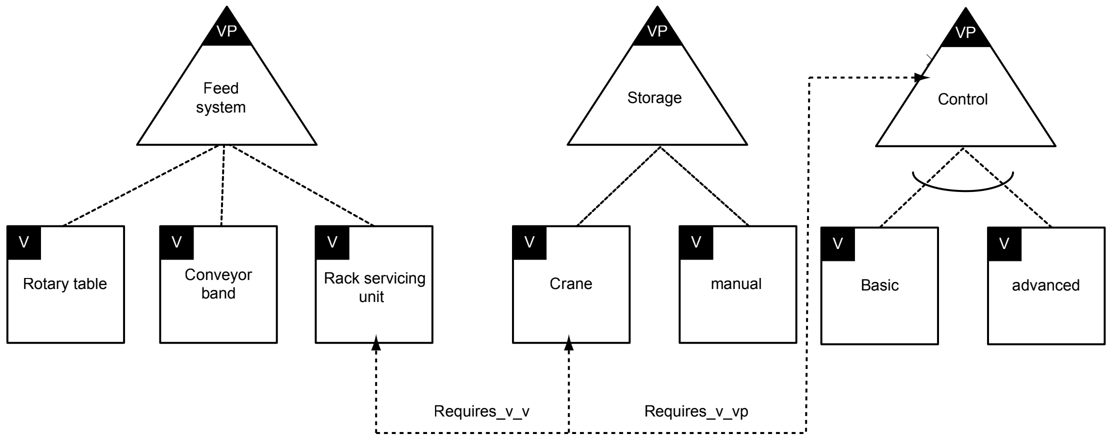


Figure 2.8: Graphical notation example

In big projects, there are thousands of variation points and variations, so the organisation and optimisation is a challenging research area [LP07]. Several tools provide different selection methods where variability can be easily handled. For example, dependencies are checked during the selection process [NTB+08] (see also 2.1.8). Variation management is a main-criteria when selecting a tool for big software product lines.

2.1.6 SPLE in action

Now the key principles were discussed, but who uses software product lines already and what are the problems in practice. Many companies switched to SPL development to create more products in a shorter time, for example Nokia®. Here the use of SPLE is obvious because of the properties of a mobile. Nokia® increased the production of different mobiles from originally 5 per year to 25-30 per year [Nor07; Bos05]. In 1999 the product line for Nokia® browsers and tools started [Jaa02].

Starting in the year 2000, Bosch Gasoline Systems® tried to reach new high cost sensitive market segments (Engine Control Unit: ECU). They had one standard platform which had to deal with small products as well as with high end systems. That was the problem, because constraints like power consumption in tiny projects with their complex platform occurred. Bosch introduced a software product line which took 6 years. Now software product lines are successful within Bosch, and a lot of change happened within the management structure [TMKG07; STB+04].

SPLE is also a topic in the development of mobile games because the mobiles provide different platforms where the games should be adapted, also screen resolutions and the keypad are relevant [ACA08].

Another interesting project in respect of this work is the use of SPL for metal processing lines which are PLC controlled [SMBU07]. Each single machine is PLC-controlled and there is a *master-controller* handling the communication between them. After introducing the product line, changes to the metal processing line, e.g. other finalisation with thinner metal were easy to realise.

2.1.7 Hall of fame

The Software Product Line Hall of Fame honors software product lines that can serve as examples to software product line developers [Wei08].

At each Software Product Line conference, the audience nominates several systems or projects for induction to the Hall of Fame [WCKK06; Ins]. The main goal of this procedure is to find and show the best examples in the field to improve SPL in practice. All members of the SPL hall of fame are listed at `http://www.sei.cmu.edu/productlines/plp_hof.html`. For example the Hall of Fame Inductee from the 11th SPLC (2007) was Bosch with Gasoline Systems Engine Control Software, as mentioned in Section 2.1.6.

There are 5 criteria for the election to the SPL hall of fame:

- There is a clear separation between whether a software is part of the product line or not.
- The commonalities and variabilities of the SPL are known and there is an underlying design for the SPL.
- The SPL has had a strong influence on other product lines. It showed how things can be done and maybe others have stolen or borrowed ideas and concepts to create their own SPL

- Commercial success of the SPL.
- Sufficient documentation of the SPL should be available.

The four judges of the SPL hall of fame are:

- David M. Weiss, Avaya Labs Research (Chair)
- Paul C. Clements, Software Engineering Institute
- Kyo Kang, Pohang University of Science and Technology
- Charles Krueger, BigLever Software

2.1.8 Supported product line approaches and tools

As already mentioned the development process of an SPL is a difficult and expensive task. So the motivation is high to buy and use tools which reduce these costs. There are several commercial tools available, like GEARS from BigLever, or pure::variants professional from pure systems [Kru08; Beu08]. GEARS is an integrated SPL development framework, which can be used during many stages of SPL engineering by extending the current software engineering toolset of a producer. More information, web seminars and some downloads are available at `http://www.biglever.com`.

The second commercial tool which is presented in this work is pure::variants from pure systems. It is an eclipse based application that supports domain engineering as well as application engineering so it handles variability in all steps of an SPL. Pure::variants also provides integration to other development tools, because this is very important for this kind of software. Further information can be found at `http://www.pure-systems.com/pv`.

Research tools like *DOPLER* (Decision-Oriented Product Line Engineering for Effective Reuse) also exist. This tool was developed by the Johannes Kepler University Linz [DRGN07].

An example of free software is the FAMA framework (FAMA FW) which is available under the lesser gnu public licence (LGPL). It is an eclipse plugin which manages variability and provides solvers and reasoners to explore the product line [TPB+08]. Additional information can be found on `http://www.isa.us.es/fama/`.

FLiP is a similar eclipse plugin dealing with SPL [ACN+08].

There are more applications available dealing with software product line engineering, but the most popular are listed above. Section 2.1.8.6 shows other metamodeling tools which are able to handle SPLE.

There are numerous approaches which support software product lines. The most salient are Domain Specific Languages (DSL), Generative Programming (GP), Model Driven Engineering (MDE) and Domain Specific Modeling(DSM) [EV05]. All these approaches are model-driven and not code centric, which is the other software development strategy and the main idea is to get a higher level of abstraction. Models are also used to hide implementation details. The use of such a development strategy is recommended if many products will be generated within this scope and a basic skill in this area is given [KT08].

The next Section gives a short introduction to all principles and shows that they share many techniques and concerns. Domain specific modeling and domain specific languages are discussed in more detail because they are important for the implementation part of this master thesis.

2.1.8.1 Domain specific languages

Domain specific languages are nothing new and sometimes they are called micro- or little languages. Most of the older programming languages like Cobol or Fortran came from specific domains. Later they have evolved to be more general to solve a broader problem-spectrum [vDKV00].
The most common definition of a DSL is :

A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [vDKV00].

The language is either textual or graphical but DSLs are not only programming languages, they can also act as specification language. If a DSL contains a compiler which generates executable code, it is called application specific language and the compilers name is application generator. A DSL is also able to generate other output like documents. An example therefore is LaTeX. Today many domain specific languages like SQL oder HTML exist. When browsing through the IEEE database, there are many publications about DSLs for various domains. The design of a DSL includes 2 basic steps:

Analysis: Identify the problem domain and explore it to gather as much information as possible. Then this knowledge is clustered in a couple of notations. The designed DSL should describe applications in the problem domain. This process is also called *domain engineering* as in SPLE.

Implementation: The designed constructs are implemented and provided to the user, as library or repository. After creating DSL based programs, a compiler translates them.

A guide for building a powerful DSL is not published yet, but useful information on what not to do can be found in [KP09].
The advantages of using a domain specific language are that problems are handled in the problem domain. Because of the high abstraction, domain experts are able to create programs without real programmer knowledge. They are self-explaining and documenting and improve the productivity but there are disadvantages too. For example, the design and setup time is high and cost intensive so an efficient management should think of the scope and the business strategy, similar to SPLE.

Domain specific languages are able to implement a software product line.

2.1.8.2 Generative programming

Generative programming is also able to implement software product lines and generate applications with a given definition. In most cases these definitions are made with domain specific languages which take reusable components within the variability. GP works not in the problem domain but rather in a feature perspective [ABM09; EV05].
The mapping between the problem- and solution-space is the key principle in generative programming [Cza04]. This perspective has at least two different views. The configuration- and transformational-view. The configuration view describes a rule-based transformation of a selected feature-set in the problem domain to implemented components in the solution space. The transformational view bases on a DSL in the problem- and an implementation language in the solution-space as well as the mapping between them.
In a simple summary, it can be stated that one or more DSLs, a mapping and components are the main parts in generative programming.

2.1.8.3 Model driven engineering

Model driven engineering defines, as the name says, the model as the basic concept. It is an initiative of the Object Management Group (OMG) [Gro09]. The definition of a model is:

The model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system [BG01].

These models should be easier to use than the original system and all models conform to a metamodel definition. The metamodel is the specification which defines a set of concepts and relations between them.
Model driven architecture (MDA) was the beginning of MDE and based on the Unified Modeling Language (UML) but the new trend in MDE is pretty close to a DSL [KT08]. MDA enables a specification of a system which is independent of the platform (PIM) and a platform specific variant called platform specific model (PSM) or independent of all computing details (CIM). The transformation between PIM and PSM can be done automatically.

2.1.8.4 Domain specific modeling

Domain specific modeling can be seen as MDE with a DSL and it is similar to generative programming. The 3 integral parts are as mentioned a DSL, a framework and a generator. DSM is unsuitable for new projects, as are many other modeling techniques [KT08].
Figure 2.9 shows how developers have bridged the gap between the idea and the finished product by raising the level of abstraction. The first step is to solve the problem in the corresponding domain with a high abstraction level, then the mapping to code is done. In case of UML or another general purpose modeling language a mapping to the specification of UML is done before generating the code.

With regard to business values, DSM increases the productivity by about 300-1000% in comparison with general purpose modeling. For example, Nokia applied a DSM solution as SPL and reported a 1000% increase in productivity (see Section 2.1.6). The motivation

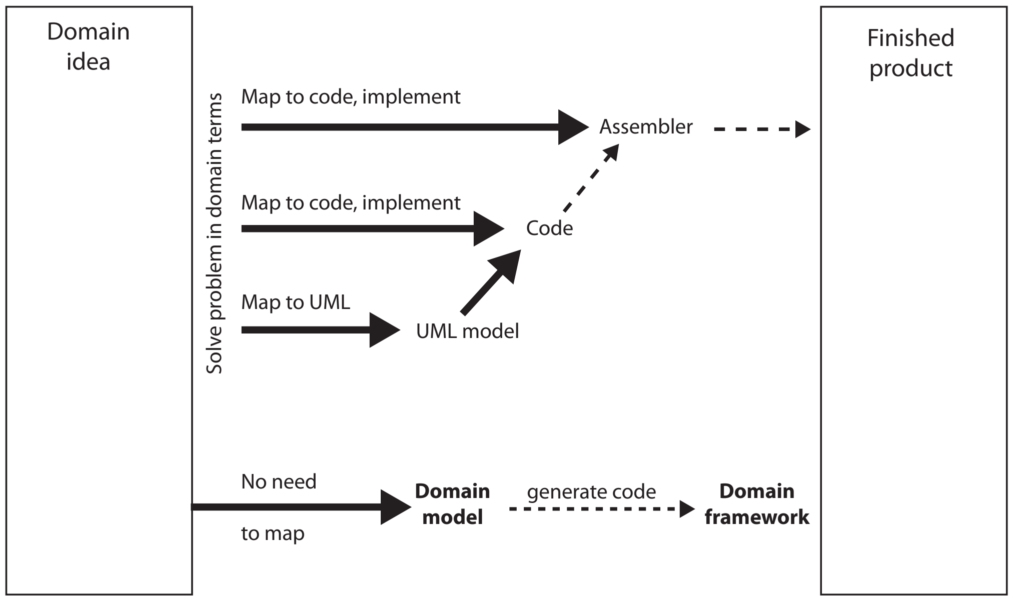


Figure 2.9: Bridging the abstraction gap from idea to implementation [KT08]

for DSM is similar to SPLE and all factors from Section 2.1.2 apply to DSM too.
The characteristics of DSM are [KT08]:

Narrow focus: The modeling can be used only for this domain and its applications which means a narrow problem and solution domain.

High level of abstraction: The models map exactly to the problem and the generators to the solution domain.

Full code generation: Static and dynamic code is created by the generators and one source model is able to create multiple targets. For example: One model is the basis for the code, suggestions, documentation and other output. See chapter 3 for the implemented targets in this work.

Representation: In most cases, graphical models are closer to the problem domain than text.

Large number of users: Because of the simplicity not only programmers are able to use DSM.

2.1.8.5 Comparison: UML vs. Domain specific modeling

The question regarding the difference between UML and DSM often arises although the names should give the answer. The main difference is the narrow focus of DSM and the tight mapping of the models to the problem space. UML models often result in large and

complex abstractions because a unified modeling is always more complex than a domain specific but there are of course advantages for UML.
Another big gap is the code generation, which is state of the art in DSM with good quality, because the generator fits to the domain. It is obvious, that one generator for all applications does not exist [KT08].
A study on maintainability shows, that DSM generated code is easier to handle than UML-made code [CRR09].

2.1.8.6 Metamodeling tools

In contrast to the tools for SPLE, other available software can be used which follows the SPLE approaches listed in Section 2.1.8. These tools are called metamodeling tools and the following pages will present the most popular. MetaEdit+® is described in detail, because it is used for the implementation.

DSL tools for Visual Studio: Microsoft® DSL tools are available as an extension for the VisualStudio. The current version is 2008 but for Visual Studio 2005, there is also a plugin called DSL tool. This extension provides all tools, which are necessary to define a domain specific language, develop models with this DSL and generate code of those new application models (metamodeling environment / modeling environment / code-generation)[GVDV08]. The following descriptions are for DSL-Tools 2005, but there are only marginal changes to Version 2008 [Mic08b]. The meta-modeling process is done in three steps. First the *domain model notation*, representing the structure is defined visually. It shows the inner structure and the relationship between classes. Possible constructs are classes, properties, inheritance, embedding and reference. Embedding means, that a class X is a subclass of another, whereas a reference implies the use of another class. Links between classes are defined with relationships. Here, for example, the cardinality or inheritance is defined.
After the domain model notation, *domain constructs* have to be defined, which is a challenging task, because everything has to be done in an XML notation. The last step is to create relationships between *domain notation and constructs*. There is also a wizard which provides several templates for languages (task Flow/ class diagrams/ minimal language/ component models).

After creating the metamodel it is possible to model a specific application. The last action is to start the generation of real code. Here, Microsofts® DSL tool is based on the transformation of templates. These templates may include control-, directive- or textblocks [KMML07][Mic08a]. Validations of the model must be written in C#.

Generic modeling environment: The Generic modeling environment (GME) is a Windows based, programmable graphical modeling tool suite (open source). It has been developed at the Institute for Software integrated Systems (ISIS) at Vanderbilt University. This metamodeling language is based on the UML class diagram notation [fSis09; Siv08b]. The constraints are written in Object Constraint Language (OCL) and managed in the constraint manager. All models are stored in a relational database as XML files. GME can be extended by writing components in any language. The metamodel concept is based on folders which contain models, Atoms, References, Connections and Sets. GME is COM based, so any COM enabled language can be used for model interpretation. (COM is a technology for inter-process communication on Windows platforms)

Generic Eclipse Modeling System: The Generic Eclipse modeling system (GEMS) is a part of the Eclipse Generative Modeling Technologies (GMT) project. GMT tries to create prototypes in the area of MDE [Ecl09b; Ecl09a]. GEMS is a plugin for the eclipse IDE and like the platform is open source on the Eclipse license. GEMS is also developed at Vanderbilt University Institute for Software Integrated Systems (ISIS) and other companies. The main goal is to bridge the gap between existing metamodeling tools like GME and Eclipse modeling techniques like Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF). GEMS provides a graphical language for the metamodel which contains entities, attributes, relationships, connections and constraints. These are written in Java, OCL or Prolog and can be used as triggers. Java based code generators are also available.

Domain Modeling Environment: The Domain Modeling Environment (DOME) is an open-source multiple platform tool under the GNU license and is written in Smalltalk [Hon09]. DOME supports graphical developing, analyzing and transforming models. Built-in notations like UML and Petri-Nets can be extended or a totally new definition can be created. New concepts are made with the DOME tool specification language. Components are objects, classes, property and relationship definitions and connector types.

openArchitectureWare: openArchitectureWare (oAW) is a modular MDD/MDA generator framework which evolved from a Sourceforge project into a rich toolkit for model driven software development [ope09; Ecl08]. oAW moved to the Eclipse GMT project (see 2.1.8.6) with several sub-projects like Xpand, Xtend and Xtext as a result. oAW was not seen as a "prototype and research project" so it had to leave GMT on 1st December 2008, but now it is back as an Eclipse project. oAW is suitable to generate, transform and check models [RBKS07]. Because of the evolution at Eclipse it has strong support for the Eclipse Modeling Framework (EMF) although other models are possible too. The metamodel is called Ecore. The validation is done using the first-order-language CHECK which is an equivalent to OCL. To generate code with oAW, all models have to be combined into one model using a model to model transformation which is provided by the language EXTEND. After combining the models, the template language Xpand can be used for code-generation.
XText is also a part of the oAW framework which supports a textual DSL development. Figure 2.10 was taken from [Sen07] and modified to illustrate the discussed basic architecture of oAW.

MetaEdit+: MetaEdit+® is an integrated environment to generate modeling tools and generators for an application. Between 1988 and 1995 this tool was developed as a prototype in the Syti and MetaPHOR research projects at the University of Jyväskylä. The commercial tool has been available since 1993 and the current version 4.5. MedaEdit+® is a repository based client server tool which is available for all major platforms (Windows, Linux, MacOs X, HP-UX, Solaris). The tool is divided

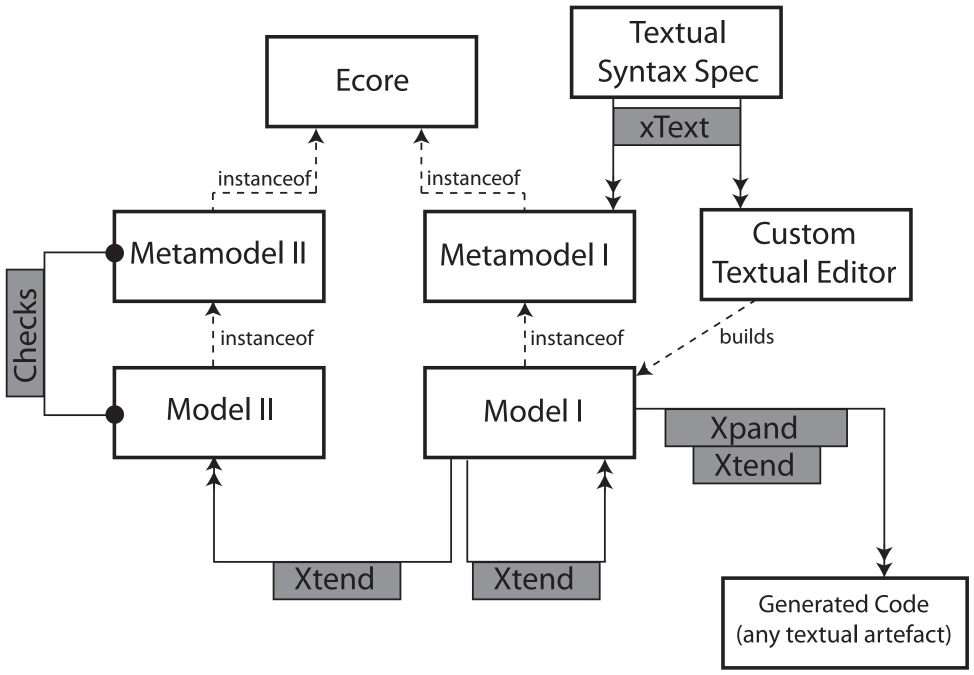


Figure 2.10: oAW architecture (modified [Sen07])

in two main programs, the MetaEdit+® Workbench and the MetaEdit+® Modeler. In the workbench, a new domain specific language can be defined, whereas the modeler follows these definitions. Figure 2.11 shows the architecture of MetaEdit+®. The software is network-based, so all definitions and the object repository can be used parallel in the network to create new applications.

MetaEdit+® applies the GOPPRR metamodeling language. This is the acronym for graph/ object/ property/ port/ role/ relationship. GOPPRR extends GOPRR with the port-concept [Siv08a],[Siv08b]. The following list describes all types of the GOPPRR language. The order is mixed to make it more logical [Met08],[Siv08b].

Graph: The graph is the collection of objects, relationships, roles and shows the connections between them. Modeling applications are done in the graph.

Object: An object is the main-element in the graph and can be placed on it's own.

Relationship: A relationship is a connection between objects. They are connected via roles.

Role: The role defines, how an object takes part in a relationship.

Port: A port is an optional but detailed specification for a connection at an object and constraints can be made here. A conveyor has for example two ports.

Property: All other types can have properties such as name and description, or other objects.

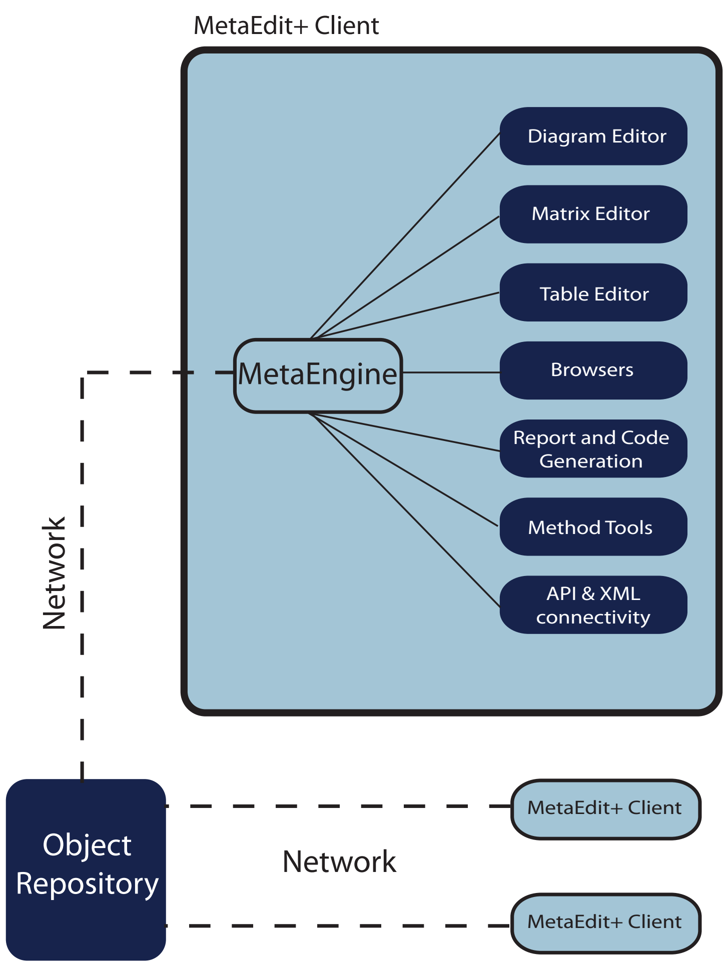


Figure 2.11: Architecture of MetaEdit+® [PK07]

Section 3.2.2 should clarify all types.

MetaEdit+® allows two ways of defining a new language, either graphical or form based. Graphical means drawing diagrams like UML class-diagrams with objects and properties. Form based generation works with dialogs for each language type. Because the definition is saved as XML, it is possible to continue a graphical language with form-editors. Graphical metamodels are more suitable for smaller languages or to document the modeling language. Form-based metamodeling is appropriate for layered structures and complex references [PK07].

In MetaEdit+® there is fixed set of possible constraints.

Object connectivity i.e. amount of objects in a relationship

Object occurrence i.e. specified number of objects in a graph

Ports involved in binding i.e. ports of a certain type must have specified values

Property uniqueness i.e. unique values for a specified property

The code-generation is done by crawling through the designed models, extracting the information and forming an output in a predefined format. This predefined format can be achieved by creating a custom generator. The generator is defined with MetaEdit reporting language (MERL®). MERL® is a simple scripting language which is tailored for creating code generation definitions. Browsing through the model is easy with several commands and the output can be launched in an application or stored in a file.

MetaEdit+® also provides an API for communication. It implements a SOAP server, so other applications must implement a client to send SOAP-calls. SOAP means simple object access protocol which is a network-based protocol to exchange data in an XML form. Therefore, MetaEdit+® functions are available in almost any programming-language. Functions may access and change conceptual elements (graphs/ objects ...) which is not possible with MERL scripts. By writing external programs, the fixed set of constraints can be extended.

At the vendor-website `http://www.metacase.com`, there are many documentations and tutorials available [Met08; Tol06; Siv08b; PK07].

A short evaluation is done in A.2.

2.1.8.7 SPLE tooling benefit analysis

The performance of an SPL implies an efficient domain- and application engineering process, the latter is more important. A way to improve these processes is visual tool support. This Section deals with a domain specific benefit analysis for SPLE software-tools in respect of the implementation.

It is difficult to compare all tools for software product line engineering because there are many approaches for the implementation. For each single approach there are also several tools. It depends on the domain and the output of the application. In cooperation with the master thesis of Andrea Leitner, a list of criteria is elaborated to rate SPLE tools [Lei09]. Some of the criteria were taken from existing papers [DSF07; DRGN07; LCP^{+}00] and new ones were added. To see the whole list please see appendix A.1. Generally, there are three main groups. Criteria regarding the product line, the management and technical criteria. Examples of SPL criteria are attribute management or product derivation. Management deals with traceability management, impact analysis and reporting. Technical criteria are for instance access modes, usability or accessibility.

Regarding a complete mapping from the problem to the solution space, domain specific modeling is an appropriate approach for a software product line in this specific domain. Table 2.2 shows a comparison of widely used tools for SPLE. MetaEdit+$^{®}$ is a tool for a DSM approach. (see Section 2.1.8.4)

The weights are domain specific for the the logistics application of the master thesis. This is just a tiny subset of available tools, but they are the most popular ones. All tools of Section 2.1.8.6 could also be a part of this table. Based on a comparison of [Siv08b], tests with the software and positive industrial feedback, MetaEdit+$^{®}$ was elected as the DSM exponent.

Table 2.2 shows the result of the comparison with the specific weights for this domain. MetaEdit+$^{®}$ is most suitable, followed by pure:variants.

As mentioned, the weighting of the single criteria depends on the application. In this work a rich attribute management to manage thousands of objects is not important, because there are only a few different parts. To find these parts, the domain specific language is used with different symbols. In contrast, product derivation is important but this should be covered by the DSL too. An essential role in this domain is rule and error checking to create reasonable and correct systems.

For these reasons it is difficult to compare different tools of software product line engineering.

Further to the information given in Section 2.1.8.6, Figure 2.12 shows the simplified architecture of MetaEdit+$^{®}$. The key element is the object with properties, ports, relationships and roles.
The generators extract the information of the graph-model to create code or reports.

	Tools	pure:variants	Gears	Feature Modeling Plugin	XFeature	MetaEdit +
Criterion	Weight	Rating	Rating	Rating	Rating	Rating
Attributes Management	2	5	7	5	5	2
Feature and Variability Modeling	4	10	9	10	7	7
Feature Metamodel Maturity	2	3	1	8	6	8
Constraint Checking and Propagation	8	10	8	8	8	9
Product Derivation	8	10	9	7	7	9
Domain Engineering Management	5	7	8	5	5	5
Repository	3	6	9	6	6	6
Traceability Management	3	4	4	1	4	5
Impact Analysis	5	4	8	2	2	9
Reporting	1	8	9	2	2	9
Access Mode	10	2	1	2	2	9
Technical Environment	10	8	4	5	5	9
Usability	6	6	8	3	4	7
Automatic Filters	1	5	3	3	3	6
Tool Configuration	7	7	2	3	10	9
Extensibility	9	10	5	6	4	5
Flexibility	10	8	6	6	6	10
AOB	6	9	5	6	7	7
Benchmark	1000	723	566	506	542	**782**

Table 2.2: Results of the SPLE tool evaluation for this thesis

MetaEdit+® also offers a SOAP interface to read or change the model definition.

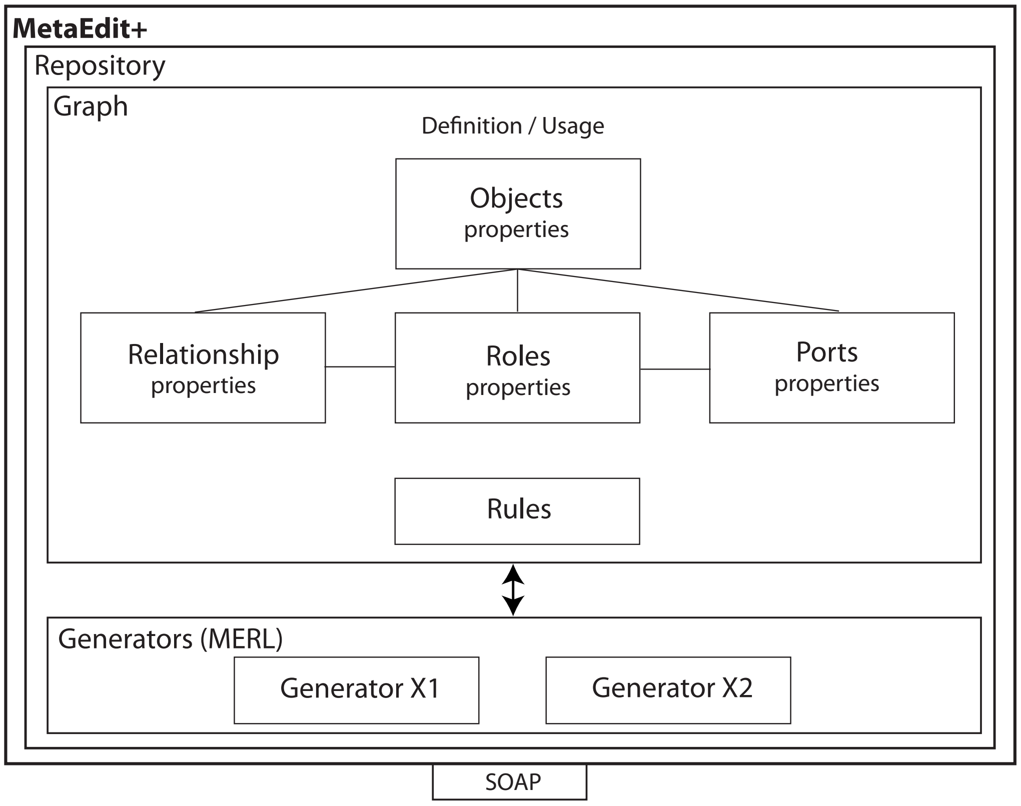


Figure 2.12: Brief overview MetaEdit+® architecture

2.2 Programmable logic controllers

This Section gives an introduction to programmable logic controllers (PLC) with the main focus on programming a PLC. Therefore, the standard IEC 61131 and the extension IEC 61499 are discussed and some approaches to new programming schemes and research projects which are relevant to this work are shown.

2.2.1 History and definition

The history of PLC is easy to understand and quick to tell. The complexity of controlling automation systems was growing very fast and most processes were critical. Functions for combinations were also needed (for example logical AND). The first implementations were relays and contactors. Later, the transistor became more interesting because it is tinier and more efficient. Then, the so called hard-wired-logic arose which offered functions to use but changes in the logic were impossible. That is why programmable logic controllers became popular. In 1968 the first concept of a PLC was introduced by General Motors. In 1969 the prime off the shelf PLC appeared (MODICON 084). The automotive sector was the first and most important field of application and from 1975 many vendors came into existence [ZLRK04]. PLCs seems to be the most cost effective way to implement realtime control in an industrial environment. In 1995 over ten million PLC had already been sold [Ver96].

The IEC 61131 definition of a PLC is:

A digitally operating electronic system, designed for use in an industrial environment, which uses a programmable memory for the internal storage of user-oriented instructions for implementing specific functions such as logic, sequencing, timing, counting and arithmetic, to control, through digital or analogue inputs and outputs, various types of machines or processes.

2.2.2 PLC platform and workflow

A typical PLC platform is built of seven basic blocks which can be seen in Figure 2.13. The blocks are the inputs and outputs, controlled by a processor. The processor includes a memory (RAM+ROM) and a communication interface. The memory can be programmed with an external device. Communication interfaces can be used to exchange information with other PLCs or extension modules. This communication may be based on different technologies like Bus systems, Ethernet or wireless links. A power supply is needed to provide the energy for all components [Bol06; ZLRK04; Kas08].

The simple basic execution of a standard PLC is shown in figure 2.14. First all inputs are scanned, then the program is executed and finally the outputs are written. The inputs control the logical behaviour of the program. This sequence will be cyclically repeated. Figure 2.14 is simplified and so watchdogs of modern PLC were omitted and communication takes place in the block *program execution* [Bol06; Kas08].

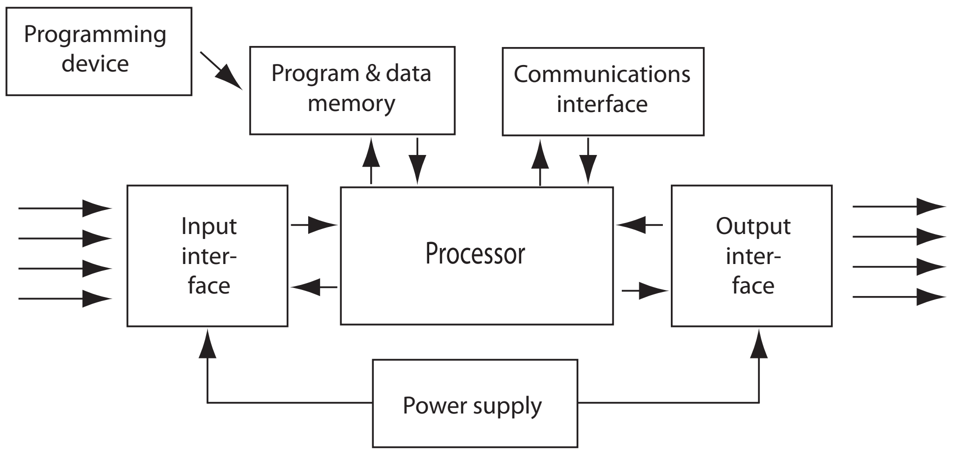


Figure 2.13: PLC hardware [Bol06]

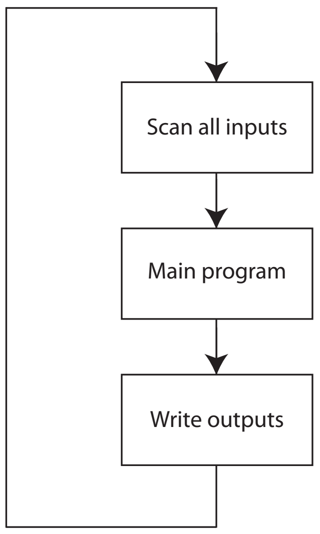


Figure 2.14: PLC workflow (modified from [Kas08])

2.2.3 The IEC 61131 standard

Programmable logic controllers became very important and many vendors tried to break into this new market. All companies had their own philosophy and so a broad spectrum of different systems came into existence. There was a need for a standard which was first addressed in the 1970s. The International Electrotechnical Commission (IEC) committed the first draft of the standard for programmable control systems, IEC 1131 in 1979. This draft was split into five parts. An international committee was set up 1987 to create part three which is called *Programming Languages for programmable controllers* (1131-3). This

Section was published in 1993, the second version with *Corrigendum* and *Amendment* in the year 2003.
Currently, a harmonisation between this standard and other relevant standards for distributed systems like IEC 61499 has been done (see Section 2.2.5 for more information about IEC 61499)[Ver96; MGM05; ope08; Bec98].

Today the standard IEC 1131-3 is adopted by the most global PLC players, some of whom have made a few changes. A deeper look inside this standard is given in Section 2.2.4. Because of the evolution, different notations of this standard exist. The German institute for standardization (DIN) started with DIN 61131 and the IEC adopted this name. IEC 61131 is popular and used in this work.

The whole standard is divided into eight parts and respects programmable logic controllers and their associated peripherals (programming tools / human machine interfaces). Table 2.3 shows an overview of all eight topics. More information can be found at

Part	**Short description**
1	General information / definitions
2	Hardware and tests
3	Programming languages (more details in Section 2.2.4)
4	User guidelines
5	Providing user oriented communication
6	Reserved part
7	Fuzzy control programming released
8	Guidelines for the application and implementation of programming languages

Table 2.3: IEC 61131 parts

PLCOpen [ope08]. PLCOpen is a global independent association supporting IEC 61131-3. Several technical committees within this organisation are dealing with the standard, functions, certification, communications, safe software and XML. The XML group tries to create XML schemes which are the basic for interoperability.

2.2.4 PLC programming (IEC 61131-3)

Part three of IEC 61131 deals with programming a PLC. An elegant way to split this part into two Sections is shown in Figure 2.15. The two parts are discussed in the following Sections in detail [ope08; vdW99].

The main goal of the standard IEC 61131-3 is, to make programs and systems interchangeable. This is the basic step for research and developing common architectures and schemes.

2.2.4.1 Common elements

Data typing

Programmable logic controllers use data types as do many other software systems. Pre-set types like boolean, integer, date and time or strings and user defined data types, so called

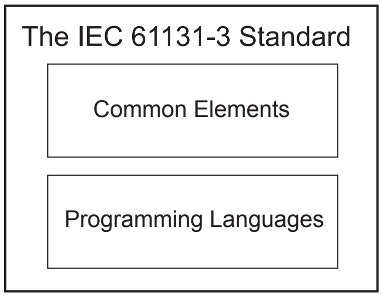


Figure 2.15: Parts of 61131-3 [vdW99]

derived data types. It is also possible to declare an input as a type. Common elements are not clearly defined, because, for example, the length of an integer varies. So on several PLCs the integer has a length of 32 bit whereas other systems use 16 bit. In this detail, the standard is ambiguous.

Variables

Variables are mapped to explicit hardware addresses (in- or outputs) in programs, resources or configurations. Detailed information on these terms is given in the following Section. This mapping creates hardware independency. The scope of the variables is local, as in other languages and an initial value may be set. It is also possible to create global variables. This is often done to exchange information between functions.

Configuration, resources and tasks

Figure 2.16 illustrates the software model of IEC 61131-3.

The highest level is the configuration. It defines the hardware composition, all processing resources as well as all in- and outputs. In such a configuration one ore more resources may be defined. These resources are able to execute IEC 61131-3 programs. Additionally they may contain one ore more tasks controlling a periodical or triggered (input) program execution.
A program may be written in any language which is defined in IEC 61131-3 and usually contains a set of functions and function-blocks, implementing the logical behaviour.

Standard PLC applications have one program running in one configuration in the standard resource. The standard offers scope for distributed and future systems.

Program organisation units (POUs)

Programs, function-blocks and functions are called program organisation units. Functions are similar to functions of higher programming languages and there are also predefined operations like time or string manipulating functions.
Function-blocks are similar to classes so they have a defined interface and internal data. Function-blocks may implement a required control operation and can be seen as a black box with in- and outputs. Many instances of a function-block are possible and also a kind

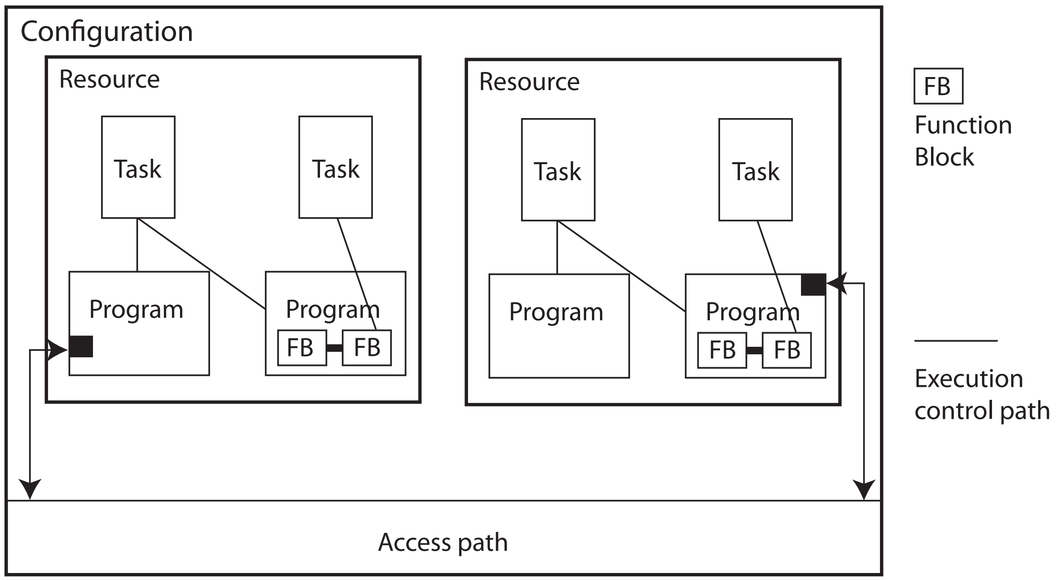


Figure 2.16: IEC 61131-3 software model [vdW99]

of derivation is feasible. Any of the defined languages may be used again.
A modern written program is almost a set of functions and function blocks.

Sequential function charts (SFCs)

The sequential function chart is related to a Petri Net or a state machine and illustrates the logical behaviour with loops and branches or parallel activities. It shows each step with a transition between the blocks. A SFC is often used to discuss the system because it is easy to read for non-technicians. Figure 2.17 shows a simple sequential function chart. Some papers declare the SFC as a programming language [Ver96].

2.2.4.2 Programming languages

Four programming languages are defined within the standard IEC 61131-3. Two of them are textual and two are graphical. Table 2.4 lists all four languages [Ver96; vdW99; Bol06].

Textual	**Graphical**
Structured Text (ST)	Ladder Diagram (LD) german: KOP *Kontaktplan*
Instruction List (IL) german: (AWL) Anweisungsliste	Functionblock Diagram (FBD) german: (FUP) Funktionsplan

Table 2.4: IEC 61131-3: Programming languages

Every programming language can be transformed to any of the other ones. For example: a ladder diagram can be converted to an instruction list without losing information

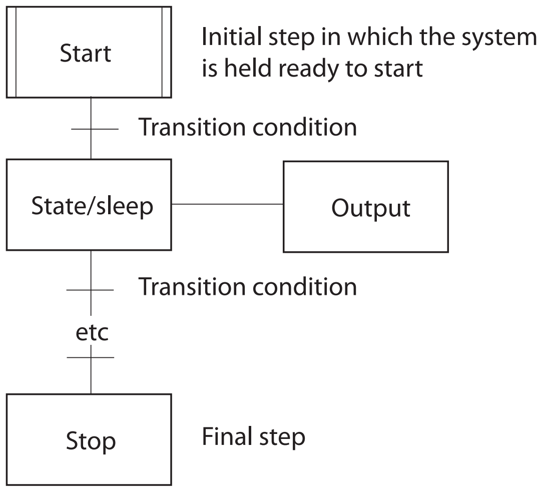


Figure 2.17: Sequential function chart [Bol06]

but some types do not offer the full functionality. Figure 2.18 shows a simple example in all four different languages.

Instruction lists are similar to assembler and were developed in Europe.

Structured text is a high level and powerful language with roots in ADA, Pascal and C. Complex or mathematical blocks can be written in this language. Structured text is not case sensitive. As in higher languages, the *IF* and *FOR* statements are available.

The function block diagram has blocks as basic programming elements which are connected. The data flow is from the inputs (left) to the outputs (right).

The ladder diagram can be seen as a relay ladder logic which is easy to read and has its roots in the United States. Normally this language is used for boolean logic. Counters and timers are not implemented in this language. The horizontal lines on both ends represent the power rails which are connected with single circuits. A ladder diagram is read from left to right and further, from top to bottom.

2.2.5 IEC 61499

This standard was published in 2005 by the International Electrotechnical Commission as an extension to IEC 61131, to manage distributed systems. The interoperability between

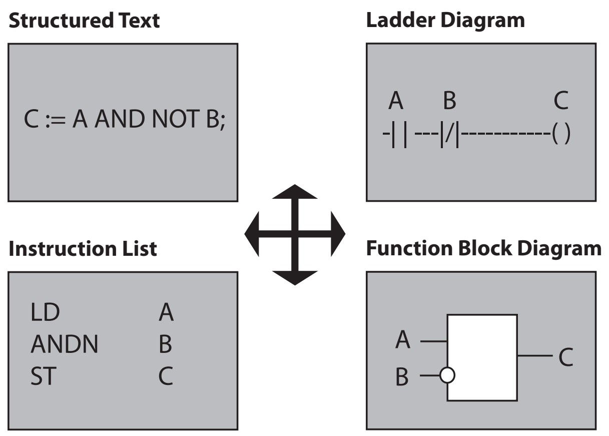


Figure 2.18: IEC 61131-3: Languages

different systems like PLCs and application specific ICs as well as covering several suppliers is a main goal of the standard. Programming languages were added to the existing ones, for example Java and Delphi. The two standards differ in the system layer, which is new in IEC 61499, the interface of function blocks and the new execution control chart [GHE08]. The new system layer allows the development of the system, with all controllers and devices in one project. The cyclic execution of programs changed to event driven processing. Therefore every function block has a port *DATA* and *EVENT* as in- and output. Figure 2.19 shows a simple average value calculator based on IEC 6131 and IEC 16499. The data ports are the same, but additionally, there are event-ports for control. The function-block is not executed in every cycle, only if the event starts the task. Further information can be found at [Hol08].

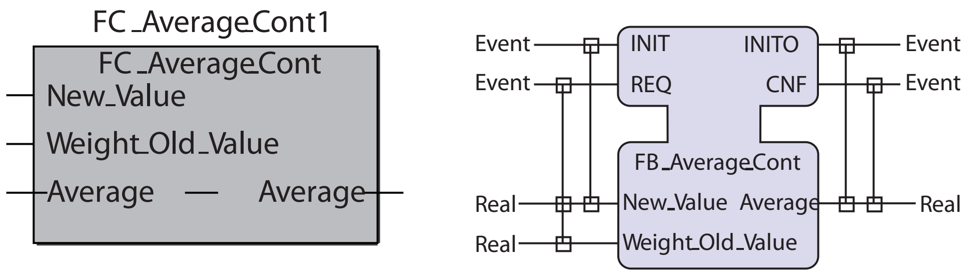


Figure 2.19: IEC 61499: Example function-block [GHE08]

The programming methods currently follows the IEC 61131 standard. This standard is dominating the PLC-market although the IEC 61499 has reached the end of its life cycle

[GHE08]. Both standards will coexist for the next 10 years. Another important point is the software of all PLC vendors. They also have to adapt their tools which will take time. Some ongoing projects show that both techniques can coexist [HGHV07].

2.2.6 Modern approaches

Conventional software development for programmable logic controllers was discussed in the previous Sections, but the productivity and the speed is moderate so much research is done in creating new schemes and code generation for PLC. This Section gives an introduction to some important new approaches. The majority are at a research stage and not all information is published especially the PLC code generation. In practice these modern approaches are rare.
The title *Software product lines for PLC* of [SMBU07] would suggest relevant information for this work, but there is no technical information given. Only financial benefits are discussed.
Some PLC vendors implement new approaches like Bernecker and Rainer (B&R) by offering Matlab® add-ons.
There are also some code generation techniques based on IEC 61499.

2.2.6.1 Object orientation

Partially, all modern programming languages are object oriented, but the IEC-61131-3 languages are not. Much research is being done in developing prototypes for PLC programming, implementing this paradigm. The description language is the unified modeling language (UML) with extensions (see Section 2.2.6.3) [HP07].
Function-blocks are like objects, but the control logic is the challenge [BF01]. Clean implementation of function-blocks leads to *meachatronic objects* including the hardware and its control software. These blocks can be implemented independently from the main program and so, object libraries may be created.

2.2.6.2 Petri nets

Petri nets allow a formal verification of an algorithm, which is almost impossible with standard PLC implementations. This is one goal of this approach. After modeling the application with Petri nets, a code generator creates the program as an instruction list or ladder diagram. Basically, there is a one to one correspondence of code and net elements, so the produced code remains readable. The Petri nets used are called *Signal Interpreted Petri Nets (SIPN)* [Fre00].
[FW06] presents a toolbox support to implement such systems. It is designed as an integrated development environment (IDE), written in C#. Figure 2.20 shows the concept of this IDE. Important parts in respect of this work are the editor(ED) which allows the creation of the SIPN and the PLC implementation generator (PIG). The PIG creates a text-file, including the program as an instruction list which is imported in the supplier depended tool. This toolbox is freely available at `http://www.eit.uni-kl.de/frey`.

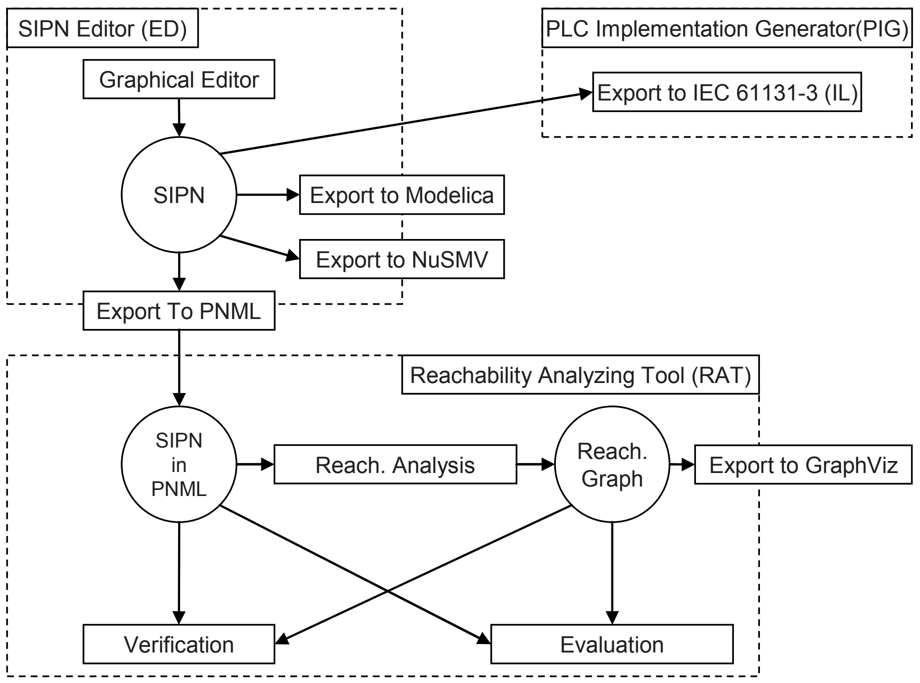


Figure 2.20: Signal Interpreted Petri Nets: toolbox concept [Fre00]

2.2.6.3 UML extension

An extension to the unified modeling language (UML) which was adapted for the special purpose control engineering is able to create code for programmable logic controllers. This allows a structured and modular development. Further, a code generator creates the PLC code as structured text [BDKB07].
[YF06] present a prototype to transform models which are defined by UML to a PLC language. *Real-Time Studio* from Artisan® was used as modeling tool. The paper also presents some problems with this approach. A further step is the adaption to UML 2.0.

2.2.6.4 Virtual reality

This approach uses a virtual environment to model the system. Every configuration is done in a graphical way in this simulated environment and every sensor and actor is also an element in this system. All single objects are connected via predefined boolean objects. Afterwards a special compiler creates the code as instruction list [BDKB07; SO96].

2.2.7 Model driven design

MEDEIA is a new collaborative EU project with the title *Model-Driven Embedded Systems Design Environment for the Industrial Automation Sector*. The project started in the year 2008 with a duration of 36 months [MED08].

The main goal is to reduce the system design time by 25%. Key elements are, for example, a formal framework for model-driven component-based development and an easy modeling method for domain experts.
The technical approach MEDEIA is based on *automation components* which are the integral part. Figure 2.21 shows the principles of this project. An automation component contains the general model of the functionality and the interface definitions.

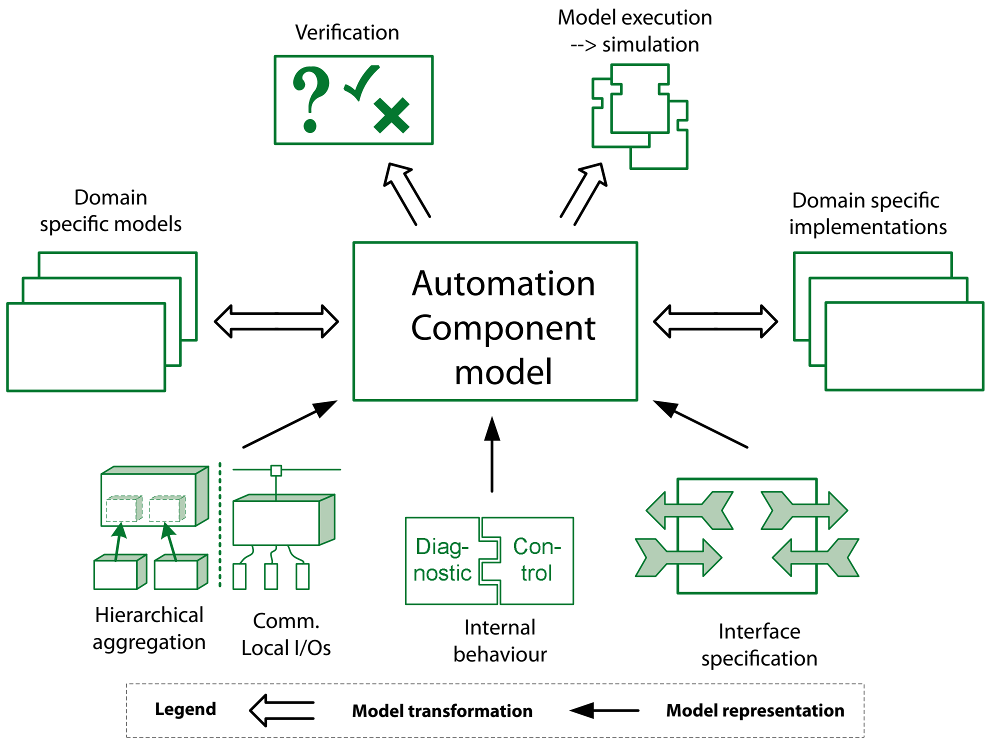


Figure 2.21: MEDEIA: Technical approach [MED08]

Chapter 3

Design and implementation

This chapter deals with the design and implementation of a domain specific architecture for programmable logic controllers in a logistics application. Firstly, all requirements which lead to an appropriate architecture are listed. Afterwards the implemented software is specified. At the end of this chapter, an evaluation is listed which includes a comparison between the new paradigm and conventional PLC programming.

3.1 Requirements

The domain of this work is a logistics system which is built of conveyors, rotary tables, cranes and a high bay racking.
Figure 3.1 shows a photo of the "real world system", which is located at the Institute of Technical Informatics. It is built with logistics parts from *Fischertechnik* and controlled by a Siemens® S7 PLC. Figure 3.2 shows some impressions of the system.

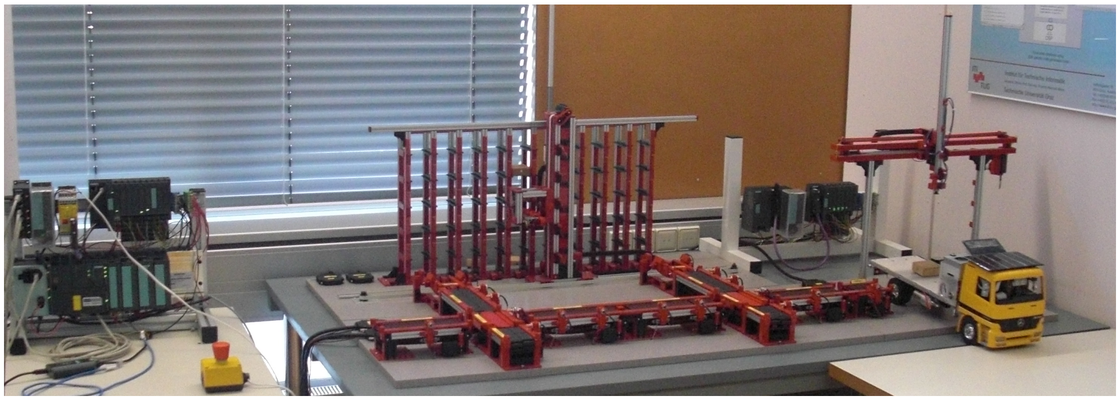


Figure 3.1: Real world logistics system

Most of the PLC basic functionality was developed in a BsC project by Wolf and Weber [GH09]. They also developed a simple fix graphical user interface with WinCC® to control the system. The resulting inflexible system led to the need for this thesis.
The requirements of the implementation can be separated into three main classes: business goals, technical goals and process goals. The analysis of all goals will lead to an appropriate

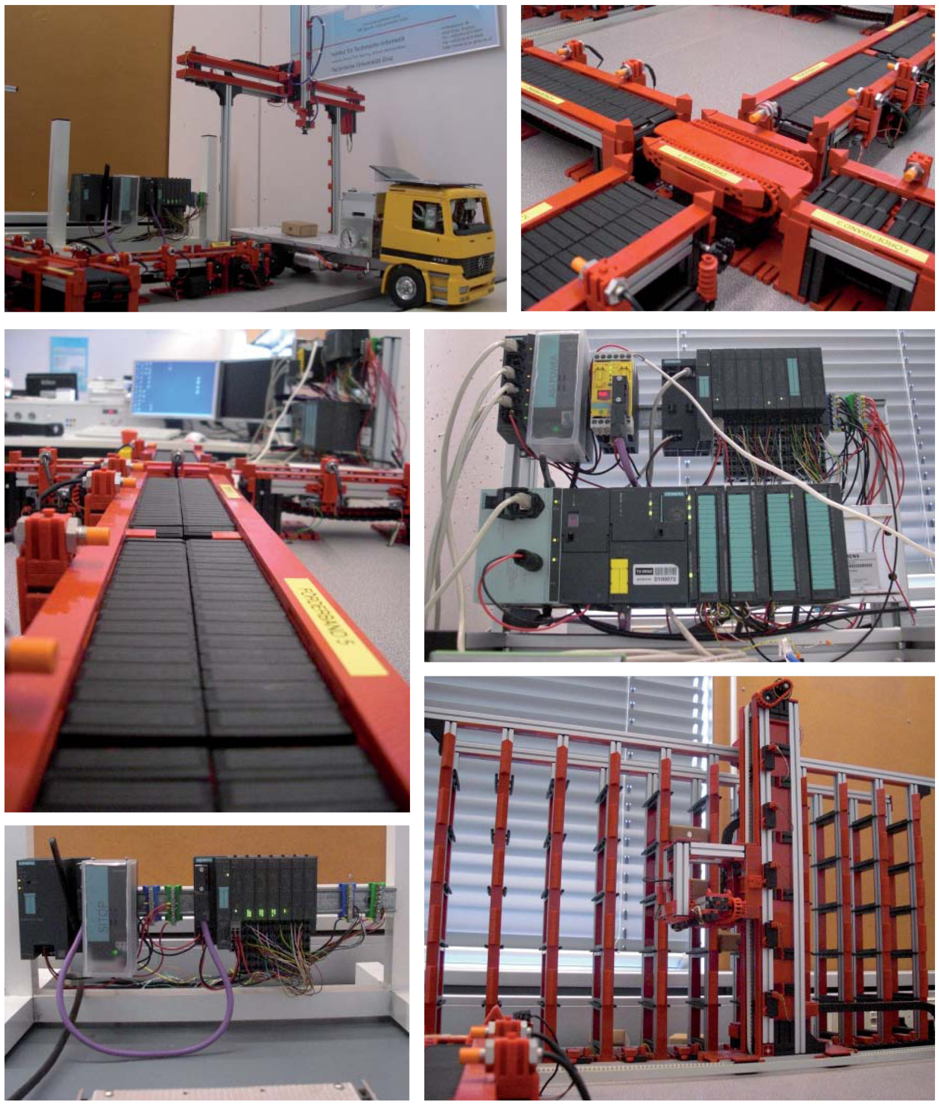


Figure 3.2: Impressions

architecture.

Business goals mainly deal with financial aspects and future development of the company domain. Reduce the development costs and speed up time to market. A wide range of individual products of high quality, so a mass customisation in the special domain and a high reuse should be achieved.

Process goals list the features which the process should provide. It should be possible to create assemblies of a logistics system in a graphical way and the set of objects should be extensible for new features (creating a reusable object pool). Single objects should be adaptable for customer needs concerning the domain. The graphical specified logistics model should be transformed to a functional PLC program with a consistent documentation. Then the code is downloaded with the SIMATIC® manager, a programming tool from Siemens®. The management controls all activities. Figure 3.3 shows the simplified target process.

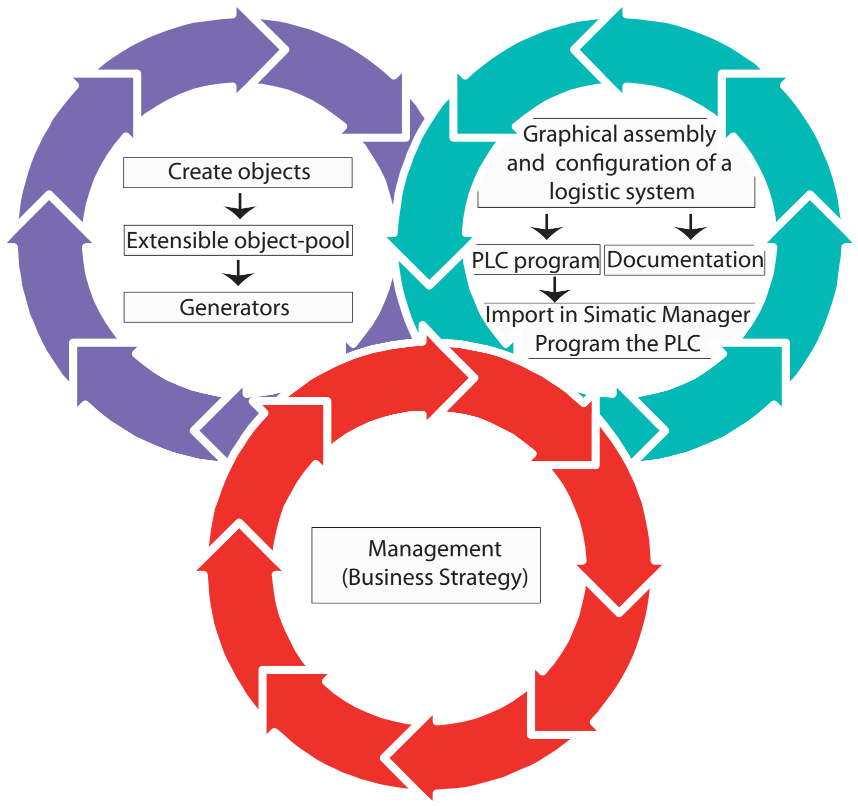

Create objects
Extensible object-pool
Generators
Graphical assembly and configuration of a logistic system
PLC program
Documentation
Import in Simatic Manager
Program the PLC
Management (Business Strategy)

Figure 3.3: Implementation SPL architecture

Technical goals contain the reuse aspect of developed function blocks and other source-files in all possible configurations, so a common interface has to be designed. Another requirement is to generate deployable code for a Siemens® PLC with the SIMATIC® Manager. This code should be readable, maintainable and extensible for developers. The target system is a SIMATIC® S7-300. The software should be network based, so the object repository is available on any computer in the local area network (LAN).

Another important aspect is the improvement in the quality in the whole process as well as in the generated code and the documentation.

3.2 Product line architecture

This software product line implementation meets all requirements of the previous Section. The challenge is to map this approach to programmable logic controllers.
With referenece to the benefit analysis of Section 2.1.8.7 MetaEdit+® is used to realise the domain specific modeling architecture.

3.2.1 Introduction

Figure 3.3 shows the simplified target architecture of the implemented domain specific SPL. Remembering the SPL development process of Section 2.1.4, the domain engineering contains the creation of domain specific artefacts. In this thesis these artefacts are all required logistics objects of the implementation and the generators. These objects are stored in a pool to be used in the application engineering process. Most of this functionality is provided by MetaEdit+®. Management, with respect to the business strategy, defines necessary and added objects and is involved in every decision. It deals with exploring the domain and future needs.

Application engineering contains the graphical assembly and configuration of the logistics system. After defining the logistics system, the PLC code and a consistent documentation is generated. Additionally generated reports, for example a hardware suggestion are discussed later. The created PLC program is imported in the Siemens® programming tool SIMATIC® Manager and then deployed on the target system.

As mentioned, most of the basic functions for the PLC of the logistics system were already implemented by a previous project. So the starting approach of the software product line is incremental (see Section 2.1.4.4). The artefacts are extracted from the existing code and defined for a useful handling in the domain. Of course, several adaptations were required to create reusable objects and a common interface as well as removing errors which are exposed.

The next step is the creation of the domain specific model. Therefore, the whole domain-knowledge is clustered and components and connections are defined. Using the MetaEdit+® GOPPRR meta-model, the models are designed. Figure 3.4 illustrates a simplified assembly.

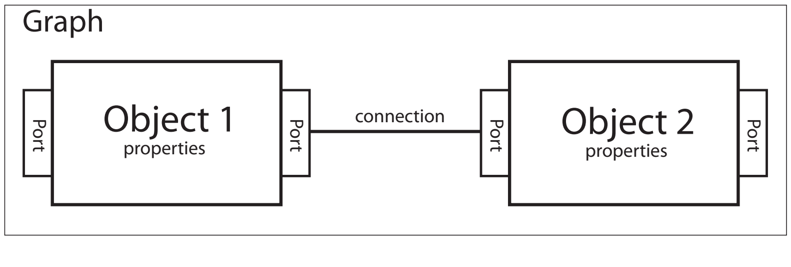


Figure 3.4: GOPPRR Model definition

Every object has ports which enable a connection. The connection to another object defines the role of the object. For example, there are the ports left and right at each conveyor and a connection of port right at object 1 to port left at object 2 defines the roles of both objects. Object 1 is the left neighbour of object two and object 2 is the right neighbour of object 1. To capture the assembly, a walk through the graph from role to role is done.

3.2.2 Domain model

As described in Section 2.1.8.6, MetaEdit+® applies the GOPPRR metamodelling language. This Section shows the implementation of the logistic model in MetaEdit+®. The logistics system is broken down into single objects with properties and constraints with respect to the platform architecture. The high bay racking, a gantry crane, conveyors, rotary table and rack servicing units are implemented. These objects can be placed and connected in a defined graph with constraints. The following Sections will deal with the modeling in the GOPPRR-order.

3.2.2.1 Graph

The implemented graph *Logistic System* contains all available objects, connections between them and roles. The rules for connecting objects are also defined here. For example: a conveyor can not be connected to the high bay racking, only rack servicing units are able to be direct neighbours. Further object and role occurrence constraints are predefined within the graph, such as the limit of right neighbours of a conveyor, which is one. The graph can be seen as an environment with available objects and the rules for creating one's own models. The graph itself has properties like name and customer.
All parts of the graph are shown in Table 3.1. Detailed information is given in the corresponding Sections.

Relationships	Roles	Objects
Connection	Left Neighbour	High bay racking
	Right Neighbour	Gantry crane
	Atledge1	Short X conveyor
	Atledge2	Short Y conveyor
		Long X conveyor
		Long Y conveyor
		Rack servicing unit
		Rotary table
		Emergency stop

Table 3.1: Parts of the model *Logistic System*

3.2.2.2 Objects and properties

Table 3.1 shows all implemented objects, which are available for the user but there are more in the model definition which are nested into other objects. For example: a PLC-connector object for a conveyor which contains all required addresses is a property of the conveyor or the superclass *conveyor* is not available for a user who is modeling the system too. All objects, their properties and a short description is given in the following Sections. Some properties are checked against rules, when they are entered. For example input and output addresses are tested with a regular expression which ensures a correct input. An example would be *4.1*: the regular expression allows a digit, followed by a dot and again a digit.

High bay racking

The high bay racking type model is illustrated in Figure 3.5. It contains a name which is numerical and checked when typed in and the ranges in X- and Y-direction (rows/-columns). Additionally the amount and the position of connections to rack servicing units are selectable. The property documentation allows a user to link an object to the high bay racking, for example, a PDF-file.

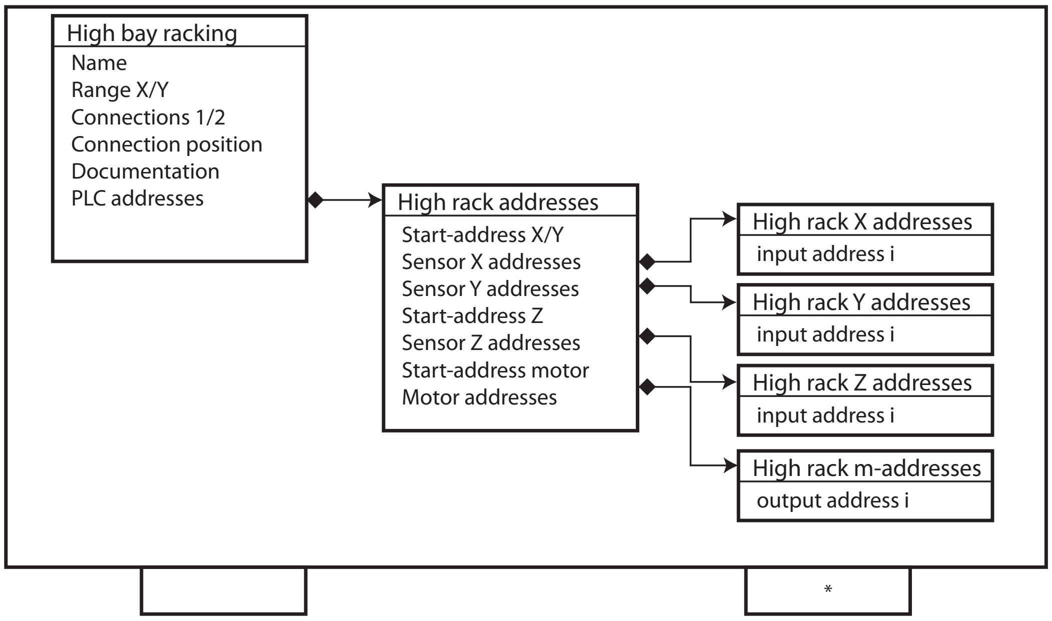


Figure 3.5: Type model: High bay racking

The property PLC addresses contains a sub-object which itself contains sub-objects for all necessary addresses. Start-addresses X/Y, Z and motors are needed to parameterise a helper function which fills all I/O addresses automatically (see Section 3.4.5). They are not necessary because the user is able to enter all values manually. The addresses are checked through regular expressions.
Currently, the high bay racking is limited to 10 columns and 5 rows, because the real world model has this maximum-size. Only smaller configurations are possible.
The high bay racking has two ports for connections. One is optional, marked with a asterisk. The only valid neighbours to a high bay racking are rack servicing units.

Rack servicing unit

Each rack servicing unit contains a name, documentation and PLC addresses which include both transportation directions (motor rotation) and the ”occupied” sensor. The model is shown in Figure 3.6.

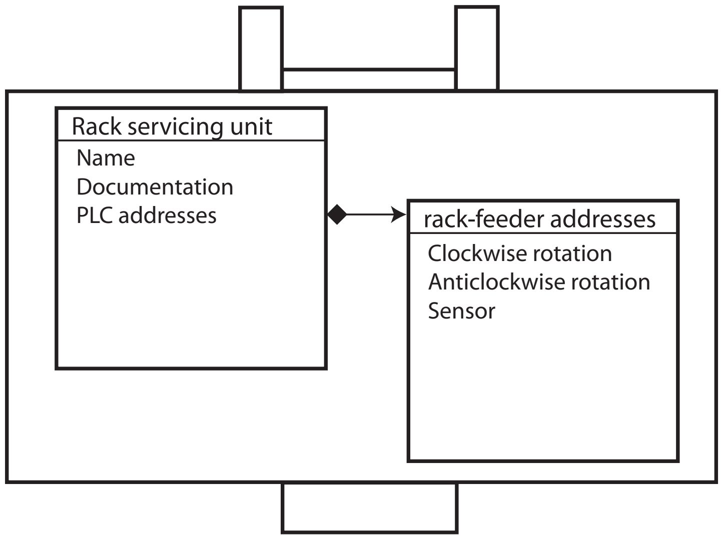


Figure 3.6: Type model: Rack servicing unit

The rack servicing unit has two connectable ports: The port with the hollow for the high bay racking which must be connected with a high bay racking or left unconnected when packets are deployed manually. The second port may be connected to any conveying unit.
Additionally a checkbox is implemented which enables the display of all addresses of an object in the layout. This feature is common to all conveying units.

Conveyors

The conveyors are implemented as four separate type models (X-Y—short-long) altough there is only one corresponding object in the PLC layer below. The reason is to support dimension checking, to provide a nice layout and adequate prompts. A further reason is a logical check. A Y-conveyor cannot be connected to a rotary table at port X or another X conveyor without a rotary table. The conveying units have a name, documentation and an address mapping which is different. Both models are shown in Figure 3.7.

A long conveyor may have one or two sensors for commission detection and is illustrated left in Figure 3.7. This is selectable and implies an additional field in the address object which may be empty, if only one sensor is used. If two sensors are used a check for a corresponding address is done. The graphical object also reflects the correct sensor count. All conveyors have two ports. If a conveyor is an endpoint of the system, one port remains unconnected.

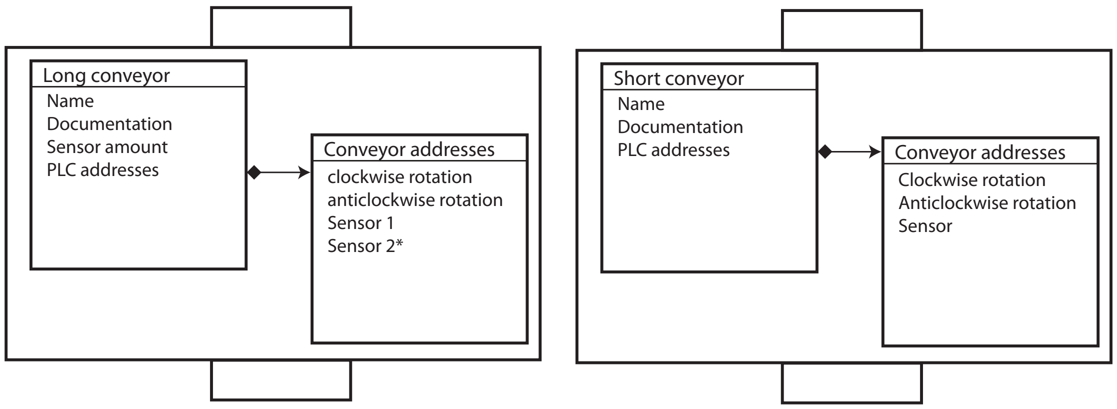


Figure 3.7: Type model: Conveyors

Rotary table

The rotary table is the only object with more than two ports and connections. It may be connected up to four times with conveyors. The model includes a name, documentation and the addresses object. This object contains all required PLC I/O's. Two outputs for the conveyor rotation, two outputs to rotate and 3 inputs for the commission sensor and position switches. Figure 3.8 shows the basic rotary table object.

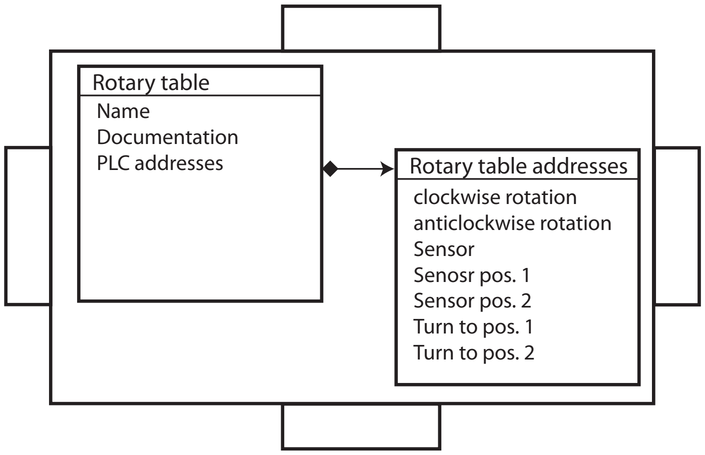


Figure 3.8: Type model: Rotary table

Gantry crane

The type model of a gantry crane is illustrated in Figure 3.9. It includes a name, documentation, start-addresses to fill the addresses for the PLC automatically (see Section 3.4.5) and the address mapping. The mapping contains the outputs to move the crane up, down, left and right. The fifth output controls the vacuum pump which is responsible for lifting objects. The inputs are for all sensors needed to recognise the position of the crane (center / left / right / up / commission detection). The gantry crane has two manual switches which allow manual movement, so these inputs are also in the address-mapping.

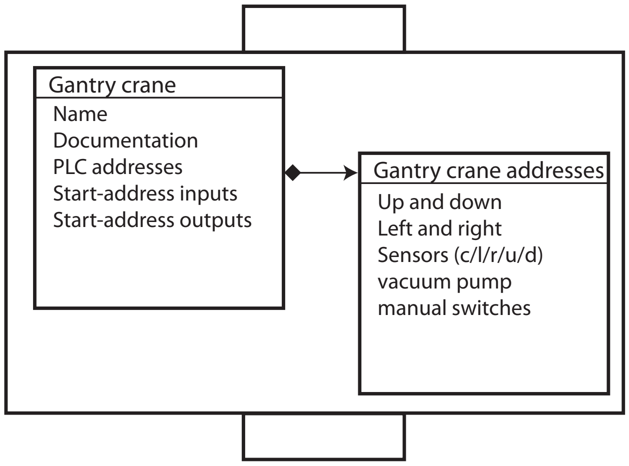


Figure 3.9: Type model: Gantry crane

Emergency stop

This object just includes a name, documentation and the PLC address of the emergency stop unit because no additional information is needed. Every graph, and so every application needs one emergency halt object.

3.2.2.3 Ports

As mentioned in the special object Sections, there are 8 different ports implemented in this work. Table 3.2 summarises all available ports which enable connections between objects.

Port	Object and description
Left	Left end of every conveying unit
Right	Right end of every conveying unit
Atledge1	Connection point for the high bay racking (ledge 1)
Atledge2	Optional connection point for the high bay racking (ledge 2)
Rotary table gate 1	Rotary table top
Rotary table gate 2	Rotary table right
Rotary table gate 3	Rotary table bottom
Rotary table gate 4	Rotary table left

Table 3.2: Available port types in the model logistics system domain

The ports allow the definition of the assembly. Additionally, error-checks can be done by analysing the ports. The arrangement around a rotary table can also be done with the help of ports.

3.2.2.4 Roles

The required roles in the implementation are a *right* and a *left neighbour* and two for at ledge. This definition is done with MetaEdit+®. If a conveyor is connected with another one using port right, the object is in the role of left neighbour. With this knowledge the whole system assembly can be interpreted. Going to the system is just going from role to role, so from the left neighbour to the right one.
Two further roles are needed to complete the system. *AtLedge1* and *2* which define the position of the rack servicing units at the high bay racking.

3.2.2.5 Relationship

The implementation includes only one relationship, the *connection*, which connects conveying units, the gantry crane and the high bay racking. This is the only needed relationship because it represents the possible transportation paths.

3.2.2.6 Model representation

The implemented MetaEdit+® model was illustrated with photos of the real-world system to reach a high degree of usability and abstraction. So domain experts are able to assemble new applications without knowledge of programming a PLC. Creating a new application is like using a CAD (Computer Aided Design) software with domain specific libraries.

Figure 3.10 shows the graphical model implementation of a rotary table , the conveyors and a rack servicing unit. The highrack and all other objects are illustrated in the same way.

The parameters are prompted with input dialogs and may be displayed, if the user desires (see conveyor 6 and 5).

3.2.2.7 Current domain model limitations and solutions

The current limitations of the system are a result of the real-world system. The high bay racking and the gantry crane are limited to one in each application. If a future system should have more than one of these objects, then a naming scheme has to be implemented to name the functions dynamically as done for the conveyor system (see Section 3.3.1.2). Additionally, a dynamic creation of some helper flags (i.e. rising edge detection for the sensors) has do be done.
The limitation of two rotary tables can be removed by increasing the length of the job word and adjusting the interface.

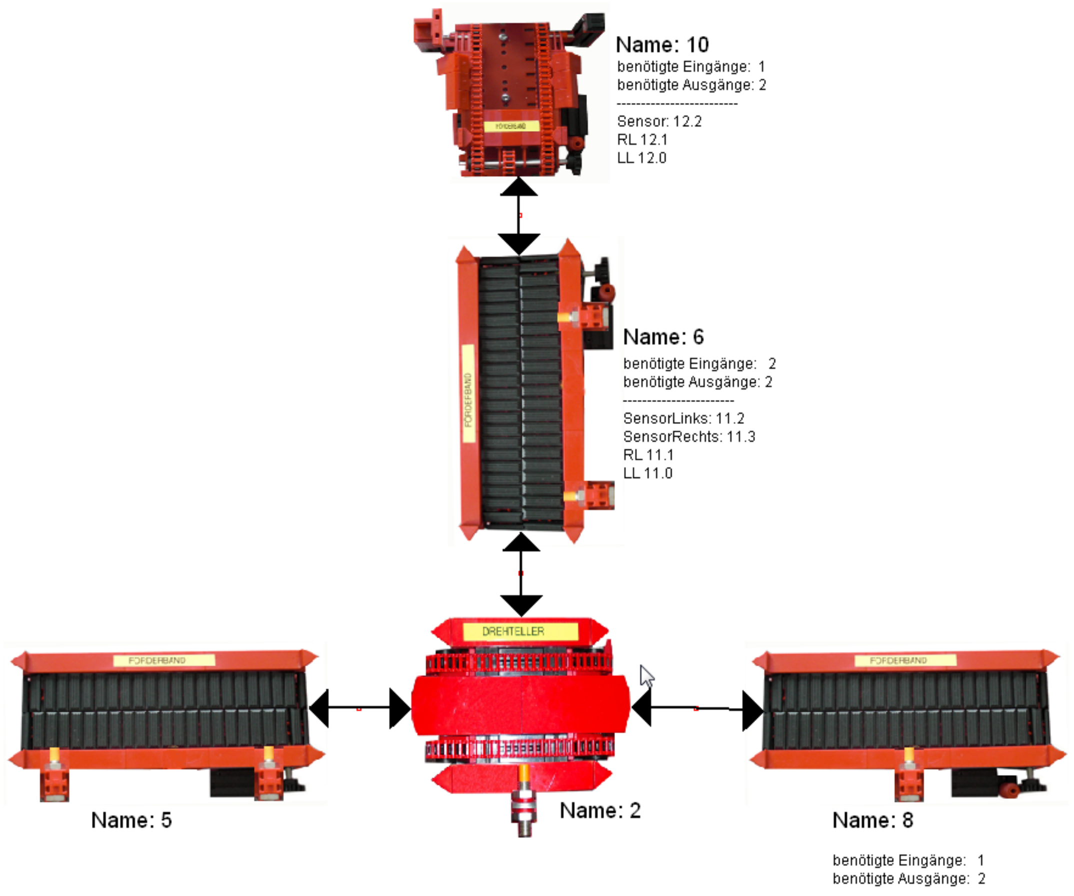


Figure 3.10: Model: Visual representation

3.2.3 Domain Specific application modeling

Figure 3.11 illustrates an overview of the whole implemented architecture. The domain model creation with all required components, rules and relationships is shown at the top of the graphic. The green frames identify the domain, whereas the black frames are application specific.
Below, the assembly of an application takes place. The components are the main parts which are instanced, parametrised and arranged as needed. The management controls both the above mentioned actions. The generators base on the graph modeling and extract needed information to generate the output. Each generator requires different input information to prepare the files. Detailed information about the generators can be found in Section 3.4. The generators are domain specific.

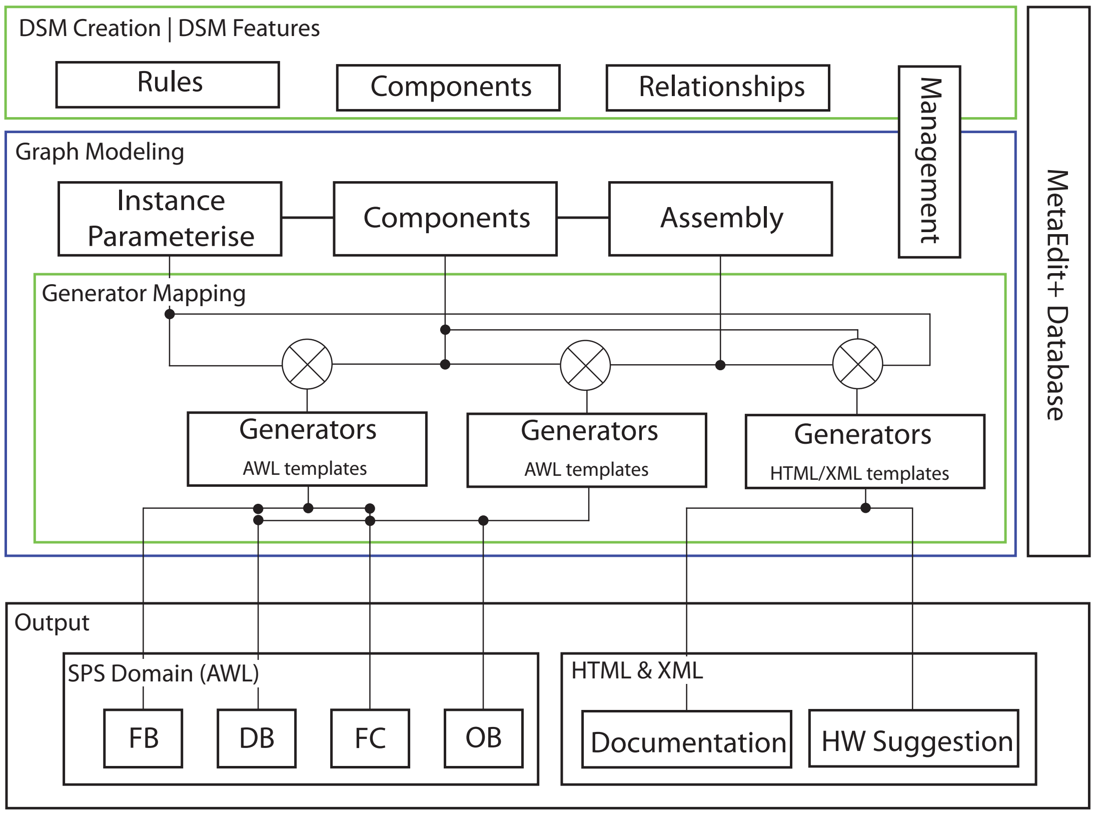


Figure 3.11: Implemented architecture

The database, respectively the repository of MetaEdit+® contains the models, rules and the generators.

The output is discussed later in more detail.

3.2.3.1 Model verification support

The error checking has three basic stages. The first stage is done continuously while creating the model, the second after the modeling process and the third during translation. The first two tests are implemented in MetaEdit+$^{®}$, the third is a syntax check in SIMATIC$^{®}$ Manager.

MetaEdit+$^{®}$ constraints include the maximal appearance of objects in the graph and valid connections are parts. It is not possible to connect objects which must not be neighbours. The roles of objects are tested too, so every conveyor can be a right neighbour once at most to avoid multiple connections. All this checking is done direct with the constraints MetaEdit+$^{®}$ provides. The input values can also be checked with regular expressions to ensure that PLC-addresses have the right format and the name is a number. As mentioned, these tests are made on the fly while modeling the system.

MetaEdit+$^{®}$ generator based error checking will be discussed in detail in Section 3.4.1.

SIMATIC$^{®}$ Manager checks the syntax during translation. This error check is more important while creating the generators and the model to see syntax errors. Incorrect variable names or missing symbollist entries as well as general syntax errors are displayed.

3.2.4 Application modeling process

Figure 3.12 shows the whole application modeling process. The MetaEdit+$^{®}$ client receives the implemented logistics metamodel by connecting to the repository in the network. For each new application a *Logistic System* graph (see 3.2.2.1) has to be created. Assembling the system is an iterative task, done by selecting and placing logistic objects in the graph. Then, the instanced objects are parameterised. During the composition integrated error checks are performed. For example, connection checks which ensure a correct binding between objects and defined values for addresses or names.
After composing the logistics system, the generators are used. Basically there are two different types. As mentioned, MERL$^{®}$ is read-only, so write access is only possible through the SOAP interface. Therefore the *Autofill* generator is implemented in JAVA to write addresses to the model. Some error-checks are also implemented in JAVA due to the rich libraries which make list-handling easier to check double addresses.

After checking the composition for correctness, the MERL generators for source code and documentation generation may be started. The hardware suggestion may be generated too.
The output of the source code generator contains AWL and SCL files as well as the symbollist. The documentation is organised as a website with HTML and CSS files. The hardware suggestion is an XML document. The target systems are the SIMATIC$^{®}$ Manager for the code, a web-browser for the documentation and an editor for the hardware suggestion.

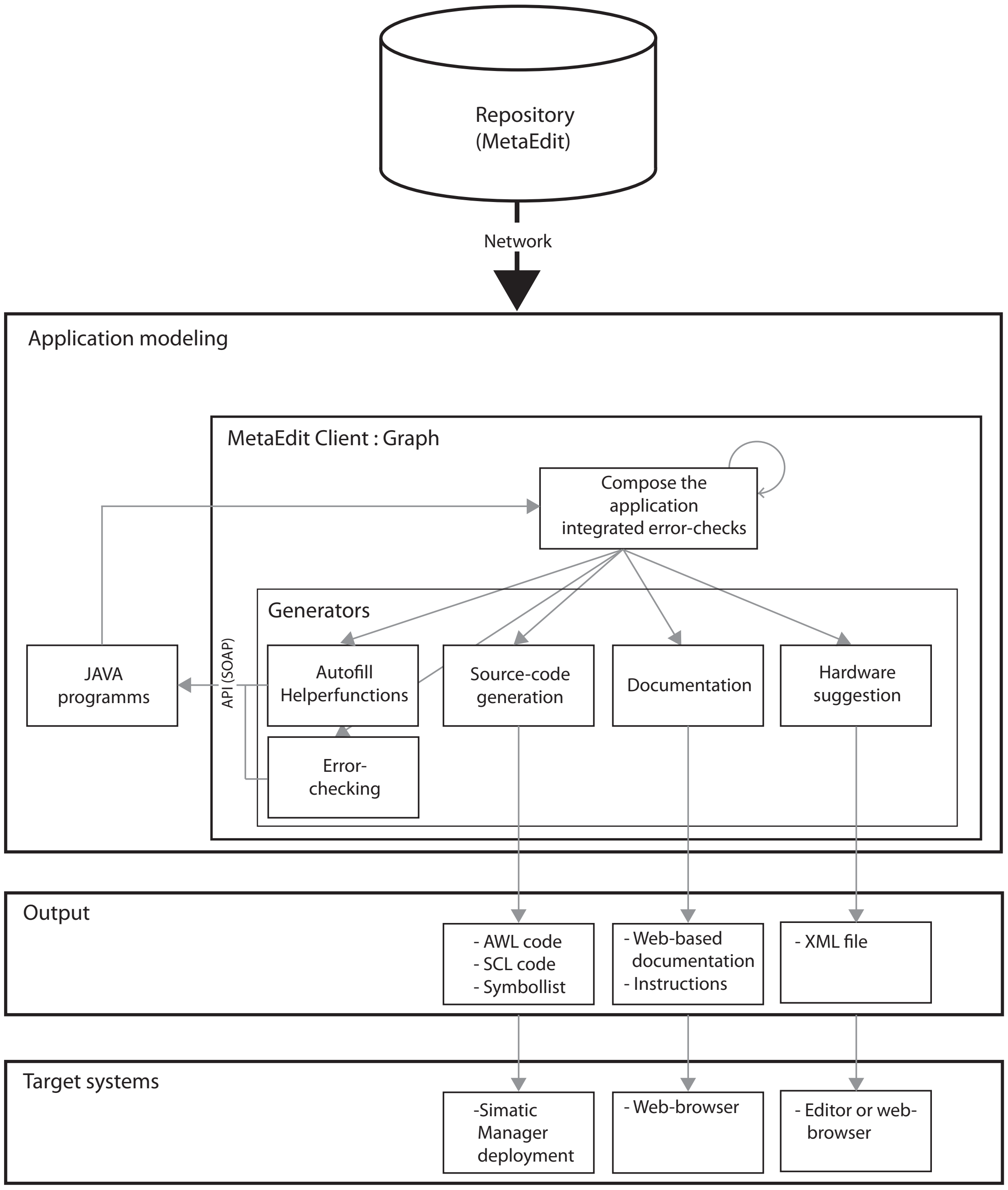


Figure 3.12: Application generation workflow

3.3 PLC target platform architecture

The target platform is a Siemens® SIMATIC® S7-300 using programming techniques of IEC 1131-3.

3.3.1 Functions and components

As already mentioned the basic functionality for a single logistics system was implemented by a BsC project [GH09]. The creation of a common, flexible and extensible software was the main task in the PLC layer in this project. Some functionality was added and other parts were removed from the basic system. This Section describes the PLC architecture of the implementation regardless of a special hardware configuration.
As depicted in 2.2.4.1, program organisation units are used for PLC programing and these blocks are the main parts in all figures which show the architecture of sub-systems. The blocks are drawn as a rectangle with the name in the top left, the internal name right aside. In the top right corner, the letters *S* or *D* indicate a static or dynamic source generation of the object. A static source generation extracts model information and reuses a given block whereas a dynamic generation extracts many information from the application model to create and parameterise one target artefact.
The body of the rectangle includes the main functionality.

3.3.1.1 Main task

The top layer of the PLC program structure is the main-task, called OB1 as is illustrated in Figure 3.13. All functions which should be provided by the system must be listed here, so the content is generated dynamically. The functions inside are dynamically generated too, depending on the modeled functionality of the system. If functions are not needed in the particular project they are not in the main task so Figure 3.13 shows an example application with a gantry crane, a high bay racking and a conveyor system.

3.3.1.2 Conveyor system

Figure 3.14 shows the composition of the conveyor function. Some objects like helper flags were omitted for simplification, only relevant parts are illustrated. Helper flags would be, for example, rising edge detection on a sensor input. The two static objects are the generic function blocks *conveyor* and *rotary table*. These static function-blocks are configurable and therefore applicable for each conveyor and rotary table of a system. The rotary table includes a second static function block: a FIFO buffer for queuing jobs.

The conveyor system function instantiates as many function blocks as needed in the modeled application which implies a dynamic set of data blocks for each conveyor. The naming scheme here is (100 + the name of the conveyor) per definition, a number. For example conveyor 3 has 103.AWL as data block file. Currently there is a limitation of 49 conveyors per system which could be extended easily because the only limitation is the chosen naming convention (the SIMATIC® Manager requires files which must have numbers as names).

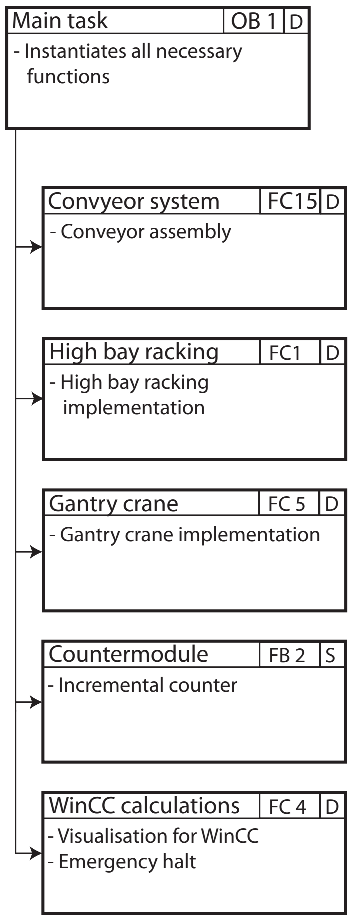


Figure 3.13: Simatic PLC program architecture: Main task (OB1)

Rotary tables are also instantiated on demand. The current maximum is two, because the job-byte is limited, but it is possible to extend this limitation. Because of the physical limitation of two objects in our lab, this was not in the scope of the work. The naming convention sets the data block label of each rotary table at 150 added to its name.
The conveyor system is created dynamically with the the information the model provides, which includes the sensor count and the neighbours. These are important, because the job has to be transfered between adjacent conveying units. The job is a double word and contains the destination information of the commission. It ”travels” with the commission along the system. Figure 3.15 shows the configuration of the job double word from Wolf and Weber [GH09]. The first 20bit are for the high bay racking and the last 12bit are for the rotary tables. The 4 info bits for the high bay racking indicate if the packet should be swapped to a rack servicing unit or switch the storage inside the rack. The rotary table bits define where to pick up and deliver the commission.

Figure 3.16 shows the usage of the rotary table bits. The job must be preset to define the path through the conveying system.

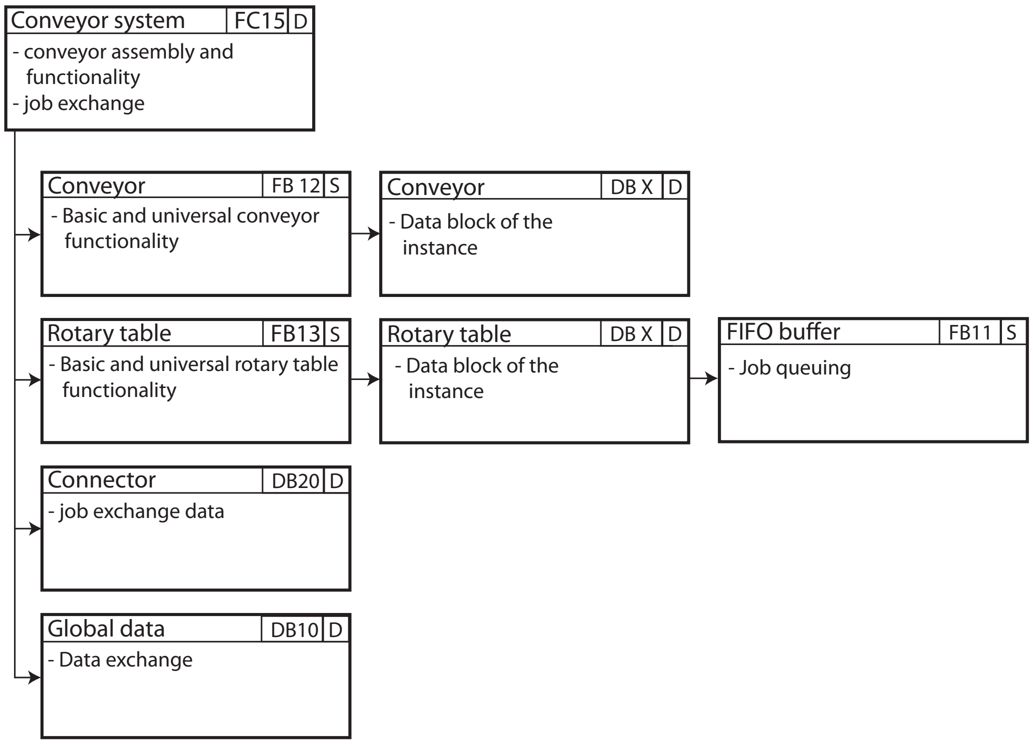

Conveyor system | FC15 | D
- conveyor assembly and functionality
- job exchange
Conveyor | FB 12 | S
- Basic and universal conveyor functionality
Conveyor | DB X | D
- Data block of the instance
Rotary table | FB13 | S
- Basic and universal rotary table functionality
Rotary table | DB X | D
- Data block of the instance
FIFO buffer | FB11 | S
- Job queuing
Connector | DB20 | D
- job exchange data
Global data | DB10 | D
- Data exchange

Figure 3.14: Simatic PLC program architecture: Simplified conveyor system

32 bit instruction word						
High bay racking			**Rotary table 1**		**Rotary table 2**	
delivery position	pick up position	info	source	target	source	target

Figure 3.15: Instruction word format

To create assemblies freely, a common and uniform interface between the conveying units is needed. This interface is discussed in Section 3.3.2.

The rotary table includes a FIFO buffer to store parallel jobs. If a conveyor has a rotary table as neighbour, the position is stored in the data block, so the rotary table knows the pick-up-position. The position is derived from the graphical arrangement in the model.
The rack servicing unit is handled like a short conveyor, so its function-block is also the standard conveyor function-block. The internal label of a rack servicing unit is 200 added to its name.

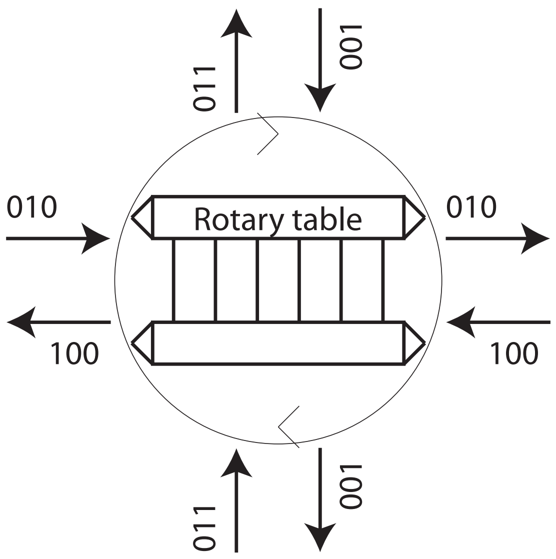


Figure 3.16: Rotary table direction codes

3.3.1.3 High bay racking

As described in Section 3.2.2.2, the amount of rows and columns is adjustable as well as the location and the number (1/2) of delivery positions. The function *High Bay Racking* (see Figure 3.13) implements the entire high bay racking functionality. Figure 3.17 illustrates the main parts.

The *controller* handles the operation sequence and is generated dynamically because of the dump position variability. The function-block *initialisation* prepares for operation. The variability is the number of columns and rows of the actual application. This applies to *Go to commission* which includes the calculation of the distance between source and target, a static function block. *Load* and *unload* a commission are responsible for manoeuvring the commission between the conveyor system and the high bay racking, again depending on the number of dump possibilities. The job buffer is a static function-block which stores up to ten jobs. The connector is needed again for job exchange between the high bay racking and the conveyor system. This exchange is only possible if the rack servicing unit is ready and the crane is in a valid deposit position. If the conveyor is not ready, the crane waits till a deposition is possible.

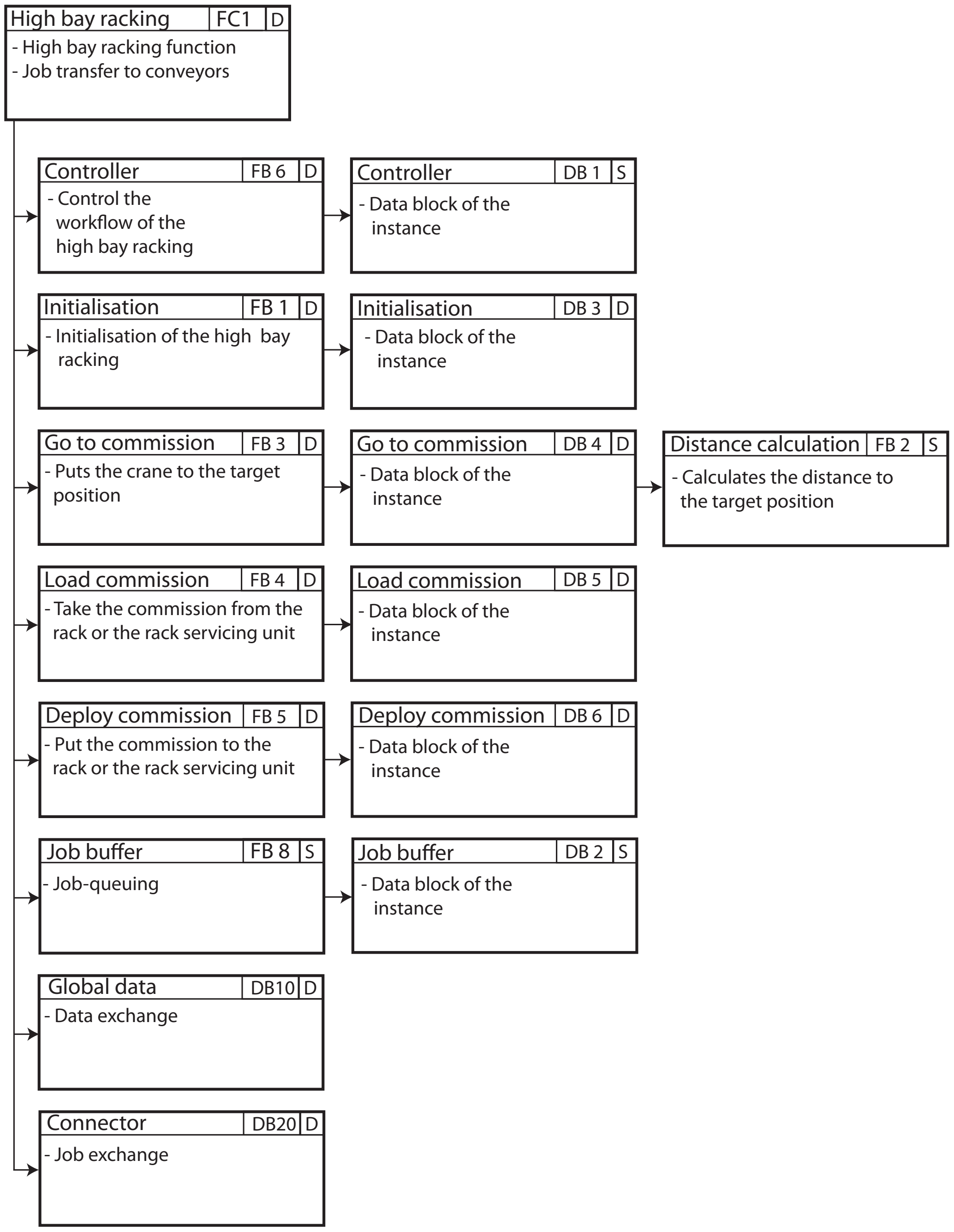

High bay racking | FC1 | D
- High bay racking function
- Job transfer to conveyors
Controller | FB 6 | D
- Control the workflow of the high bay racking
Controller | DB 1 | S
- Data block of the instance
Initialisation | FB 1 | D
- Initialisation of the high bay racking
Initialisation | DB 3 | D
- Data block of the instance
Go to commission | FB 3 | D
- Puts the crane to the target position
Go to commission | DB 4 | D
- Data block of the instance
Distance calculation | FB 2 | S
- Calculates the distance to the target position
Load commission | FB 4 | D
- Take the commission from the rack or the rack servicing unit
Load commission | DB 5 | D
- Data block of the instance
Deploy commission | FB 5 | D
- Put the commission to the rack or the rack servicing unit
Deploy commission | DB 6 | D
- Data block of the instance
Job buffer | FB 8 | S
- Job-queuing
Job buffer | DB 2 | S
- Data block of the instance
Global data | DB10 | D
- Data exchange
Connector | DB20 | D
- Job exchange

Figure 3.17: Simatic PLC program architecture: Simplified high bay racking

3.3.1.4 Gantry crane

The gantry crane function provides the functionality to pick and put commissions. Its basic components are shown in Figure 3.18. This function has only one variability which is the width of a commission. All other functionalities are static. Initialisation is required for the use of the crane, to go to a defined initial position. *Load* and *unload* are implemented as state machines and are responsible to take and deposit commissions from a mobile unit to the conveyor system. The gantry crane requires an incremental counter to go to required positions. The counter is a static predefined Siemens® library module.

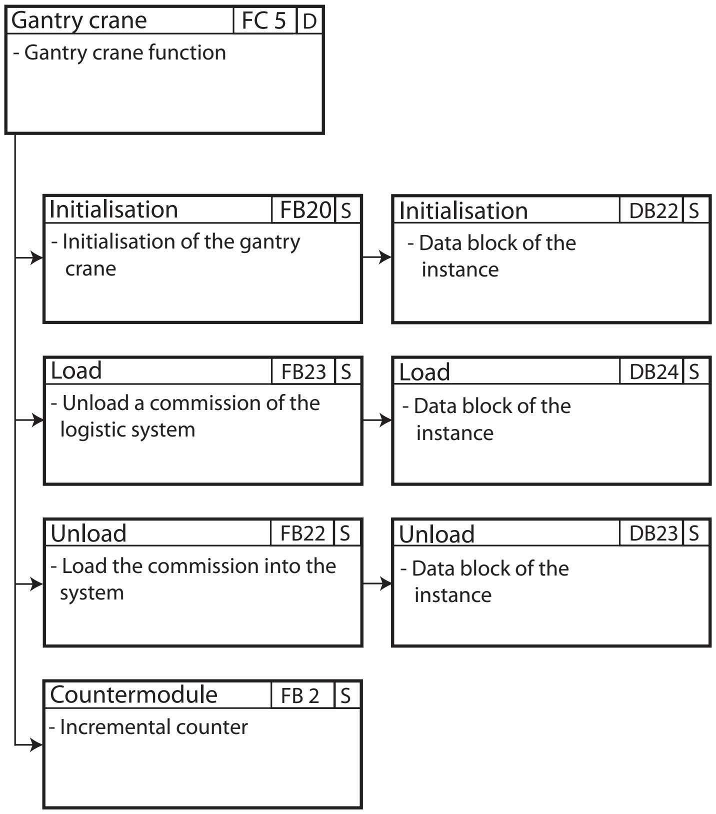

Gantry crane FC 5 D
- Gantry crane function
Initialisation FB20 S
- Initialisation of the gantry crane
Initialisation DB22 S
- Data block of the instance
Load FB23 S
- Unload a commission of the logistic system
Load DB24 S
- Data block of the instance
Unload FB22 S
- Load the commission into the system
Unload DB23 S
- Data block of the instance
Countermodule FB 2 S
- Incremental counter

Figure 3.18: Simatic PLC program architecture: Simplified gantry crane

3.3.2 Transportation interface and queuing

The basic interface principle is similar to a handshake protocol. Every involved object has a ready-out to signalise the readiness. In addition there are request inputs, either for a left- or right running direction. Communication is done via variables which are stored in a global data-structure (connector). This is a common way for data exchange in standard IEC 1131. The communication pipes are generated dynamically which map adjacent job in and outs. All communication variables needed for each conveying unit are listed in Table 3.3.

Variable
ready-out
request for right running direction
request for left running direction
packet count down
job-in-left
job-in-right
job-out-left
job-out-right

Table 3.3: Transportation interface variables

Additionally a counter is necessary for the interface the value of which is one if there is a commission processed and is reset from the following neighbour. The rotary table follows this interface by doubling all variables for both directions and a FIFO buffer to store queued jobs. In addition there are four ready outs for each neighbour.

An example data-exchange between two conveyors is:
Generate a request for a running direction. If the neighbour is ready, the conveyor starts, sets his ready to false and increments the counter. Additionally, the job is stored in the variable job-out. Depending on the amount of commission detection sensors the neighbour starts and waits to detect the commission with the first sensor. If the packet is there, the counter from the first conveyor is reset and the job will be copied in.
If the neighbour is not ready due to a holdup the conveyor stops and waits till readiness. A rotary table will copy the request into its FIFO so no request will be lost.

3.3.3 Assembly generation

This Section deals with the assembly of the platform functions and components. It helps to understand the work, done by the generators.
Depending on the model constellation, a defined functionality is needed. This logic is covered by the function-blocks and therefore instances (datablocks) have to be created, connected and parametrised. The parameters result due to the model definition.
The function-block itself has several dependencies which have to be considered, for example, a FIFO buffer of the rotary table or the counter module for the gantry crane. These additional function-blocks have to be added too. Several logic implementations are created using the format SCL. These files have to be linked to the corresponding functions.

As shown in Section 3.3.1 different functionality is separated in functions which also have to be created, if these components are in the assembly. This procedure has to be done up to the OB1, the master function. The illustrations of the previous Section show most of the dependencies between the blocks.

The interface for conveying units has to be built if required. The exchange data block with all pipes is needed too.

3.4 Generators

Almost all generators are implemented in MERL®, the reporting language which is provided by MetaEdit+®. These generators are only able to read the system specification and produce output. Whenever write access is needed, a MERL® generator starts an external Java program which communicates via SOAP with MetaEdit+®. The generators are an integral part of this work. They contain the domain specific knowledge to build all needed outputs. Figure 3.11 illustrates the information extraction of the different sources of the model. The output format is tailored to the target and no intermediate format is needed.

3.4.1 Error checking

The generator based error checking is implemented in two ways: directly in MERL® and using of the Java API. These checks include the size of the high bay racking which is limited and the connections of the rack servicing unit which must fit to the dimensions. Additionally, the existence of all important values is tested. This could be done while modeling with constraints, but it is not practical if some values are entered later. Objects with no neighbours are listed as a warning. If there is no emergency halt, an error is shown because it is mandatory.
A Java program is used to find double used addresses and names in the system. It is easier to implement such a functionality in Java than in MERL®. MetaEdit+® would provide a uniqueness constraint, but this matches all models of the type *Logistic system* which is not acceptable, because it should be possible to model many different applications.

3.4.2 PLC code generator

The PLC code generator is the most complex one and creates all necessary files for the PLC, which can then be translated and directly deployed. This Section gives a brief overview of the generator tasks. Figure 3.19 shows an example procedure of code generation. This diagram shows the simplified creation of artefacts for a conveyor.

If there are conveyors in the model definition, the function conveyor has to be created which contains all conveying units and this function has to be included in OB1 to execute it. Afterwards the required function-block has to be created. Now the model is scanned to find all conveyors and extract information. All conveyor-addresses are written in the *symbollist* file. Essential helper flags for all conveyors are created (i.e. rising edge detection). Afterwards the interface is generated, which means the creation of the corresponding entries in the connector data block. Finally, an instance of the function-block conveyor is created and parametrised with the model-information. This process is illustrated on the right of the diagram.
The conveyor is connected with global addresses like the *emergency stop* and *reset*. The neighbours define further connections. If the neighbour is a conveyor or a rack servicing unit, the interface definition is written and the corresponding sensors are wired. If there is a rotary table alongside once, again the interface definition is written and the position (left/right/top/bottom, see 3.16) is defined. If there a conveyor is of the type X and the left neighbour of the rotary table, then it is connected at the left side. The rotary table contains a FIFO buffer, so the ready-IN of the FIFO must be connected to the conveyor.

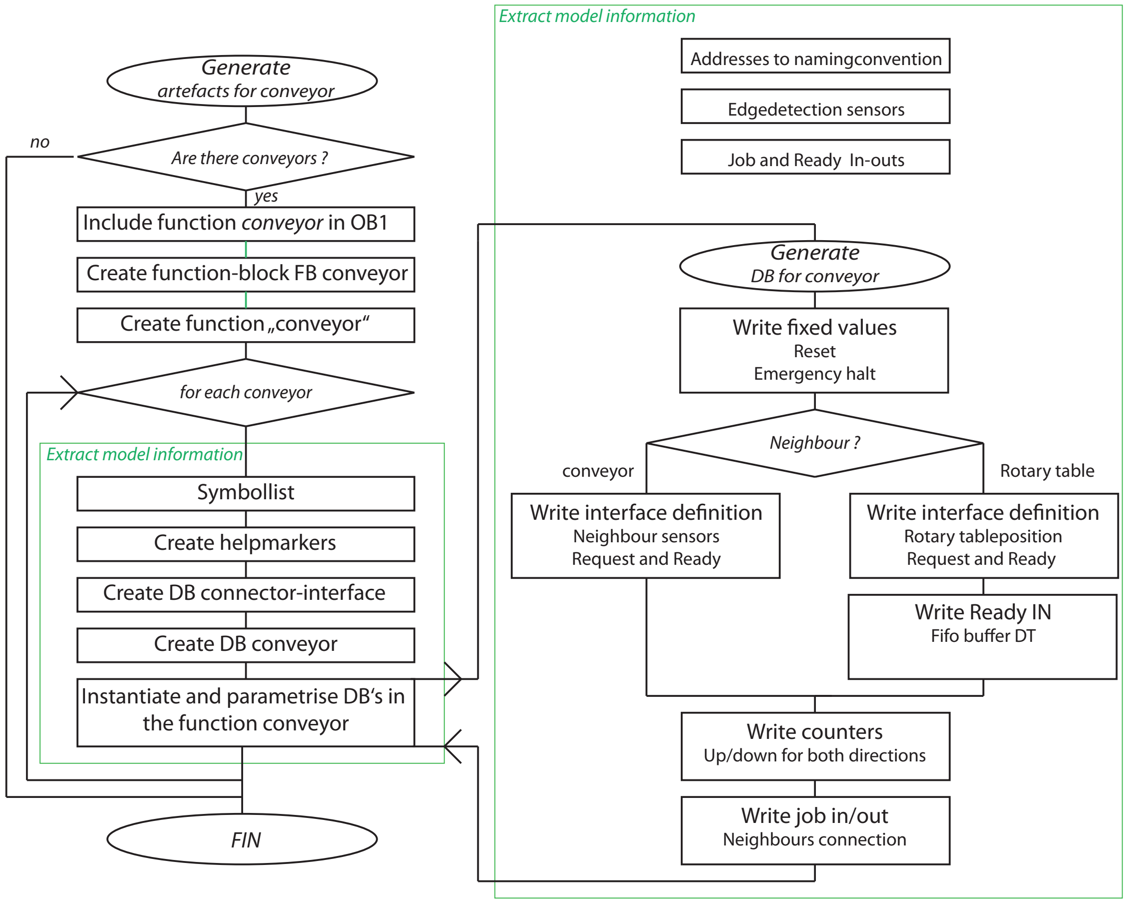


Figure 3.19: Simplified PLC code generator functionality (conveyor example)

The final step is to initialise the counters which must be unique in the whole system and to connect the corresponding job in and outs of the neighbours.

This example will produce several to each other consistent files as output: The symbollist, conveyor FB, X conveyor DB, connector DB or entries in the DB, helper flags DB or entries in the DB, conveyor function, OB1.

Seen more abstract, as mentioned, the knowledge is stored in the generator and the required AWL or SCL files are generated.

3.4.3 PLC hardware parts list generator

This generator creates a hardware suggestion by a given logistics system. To create this recommendation, several hardware specific assumptions were necessary, which are listed below.

- Possible in- and output blocks of the PLC hardware have four or eight ports.

- Every conveyor and rotary table has its own Profi Net switch with in- and outputs. This is due to the distances in real world applications so every switch is connected through a bus system with the controller. One switch also implies, that no in- and outputs of different components are combined.
- The high bay racking uses a WLAN link for the in- and outputs.

The generator extracts the model information of each single object, with all dependencies, to create the output file. The format is XML, because this is a common file format for interacting with future software implementations. Listing 3.1 shows a sample hardware suggestion and includes a high bay racking and a gantry crane, so the extra objects *IWLANLINK* and *MOBY-Counter module* are needed. Additionally all in- and outputs of each object with a corresponding recommendation are shown.

Listing 3.1: Hardware parts list suggestion (XML excerpt)

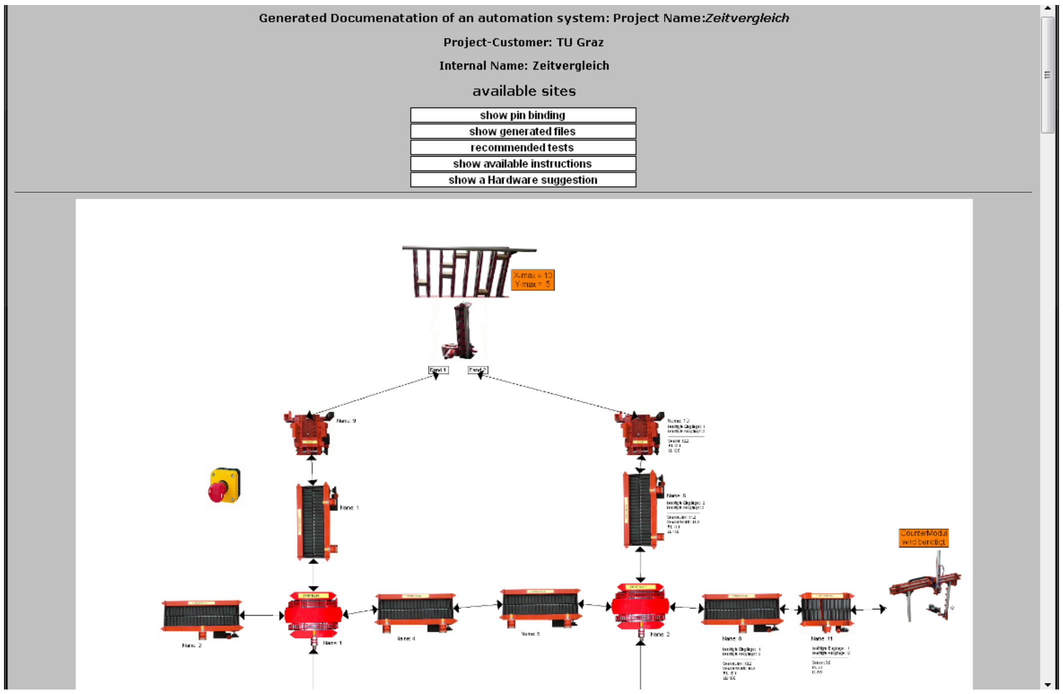
```
<?xml version="1.0" encoding="UTF-8" ?>
<HardwareSuggestion>
  <ProjectName>Demo</ProjectName>
  <Customer>Haselsberger</Customer>
  <ExtraObjects> IWLANLINK [Hochregal]</ExtraObjects>
  <ExtraObjects> MOBY-Countermodule [Portalkran]</ExtraObjects>
  <rotary table>
   <Name>1</Name>
   <NeededDIN>3</NeededDIN>
   <NeededDOUT>4</NeededDOUT>
   <Suggestion>
    <Switch>1</Switch>
    <DI4>1</DI4>
    <Startaddress>1</Startaddress>
    <DO4>1</DO4>
    <Startaddress>1</Startaddress>
   </Suggestion>
  </rotary table>
   ...
   ...
```

3.4.4 Application documentation generator

The documentation is dynamically created and organised as a website. The documentation MERL® script generates *.HTML files with embedded Javascript functions and a corresponding stylesheet. The advantage of a web site is the interactive behaviour. It is possible to link corresponding objects in contrast to a text based documentation.
MetaEdit+® allows to export a click able application model image with Javascript commands. This image is shown on the main page of the documentation. Clicking on an object shows detailed information.

Generally spoken the documentation is separated into six areas. (Screenshots are shown in the Appendix A)

Home. The start page allows direct access to all sub pages and shows the schematic of the system. A table with all objects used is available as well as a detailed listing of them. The single objects also include documentation which was linked into the objects, for example, a *.pdf file and the corresponding link to the pin binding page. Figure 3.20 shows the index page of the documentation (A.6 illustrates the information below the clickable image).


Generated Documenatation of an automation system: Project Name:*Zeitvergleich*

Project-Customer: TU Graz

Internal Name: Zeitvergleich

available sites

Figure 3.20: Documentation: Main site

Pin binding page shows all addresses and their usage. It is thought of as a wiring diagram and may be printed (see Figure A.7).

List generated files opens a folder, where all files are stored (see Figure A.8).

Recommended tests are dynamically generated and include standard test instructions for the high bay racking and the conveyors. The implementation suggests so called *In-Rack-Tests* to check the ranging as well as all the possibilities for roll in and out. If a gantry crane is used a pick and place test is suggested. These tests have to be done manually (see Figure A.9).

Available instructions. The protocol implies a specific command for a packet on the way through the system. Each new system has new commands which are calculated dynamically and presented on this site. Additionally, a Javascript calculator helps to generate commands which have to be in a hexadecimal format (see Figure A.10).

Show the hardware suggestion shows the XML based file with a suggestion for the possible hardware of the system. The DDD can be found at A.6, a screenshot at A.11.

3.4.5 Address filling assistant

Two generators are implemented for address filling. One for the high bay racking and one for the gantry crane. Both generators use extern programs which are implemented in Java, because MERL$^{®}$ scripts treat the model as read only. The start is done with a BAT file and response is given through a DOS window. These Java programs use the API (SOAP) to get write access. The high bay racking and the gantry crane have option fields *start-address* for in- and outputs. Starting from these addresses the external generator fills in all addresses automatically. Previously manually declared values are not overwritten.

3.4.6 Online help generator

The *Help* generator opens a window with a short description of each generator and the abbreviations.
CleanCodeDir deletes all previously generated files in the code directory. This helps to find actual code files.

3.5 Application development environment

This Section discusses the software packages used and gives a short introduction to the created user interface. Downloading the program to the Siemens® PLC is also covered.

3.5.1 Used hardware and software

As mentioned in several Sections, MetaEdit+® (version 4.5 Dev 92) is used to model the logistics system and to implement the generators. Some of the generators are implemented in Java by using Eclipse (3.3.0 SDK). The Java runtime environment is version 1.6.0-07-b06.

Additionally, Siemens® Software is needed to implement the platform architecture. For this, the SIMATIC® Manager version 5.4 SP3 is used. A simple GUI with WinCC® flexible Version 2007 is also created.

3.5.2 User interface

This Section describes the most important parts of the basic GUI which is shown in Figure 3.21.

The top left frame shows all available objects for the domain specific application. By selecting the required object, it may be placed in the drawing area. To the right of these objects is the connection for linking objects in the area.
Below the available models there is a list of all instanced objects with their names. At bottom left are all the parameters of a selected object.
The tool bar in the middle at the top shows all generators. The names are short forms of the generators discussed in Section 3.4. Sequential screenshots of this GUI in use are shown at A.7. Build in functions like zoom or using the grid are available too.

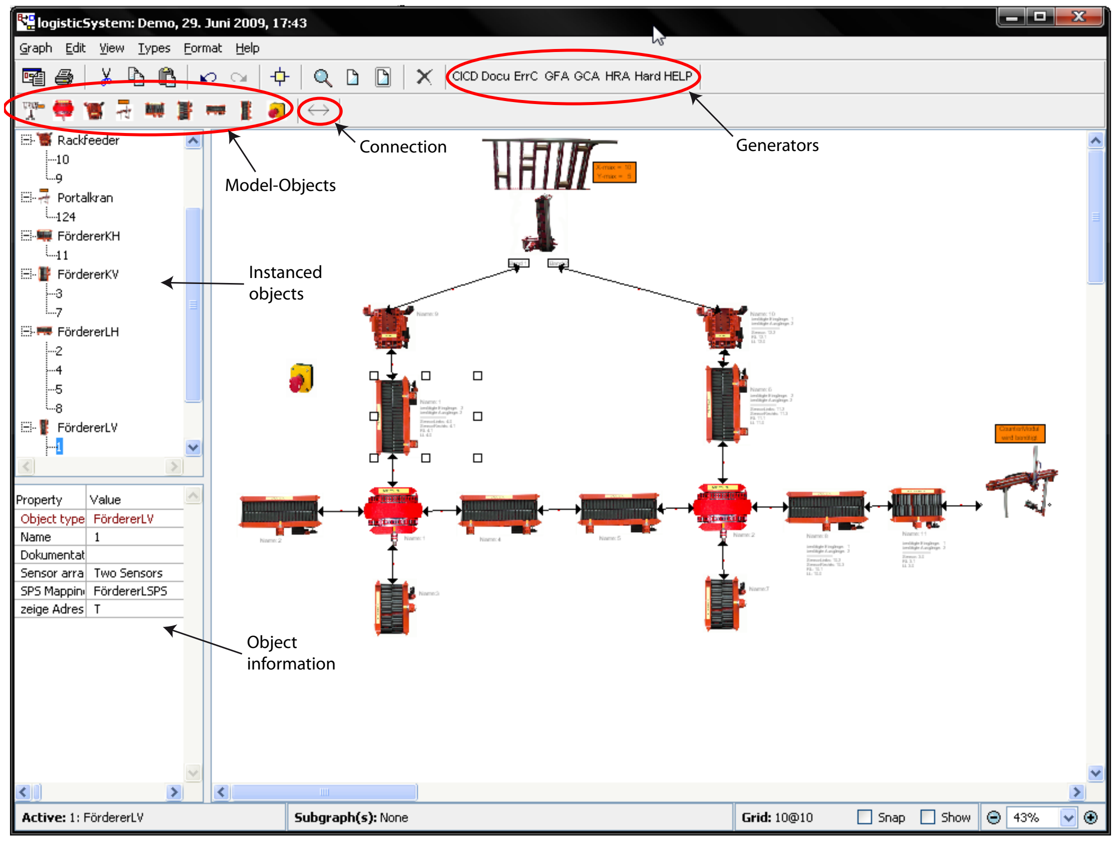
3.5.3 Migrating the application to the PLC

If the modeling of a new logistics system is done and all files are generated, the PLC programming may start. To copy programs to a PLC, the SIMATIC® Manager from Siemens® is used.

Hardware configuration

First of all the hardware configuration has to be defined with the integrated hardware configuration tool. If the hardware is similar to a previous project, it is possible to copy existing configurations for new applications. These settings specify all PLC hardware modules like counter modules, basic in- and outputs, communication between Siemens® modules in the system and their addresses. The generated XML file hardware suggestion helps to specify and address new configurations.

If this configuration is done, the hardware is ready to execute programs.



Property	Value
Object type	FördererLV
Name	1
Dokumentat	
Sensor arra	Two Sensors
SPS Mappin	FördererLSPS
zeige Adres	T

Figure 3.21: Application modeling user-interface

Application Software

When creating programs for a PLC there are two ways to use the inputs. The simpler way is to use the direct address in the source code. The more elegant method is a *symbollist*, where the mapping between symbolic names and addresses is done. To generate code with symbols the symbollist is the first required input. This list is generated by the implemented software and may be imported directly to the SIMATIC® manager.
Now the symbolic names are mapped to the hardware and the system is ready to be used with these names.
The next step is importing all generated files from the code directory. There is a naming convention, so all files have a number as prefix. This number is important because it specifies the sequence of translation which must be in the right order. First, all function-blocks, SCL functions and global data blocks with the prefix *1* are needed, afterwards the data blocks (*2*) of all function-block instances. The next required input are the functions (*3*) and the last source is OB1, the main task with prefix *4*. The sequence should be put in the correct order and now the translation can be started by pressing CTRL+B in the Simatic® Manager. This process will take about a minute depending on the size of the project. After the translation all compiled objects are available in the folder *blocks* as FUP sources.

Now all files can be downloaded to the PLC and the system is ready to work.

3.5.3.1 Files

All functions, function-blocks and data blocks are generated as separate files. Most of the files are *.AWL which is the implementation of instruction list (see Section 2.2.4.2) from Siemens®. Buffers and state machines are implemented in *structured control language* (*.SCL), a structured text approach. The files are generated dynamically, so the number of conveyors in the model implies the number of data block files. The advantage of using AWL files is the visualisation in the SIMATIC® development environment. After translating them, the view can be switched to FUP which is often favoured by programmers. Thus the generated source is easily read- and maintain-able.

3.6 Evaluation

This Section tests the implementation against the requirements of the first Section in this chapter (see 3.1).

3.6.1 Business evaluation

A business goal was to reduce the time to market which is related to the development speed, so a comparison between the conventional PLC programming and the use of the implemented domain specific generation is done to assess the success of this business goal. Therefore a test-setup was modeled in both possible ways. This setup is illustrated in Figure 3.22 .

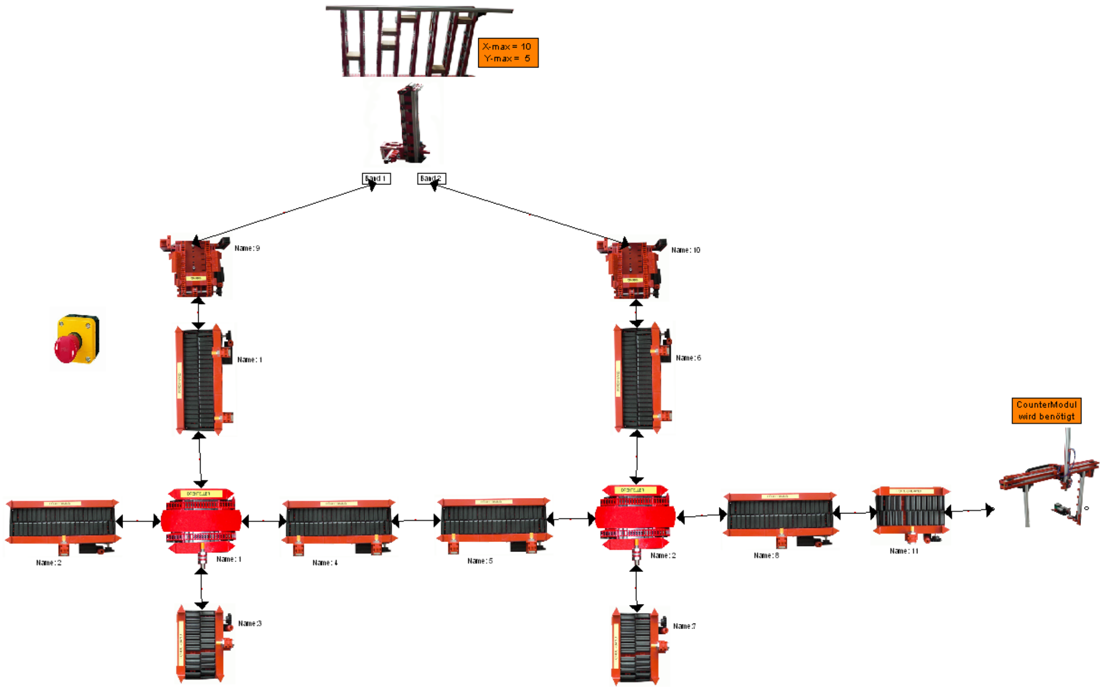


Figure 3.22: Assembly used for comparing DSL- and conventional development

3.6.1.1 Application development

Hardware modeling is ignored because this effort is equal in both variants. The reference workstation, which is used for the test is a Pentium Core 2 Duo with 2.2GHz (E4500) and 1GB RAM. The installed operating system is Windows XP Professional (SP2) and the Siemens® Software SIMATIC® Manager V 5.4 SP3. When doing conventional programming it is assumed that the basic function-blocks were available and adaptations are only done as in real world *clone and own* reuse, so the comparison is justifiable. Furthermore there is no debug time included, which is high in PLC programming and the skill of the developer is as high as those of the function-block developers.

Conventional development

First of all a graphical layout is drawn to help during the whole development process to recognize the neighbours and all required objects and their relations. The first programming step is to create the symbollist which maps all in- and output addresses to symbolic names which are used for programming (14 min). Then, the interface module (Connector) is created and includes all involved objects (15 min). Additionally, helper flags are needed for most of the instances (15 min). The main conveyor function with all the dependencies and interfaces is implemented next (70 min). The high bay racking adaptation includes reworking of all sensors in all functions and function-blocks and the interface to the conveyor system. The connections to the rack servicing unit which implies the job exchange and the location have to be defined (60 min).
The modifications of the gantry crane are moderate, only the packet width and the interface to the conveyor system has to be adapted (10 min).
Overall, the development modification time is about 3 hours, without any debugging, documenting and other output-files.

Domain specific development

Modeling the system starts with dragging and dropping the required objects into the plane. Then, the addresses are set, based on the hardware suggestion. Some of the conveyors were set manually, the others by using the helper functions. Existing documentation of the function-blocks was added to the single objects. This work was done in 11 minutes. This time includes generation of all required source files, and documentation too (see 3.4). The translation of the source files takes about 90 seconds with the SIMATIC® Manager. All in all the project is ready for use in about 13 minutes.

Comparison of development methods

Three hours development time with the conventional PLC programming method seems to be very optimistic: a work day with 8 hours would be more realistic but it could also take longer if a short documentation is added and some problems occur during programing. With regard to the persistent documentation and high quality code, or system updates and all other benefits of a software product line the 8 hours are easily justifiable and of course a lower bound. A more serious estimation is difficult.
In contrast the domain specific implementation is very much faster (13 minutes). An interesting point is the symbollist, which is generated first in conventional programming and takes the same time as the whole domain specific generation, where it is generated after all objects are known.

The increase in productivity is much more than the average which is between 300% and 1000% but the setup time has to be considered (see Section 3.6.1.2). Learning the software and modeling the language takes time (see A.2).

A major benefit is that no programming expert is needed after creating the software product line. The domain knowledge is stored in the generator and anyone can reuse this knowledge to create new applications after a short introduction. To create this SPL, however, a domain expert and a competent PLC programmer are required.

3.6.1.2 Time to market and costs

Discussing the goal *time to market* is difficult. The previous Section has shown a massive reduction in development time so new applications have a faster time to market. In contrast, the setup time of the full software product line takes about twice the time compared to that of a single system.
The BsC PLC automation project took about 430 hours and the implementation of this work was done in approximately 900 hours. These 900 hours can be split into two equal parts for implementing the SPLE PLC platform and the SPL architecture tool.
Figure 3.23 illustrates this times (simplified). The slopes equate the development time of the BsC project and the time of creating a domain specific application with the tool.

A comparison of the SPLE can be done in two ways which is discussed in the following description.

Single System development takes about 430 hours so the break even point in Figure 3.23 is between 2 and 3 systems. The often found break even point in the literature could be approved (see Section 2.1.2, Figure 2.2).

Clone and own is a common way of reuse in the industry so a comparison with the SPLE implementation is interesting. Alltough the speedup is about the factor 35, the implemented system needs about 59 different applications to reach the break even point. The slope of *clone and own* in Figure 2.2 is much more flat than that of single system development. Tue to the fact, that 8 hours development time for *clown and own* is the best case, 59 different systems is the worst case for the break even point in the implemented system.

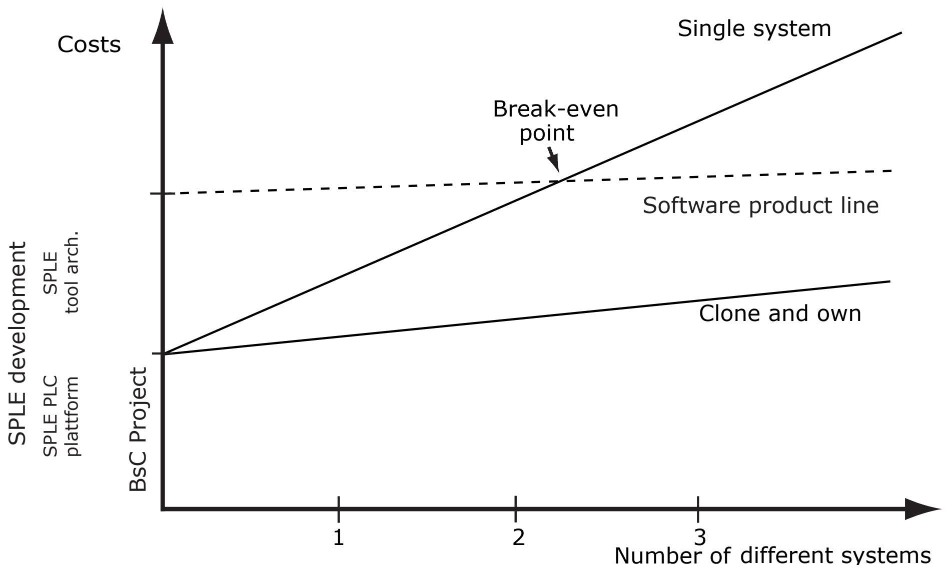
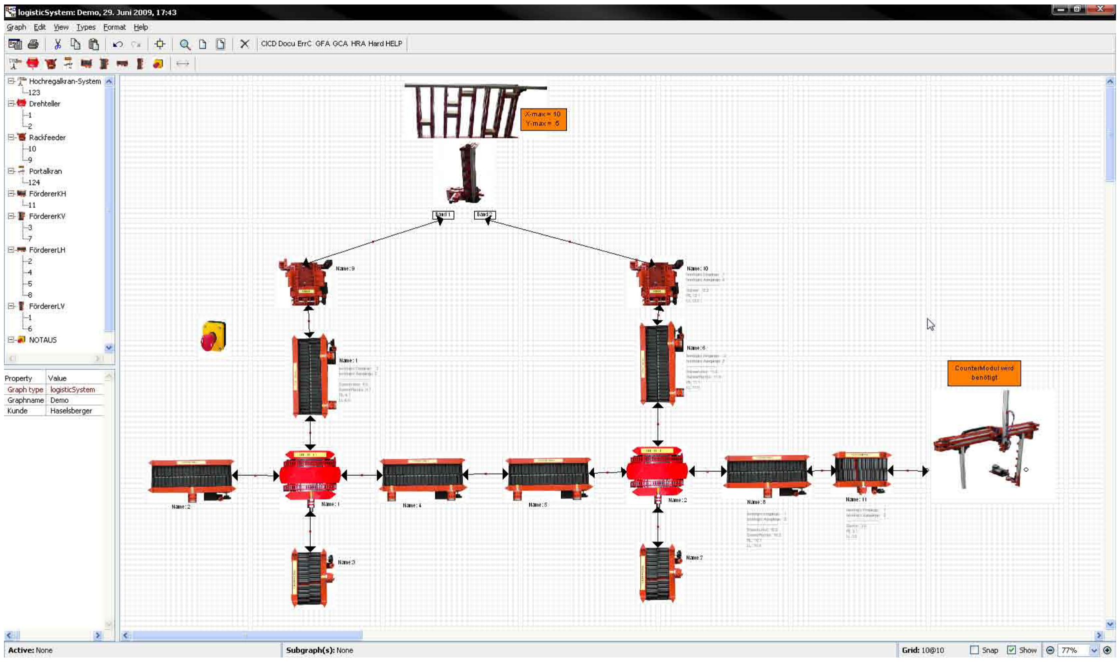


Figure 3.23: Evaluation: Costs

3.6.2 Process evaluation

The target process of Section 3.1 (Figure 3.3) was implemented. The MetaEdit+® DSM implementation offers a domain specific object pool which is extendable.
Composition of new applications is done in a graphical way (Figure 3.24). Adding and connecting objects is done by drag and drop. The instanced objects are customisable to specific needs.
After defining the application, all source files and the documentation are generated automatically with generators. These source files are copied to the PLC to use the new system.



Property	Value
Graph type	logisticSystem
Graphname	Demo
Kunde	Haselsberger

Figure 3.24: Implementation GUI

3.6.3 Technical evaluation

The requirement of reusing function-blocks could be achieved due to a robust platform architecture. A common interface between conveying units lead to a flexible application generation.
The generated source files are in the *.AWL format and may be imported into the SIMATIC® Manager to download them to the target PLC. Due to the format, they are read-, maintain- and translatable.
The MetaEdit+® architecture enables a network usage of the implementation.

3.6.4 Notes on quality

Section 2.1.2 deals with the theoretical quality increase of software product lines due to the review of the platform many times. This work approves this statement.

Many errors in the PLC code were found in different assemblies of the systems. These errors had no impact in previous, only in new compositions. So the effective reuse leads to less errors in the single components and improves the quality of the SPL and the applications.
The review and use of single components also leads to more comments in the code and a better readability.

The persistent documentation also brings an increase in quality. The level of information is tailored to the application and no unnecessary extra information is given.

Chapter 4

Outlook

General trends in the field of *software product line engineering* and *programmable logic controllers* are discussed in this chapter. Additionally some ideas for future work relating to this thesis are given.

4.1 General trends

Time to market, high quality and flexibility are, nowadays, important success factors. The market will be more aggressive for companies in the near future so companies have to deal with software reuse to be competitive. Software product line can be useful guides but they are not a magic formula. Other software approaches with respect of reuse and management may also be successful.

Programmable logic controllers are the main players in the automation sector and they will stay in this market segment, so the software development has to change. Not only code generators for PLC such as code generation from Petri nets will be the solution, the whole design flow should be reviewed.
A further important step will be a harmonisation between PLC vendors and a clear standard. So programs and configurations would be easily portable between different systems. In times where products and systems are getting cheaper and cheaper, the price is a main factor. So every PLC code should work on every system and the need for special, vendor dependent tools would be obvious.
The harmonisation between the two standards IEC 61131 and IEC 61499 is also challenging. The question is, if this is really the right way. Programmable logic controllers are now time-triggered but then they lose this behavior due to the events in the new standard. The basic idea is good, because reuse and interoperability are key motivations.

4.2 Ideas for future work

The current visualisation of the whole project is static, so it would be a nice challenge to create it dynamically from the application model. WinCC® is unsuitable because it uses predefined addresses for the control. An OPC Server (object linking and embedding for process control) in combination with a web-server could be generated on project demand. The current implementation provides all instructions already in HTML pages, so this can be the base.
Another possibility is the further development of variations and more functionality in the current implementation. For example, RFID-readers (Radio Frequency Identification) for detecting packets could be added to the model.
During this master thesis it became apparent that the hardware configuration may be done also automatically. Creating this configuration file out of the model definition would be a next step completing a full code generation for software and the hardware. An example for a usable hardware is generated as an XML file with a generator, which could be the basis for further development.
The transformation of the domain specific model only works with the SIMATIC® software (AWL which is different to other textual languages) and a Siemens® hardware currently, so it would be interesting to implement generators for other manufacturers.

Chapter 5

Concluding remarks

The design and implementation of a domain specific architecture for programmable logic controllers was the work of this thesis, applying a modern software paradigm to PLC. This thesis shows, that the paradigm software product line engineering is able to be implemented for a specific domain for PLC to achieve strategic reuse.

The thesis started with an introduction and motivation to create a domain specific architecture for PLC. Chapter 2 contained the related work, starting with Section 2.1 by discussing the field software product line engineering as state of the art in effective software reuse. Section 2.2 covered the topic of programmable logic controllers. After getting an overview on both main topics, Chapter 3 discussed the design and implementation of the domain specific architecture for the logistic system. The last Chapter 4 provided a further outlook on general trends and ideas for future work.

Appendix A

Appendix

A.1 Product line and DSM tool comparison criteria

Some of the criteria were taken from existing papers [DSF07; DRGN07; LCP+00] and new ones were added in cooperation with the Master thesis of Andrea Leitner [Lei09].

Nr.	Criterion	Definition
	Product Line Engineering criteria	
1	Attribute management	• Differentiate between SPL requirements and product requirements • Manage requirements attributes (identifier, description, justification, cost,...) • Ability to capture future requirements • Ability to capture new requirements during derivation • Autobuild with given specifications (mining)
2	Feature and variability modeling	• Help to model FODA-like concepts (feature decomposition, feature type, cardinalities, dependency links,...) • Support different abstraction levels • Support global constraints
3	Feature metamodel maturity	• Allow to define a PL metamodel • The tool should be unambiguous • Support product line evolution

Nr.	Criterion	Definition
4	Constraint checking and propagation	• Support validation checking for the PL model and metamodel • Check consistency of product model and PL model • Check consistency of model and artefact base • Support constraint propagation • Compare artefacts to a ”standard” • Rule-checking
5	Product derivation	• Help to derive specific products with guidance and visualization
6	Domain engineering management	• Support the creation of domain artefacts • Support the management of domain artefacts • Map domain artefacts to corresponding features • Search functions, to find a suitable artefacts
7	Repository	• Version management of artefacts, documents or possibility to integrate such a tool • Re-create any version of a product • Compare different versions of a product
Management criteria		
8	Traceability management	• Support requirements traceability with external documents • Support traceability management of inter-requirements links • Support metamodel traceability • Traceability between and within assets (linkage ...)

Nr.	Criterion	Definition
9	Impact analysis	• Perform impact analysis when changing requirements or models • Perform impact analysis when changing interlink requirements
10	Reporting	• Ability to generate reports
Technical criteria		
11	Access mode	• Allow multi-user access • Allow access with profiles (define the metamodel / use it)
12	Technical environment	• Support synchronization • Interoperability: support import and export from other tools (APIs, neutral format files, etc.)
13	Usability	• Intuitive usage • Stability and efficient support • Offer high accessibility of functions, zoom, views, ... • Ability to handle great amount of artefacts
14	Automatic filters	• Automatic filters on requirements presentations and report generation
15	Tool configuration	• The tool should be configurable for specific user needs • Adaption to current organisation
16	Extensibility	• Should be extensible to integrate existing platforms into the PL
17	Flexibility	• Changes should be possible at each stage of development (also in derived products).

Nr.	Criterion	Definition
18	AOB	• Tool costs and training costs /amortisation time • Light charge of installation, maintenance and migration cost • Vendor stability • Flexible licensing service

Table A.1: Criteria to rate SPL tools

A.2 Small evaluation of MetaEdit+

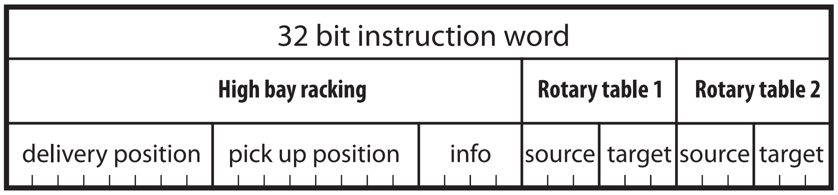
The installation of MetaEdit+® is very simple and causes no problem when using a Windows® environment. There are several examples which came within the distribution. The included help is organized as a website which is detailed and in most cases enough for upcoming problems.

Help and examples are also provided at www.metacase.com/ and in the forum http://www.metacase.com/forums/. The developers of the software try to help as well and quickly as they can. Very powerful illustrations are given with webcasts at http://www.metacase.com/resources.html. There is also a book available dealing with domain specific modeling which includes many examples for in MetaEdit+® (see [KT08]).

At the MetaCase® website it is possible to download an operative 30 day trial version. Getting familiar with MetaEdit+® takes about one week, but more refinement is learned later.
Giving an accurate time is difficult, because the learning is iterative project work. The scripting language MERL® is easy although control-loops have to be made in a recursive way. The *Generator Editor* helps with creating scripts. The API has to be used if the model is changed, because the MERL® scripts are read only. In this work Java was used to update the model via SOAP. MetaEdit+® provides a WSDL (Web Service description language) file which can be imported to Eclipse to generate all necessary classes for communicating with the server. The provided functionality is a little bit bulky but all required functions could be implemented.

A.3 Application workflow: Example

This Section shows an example work flow: from opening the software, to deploying the code to the PLC. The first step is to start MetaEdit+®. The Startup Launcher will list all available repositories. After login to the repository *master* MetaEdit+® starts this project. Now there are three columns: *Projects* which shows the current tree (master), *Graphs* with already assembled logistics systems and *Objects* of a graph. A new graph is created by using a toolbar-button and entering the properties. A stored graph is opened just by double-clicking. Now, an empty real modeling window appears (see Section A.12). Logistics objects are taken from the list and placed in the drawing area using drag and drop. Each object is parameterised by double-clicking. To connect the objects the relationship-object (arrow symbol) has to be used (An illustration of this iterative work is given at A.7). The addresses of the gantry crane and the high bay racking may be filled automatically with a generator by giving a start-address-parameter. After assembling the logistics system model an error-check is recommended which is implemented as a generator. If the logistics system is properly designed and error-free the code and documentation generation can start by activating the appropriate generator. Now all relevant files are generated and stored in the default working (owners documents) directory of MetaEdit+® in the *CODE* directory whereas the documentation is in the *FSDOKU* directory. Code can be imported into a SIMATIC® Manager project as source. The deployable blocks are generated by compiling the objects in the correct order (ascending filenames). Afterwards the blocks are available in the corresponding directory and ready to be deployed to the hardware.
Now the logistics system is ready to use and execute job commands after an initialising process. If a high bay racking or a gantry crane are part of the system, the implemented WinCC® GUI has to be started (see illustration A.4). There are graphical user interfaces for each object: the *init* button triggers initialisation. At the high bay racking the switch *manual/auto* has to be switched to auto, because the initial value is not defined in WinCC®. The format of the command is shown in Figure A.1. Generally the command of Weber and Wolf is used. It is a 32 bit word with all the information a packet needs for handling. The first 20 bit are for the high bay racking, and the last 12 bit are for the rotary tables.



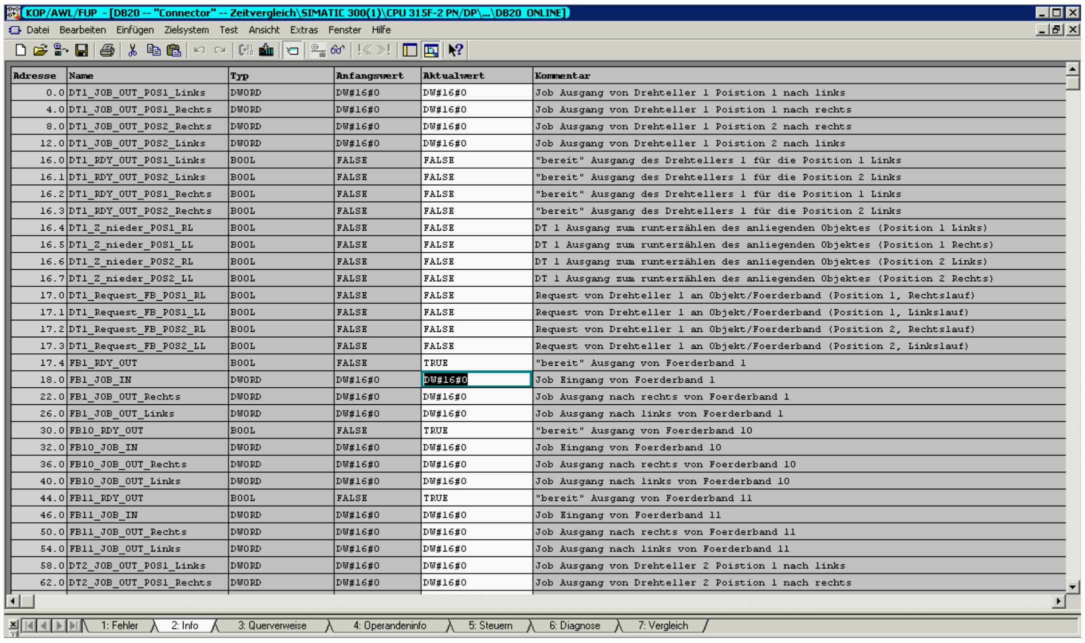
32 bit instruction word						
High bay racking			**Rotary table 1**		**Rotary table 2**	
delivery position	pick up position	info	source	target	source	target

Figure A.1: Transport instruction word

The connector data block handles data exchange between all objects, so a control command must be written to the corresponding address in the connector datablock (DB20). The simplest way to set values in the connector data block is to start the online-view of the hardware and open the block. To submit a job, for example, to a conveyor the instruction has to be copied into this field.

BIT		Instruction	BIT		Instruction
d	c		b	a	
0	0	Nothing	0	0	Nothing
1	0	Deploy to conveyor	1	0	Load from conveyor
1	1	Deploy to high bay racking	1	1	Load from highrack

Table A.2: Instruction notation


KOP/AWL/FUP - [DB20 -- "Connector" -- Zeitvergleich\SIMATIC 300(1)\CPU 315F-2 PN/DP\...\DB20 ONLINE]
Datei Bearbeiten Einfügen Zielsystem Test Ansicht Extras Fenster Hilfe
1: Fehler 2: Info 3: Querverweise 4: Operandeninfo 5: Steuern 6: Diagnose 7: Vergleich

Adresse	Name	Typ	Anfangswert	Aktualwert	Kommentar
0.0	DT1_JOB_OUT_POS1_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 1 Poistion 1 nach links
4.0	DT1_JOB_OUT_POS1_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 1 Poistion 1 nach rechts
8.0	DT1_JOB_OUT_POS2_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 1 Poistion 2 nach rechts
12.0	DT1_JOB_OUT_POS2_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 1 Poistion 2 nach links
16.0	DT1_RDY_OUT_POS1_Links	BOOL	FALSE	FALSE	"bereit" Ausgang des Drehtellers 1 für die Position 1 Links
16.1	DT1_RDY_OUT_POS2_Links	BOOL	FALSE	FALSE	"bereit" Ausgang des Drehtellers 1 für die Position 2 Links
16.2	DT1_RDY_OUT_POS1_Rechts	BOOL	FALSE	FALSE	"bereit" Ausgang des Drehtellers 1 für die Position 1 Links
16.3	DT1_RDY_OUT_POS2_Rechts	BOOL	FALSE	FALSE	"bereit" Ausgang des Drehtellers 1 für die Position 2 Links
16.4	DT1_Z_nieder_POS1_RL	BOOL	FALSE	FALSE	DT 1 Ausgang zum runterzählen des anliegenden Objektes (Position 1 Links)
16.5	DT1_Z_nieder_POS1_LL	BOOL	FALSE	FALSE	DT 1 Ausgang zum runterzählen des anliegenden Objektes (Position 1 Rechts)
16.6	DT1_Z_nieder_POS2_RL	BOOL	FALSE	FALSE	DT 1 Ausgang zum runterzählen des anliegenden Objektes (Position 2 Links)
16.7	DT1_Z_nieder_POS2_LL	BOOL	FALSE	FALSE	DT 1 Ausgang zum runterzählen des anliegenden Objektes (Position 2 Rechts)
17.0	DT1_Request_FB_POS1_RL	BOOL	FALSE	FALSE	Request von Drehteller 1 an Objekt/Foerderband (Position 1, Rechtslauf)
17.1	DT1_Request_FB_POS1_LL	BOOL	FALSE	FALSE	Request von Drehteller 1 an Objekt/Foerderband (Position 1, Linkslauf)
17.2	DT1_Request_FB_POS2_RL	BOOL	FALSE	FALSE	Request von Drehteller 1 an Objekt/Foerderband (Position 2, Rechtslauf)
17.3	DT1_Request_FB_POS2_LL	BOOL	FALSE	FALSE	Request von Drehteller 1 an Objekt/Foerderband (Position 2, Linkslauf)
17.4	FB1_RDY_OUT	BOOL	FALSE	TRUE	"bereit" Ausgang von Foerderband 1
18.0	FB1_JOB_IN	DWORD	DW#16#0	DW#16#0	Job Eingang von Foerderband 1
22.0	FB1_JOB_OUT_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang nach rechts von Foerderband 1
26.0	FB1_JOB_OUT_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang nach links von Foerderband 1
30.0	FB10_RDY_OUT	BOOL	FALSE	TRUE	"bereit" Ausgang von Foerderband 10
32.0	FB10_JOB_IN	DWORD	DW#16#0	DW#16#0	Job Eingang von Foerderband 10
36.0	FB10_JOB_OUT_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang nach rechts von Foerderband 10
40.0	FB10_JOB_OUT_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang nach links von Foerderband 10
44.0	FB11_RDY_OUT	BOOL	FALSE	TRUE	"bereit" Ausgang von Foerderband 11
46.0	FB11_JOB_IN	DWORD	DW#16#0	DW#16#0	Job Eingang von Foerderband 11
50.0	FB11_JOB_OUT_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang nach rechts von Foerderband 11
54.0	FB11_JOB_OUT_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang nach links von Foerderband 11
58.0	DT2_JOB_OUT_POS1_Links	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 2 Poistion 1 nach links
62.0	DT2_JOB_OUT_POS1_Rechts	DWORD	DW#16#0	DW#16#0	Job Ausgang von Drehteller 2 Poistion 1 nach rechts

Figure A.2: Connector data exchange

The documentation helps to generate the commands which is illustrated in figure A.10. A list shows the available targets and the corresponding numbers. These numbers are decimal, but the SimaticManager needs hexadecimal values. Therefore, the calculator in the page can be used, which is also able to include high bay racking options. The instruction to send a packet to conveyor 11 would be 130, for example. By copying this value to the calculator field and clicking *Generate* the command for the data block is generated. If a high bay racking option is selected and a target position is given, the instruction is generated accordingly. After copying this value which always starts with DW#16# to a JOB_IN_X in the *Connector* data block and storing the value to the hardware the conveyor is ready to execute the instruction. This process is a bit difficult, but dynamic creation of the system would need a dynamic user-interface. This could be work for the future. The usage of the high bay racking and the gantry crane is simpler because there exists a WinCC® interface where the JOB may be inserted as a decimal value. These parts are semi-static and modified from [GH09]. A job word must be stored to the data block before a packet is on the conveyor.

Figure A.3 again shows the bits for the rotary table. If a packet should go from left to top at rotary table one, the corresponding command without high bay racking interaction would be : XXXXXXXX.XXXXXXXX.XXXX.XXXXXX.010011

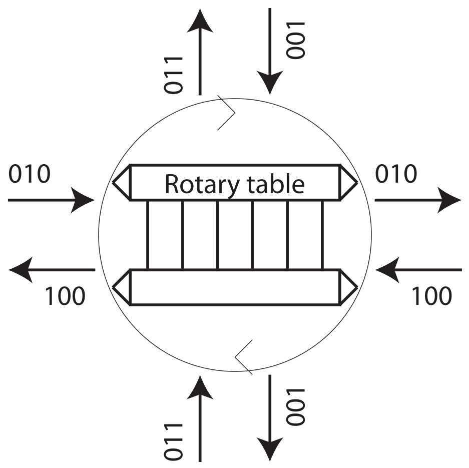
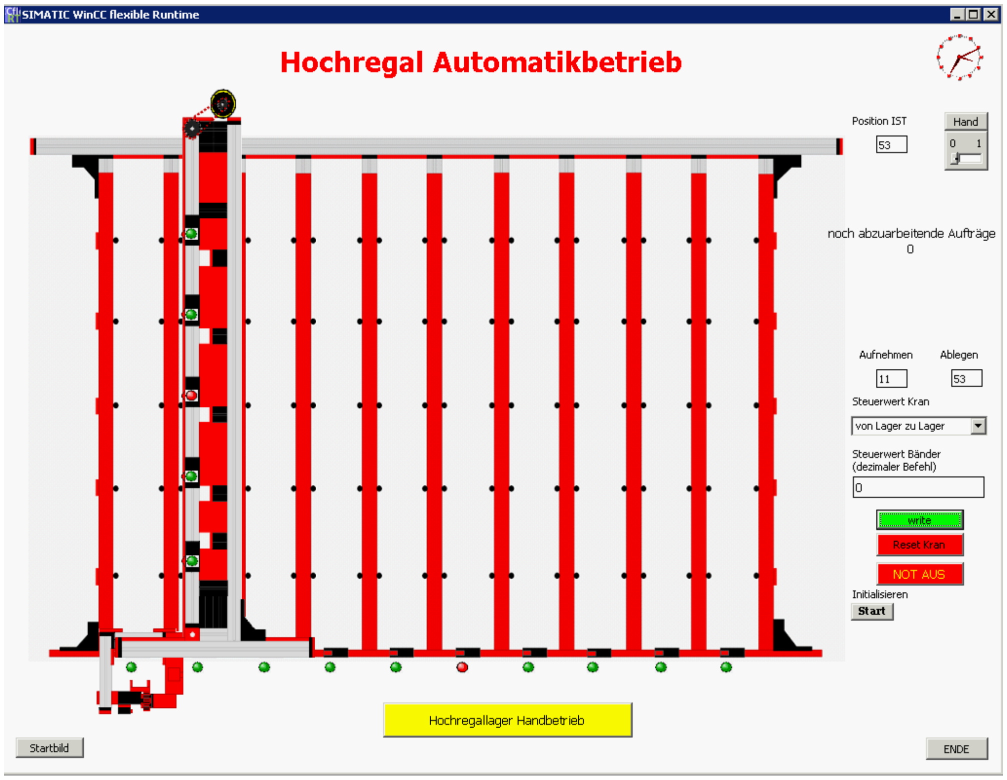


Figure A.3: Rotary table direction codes


SIMATIC WinCC flexible Runtime
Hochregal Automatikbetrieb
Hand
0 1
write
Reset Kran
NOT AUS
Initialisieren
Start
Hochregallager Handbetrieb
Startbild
ENDE

Position IST
53

noch abzuarbeitende Aufträge
0

Aufnehmen 11
Ablegen 53

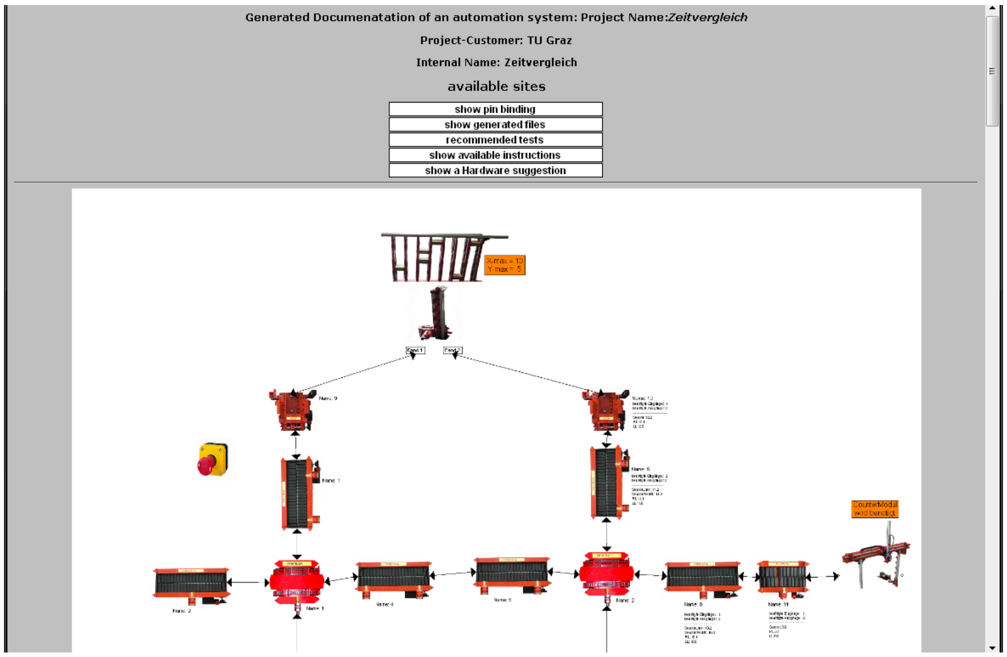
Steuerwert Kran
von Lager zu Lager

Steuerwert Bänder (dezimaler Befehl)
0

Figure A.4: WinCC High bay racking GUI

A.4 Generated documentation: Screenshots

This Section shows screenshots of the generated documentation.



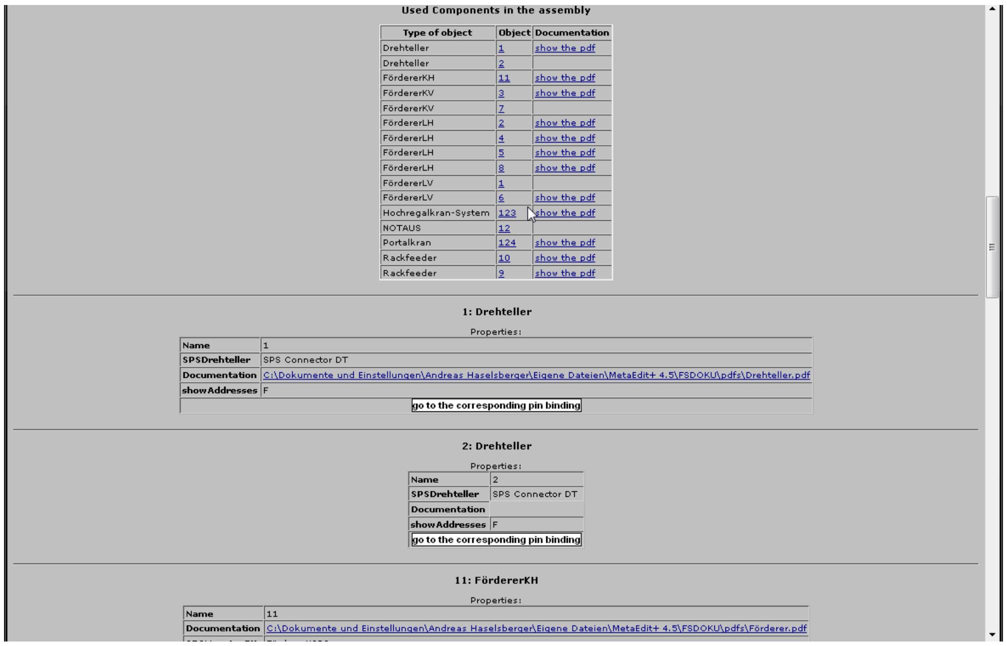
Generated Documenatation of an automation system: Project Name:*Zeitvergleich*

Project-Customer: TU Graz

Internal Name: Zeitvergleich

available sites

Figure A.5: Documentation: Main page (1) Physical representation



Used Components in the assembly

Type of object	Object	Documentation
Drehteller	1	show the pdf
Drehteller	2	
FördererKH	11	show the pdf
FördererKV	3	show the pdf
FördererKV	7	
FördererLH	2	show the pdf
FördererLH	4	show the pdf
FördererLH	5	show the pdf
FördererLH	8	show the pdf
FördererLV	1	
FördererLV	6	show the pdf
Hochregalkran-System	123	show the pdf
NOTAUS	12	
Portalkran	124	show the pdf
Rackfeeder	10	show the pdf
Rackfeeder	9	show the pdf

1: Drehteller

Properties:

Name	1
SPSDrehteller	SPS Connector DT
Documentation	C:\Dokumente und Einstellungen\Andreas Haselsberger\Eigene Dateien\MetaEdit+ 4.5\FSDOKU\pdfs\Drehteller.pdf
showAddresses	F

2: Drehteller

Properties:

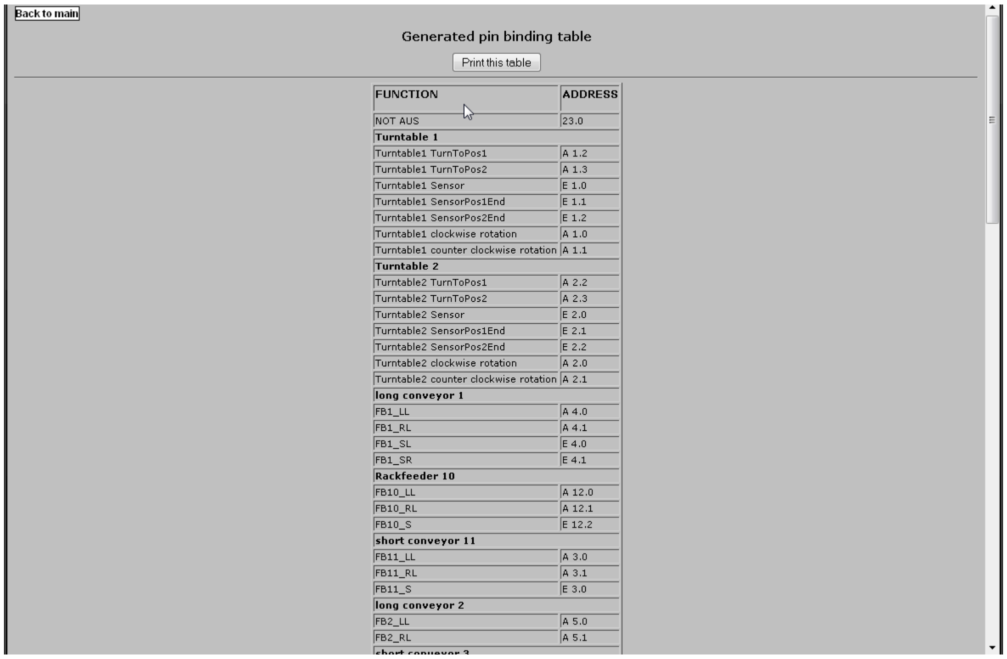
Name	2
SPSDrehteller	SPS Connector DT
Documentation	
showAddresses	F

11: FördererKH

Properties:

Name	11
Documentation	C:\Dokumente und Einstellungen\Andreas Haselsberger\Eigene Dateien\MetaEdit+ 4.5\FSDOKU\pdfs\Förderer.pdf

Figure A.6: Documentation: Main page (2) Component listing

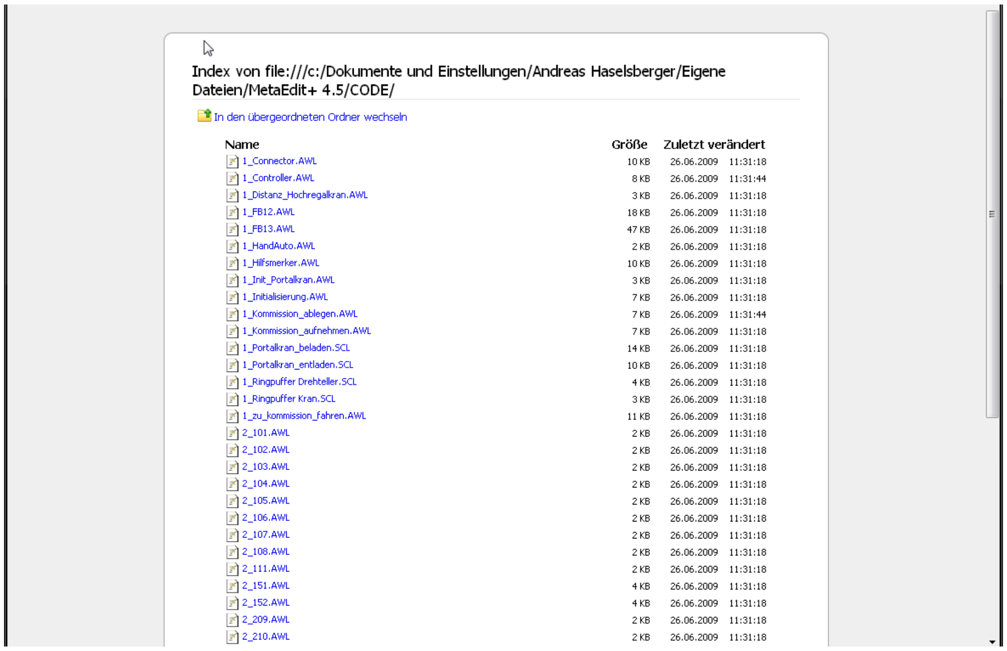

Back to main

Generated pin binding table

Print this table

FUNCTION	ADDRESS
NOT AUS	23.0
Turntable 1	
Turntable1 TurnToPos1	A 1.2
Turntable1 TurnToPos2	A 1.3
Turntable1 Sensor	E 1.0
Turntable1 SensorPos1End	E 1.1
Turntable1 SensorPos2End	E 1.2
Turntable1 clockwise rotation	A 1.0
Turntable1 counter clockwise rotation	A 1.1
Turntable 2	
Turntable2 TurnToPos1	A 2.2
Turntable2 TurnToPos2	A 2.3
Turntable2 Sensor	E 2.0
Turntable2 SensorPos1End	E 2.1
Turntable2 SensorPos2End	E 2.2
Turntable2 clockwise rotation	A 2.0
Turntable2 counter clockwise rotation	A 2.1
long conveyor 1	
FB1_LL	A 4.0
FB1_RL	A 4.1
FB1_SL	E 4.0
FB1_SR	E 4.1
Rackfeeder 10	
FB10_LL	A 12.0
FB10_RL	A 12.1
FB10_S	E 12.2
short conveyor 11	
FB11_LL	A 3.0
FB11_RL	A 3.1
FB11_S	E 3.0
long conveyor 2	
FB2_LL	A 5.0
FB2_RL	A 5.1
short conveyor 3	

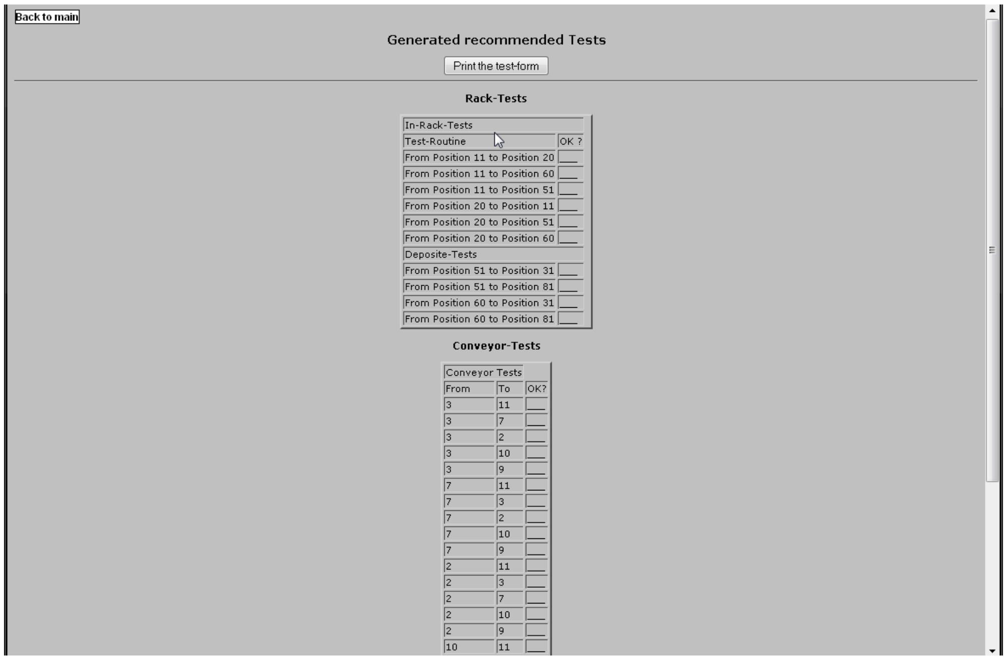
Figure A.7: Documentation: Pin binding


Index von file:///c:/Dokumente und Einstellungen/Andreas Haselsberger/Eigene Dateien/MetaEdit+ 4.5/CODE/

In den übergeordneten Ordner wechseln

Name	Größe	Zuletzt verändert	
1_Connector.AWL	10 KB	26.06.2009	11:31:18
1_Controller.AWL	8 KB	26.06.2009	11:31:44
1_Distanz_Hochregalkran.AWL	3 KB	26.06.2009	11:31:18
1_FB12.AWL	18 KB	26.06.2009	11:31:18
1_FB13.AWL	47 KB	26.06.2009	11:31:18
1_HandAuto.AWL	2 KB	26.06.2009	11:31:18
1_Hilfsmerker.AWL	10 KB	26.06.2009	11:31:18
1_Init_Portalkran.AWL	3 KB	26.06.2009	11:31:18
1_Initialisierung.AWL	7 KB	26.06.2009	11:31:18
1_Kommission_ablegen.AWL	7 KB	26.06.2009	11:31:44
1_Kommission_aufnehmen.AWL	7 KB	26.06.2009	11:31:18
1_Portalkran_beladen.SCL	14 KB	26.06.2009	11:31:18
1_Portalkran_entladen.SCL	10 KB	26.06.2009	11:31:18
1_Ringpuffer Drehteller.SCL	4 KB	26.06.2009	11:31:18
1_Ringpuffer Kran.SCL	3 KB	26.06.2009	11:31:18
1_zu_kommission_fahren.AWL	11 KB	26.06.2009	11:31:18
2_101.AWL	2 KB	26.06.2009	11:31:18
2_102.AWL	2 KB	26.06.2009	11:31:18
2_103.AWL	2 KB	26.06.2009	11:31:18
2_104.AWL	2 KB	26.06.2009	11:31:18
2_105.AWL	2 KB	26.06.2009	11:31:18
2_106.AWL	2 KB	26.06.2009	11:31:18
2_107.AWL	2 KB	26.06.2009	11:31:18
2_108.AWL	2 KB	26.06.2009	11:31:18
2_111.AWL	2 KB	26.06.2009	11:31:18
2_151.AWL	4 KB	26.06.2009	11:31:18
2_152.AWL	4 KB	26.06.2009	11:31:18
2_209.AWL	2 KB	26.06.2009	11:31:18
2_210.AWL	2 KB	26.06.2009	11:31:18

Figure A.8: Documentation: Generated files



Generated recommended Tests

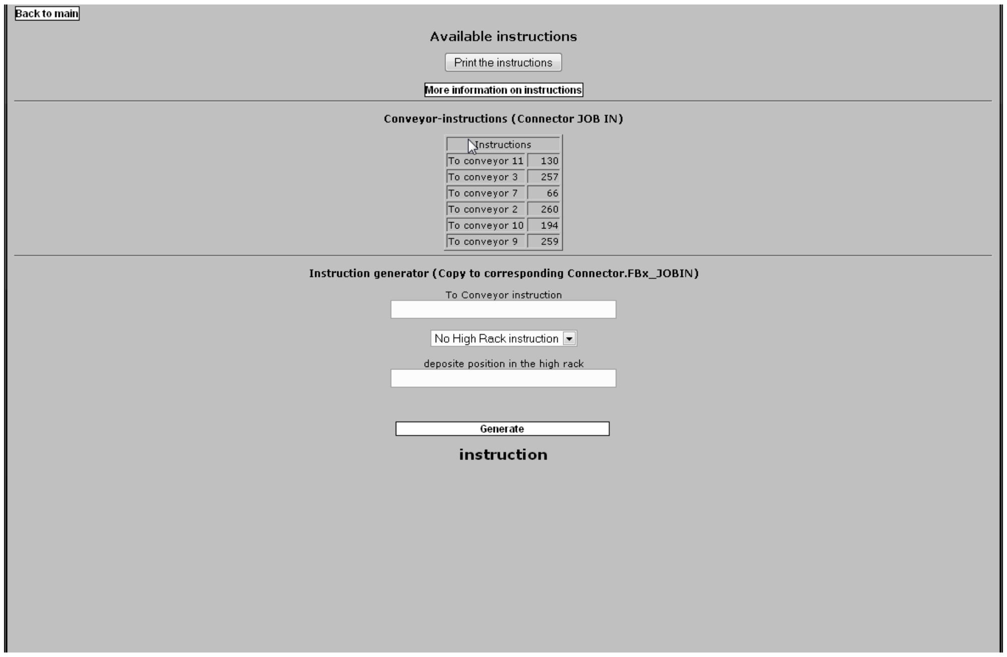
Rack-Tests

In-Rack-Tests	
Test-Routine	OK ?
From Position 11 to Position 20	___
From Position 11 to Position 60	___
From Position 11 to Position 51	___
From Position 20 to Position 11	___
From Position 20 to Position 51	___
From Position 20 to Position 60	___
Deposite-Tests	
From Position 51 to Position 31	___
From Position 51 to Position 81	___
From Position 60 to Position 31	___
From Position 60 to Position 81	___

Conveyor-Tests

Conveyor Tests		
From	To	OK?
3	11	___
3	7	___
3	2	___
3	10	___
3	9	___
7	11	___
7	3	___
7	2	___
7	10	___
7	9	___
2	11	___
2	3	___
2	7	___
2	10	___
2	9	___
10	11	___

Figure A.9: Documentation: Recommended tests



Available instructions

Conveyor-instructions (Connector JOB IN)

Instructions	
To conveyor 11	130
To conveyor 3	257
To conveyor 7	66
To conveyor 2	260
To conveyor 10	194
To conveyor 9	259

Instruction generator (Copy to corresponding Connector.FBx_JOBIN)

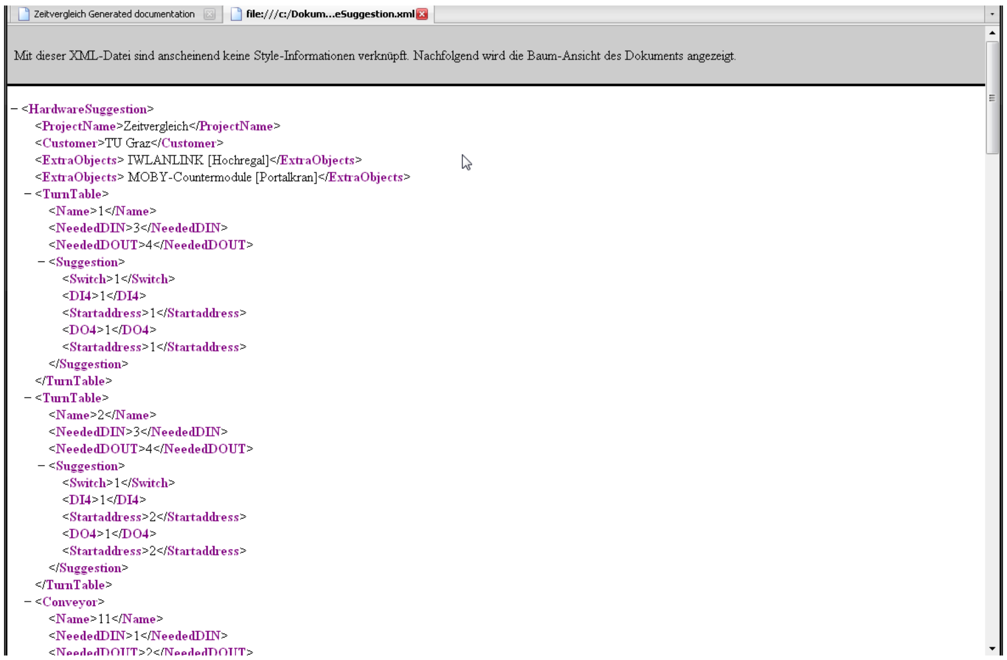
To Conveyor instruction

No High Rack instruction

deposite position in the high rack

instruction

Figure A.10: Documentation: Instructions


Zeitvergleich Generated documentation
file:///c:/Dokum...eSuggestion.xml

Mit dieser XML-Datei sind anscheinend keine Style-Informationen verknüpft. Nachfolgend wird die Baum-Ansicht des Dokuments angezeigt.

```
- <HardwareSuggestion>
    <ProjectName>Zeitvergleich</ProjectName>
    <Customer>TU Graz</Customer>
    <ExtraObjects> IWLANLINK [Hochregal]</ExtraObjects>
    <ExtraObjects> MOBY-Countermodule [Portalkran]</ExtraObjects>
  - <TurnTable>
      <Name>1</Name>
      <NeededDIN>3</NeededDIN>
      <NeededDOUT>4</NeededDOUT>
    - <Suggestion>
        <Switch>1</Switch>
        <DI4>1</DI4>
        <Startaddress>1</Startaddress>
        <DO4>1</DO4>
        <Startaddress>1</Startaddress>
      </Suggestion>
    </TurnTable>
  - <TurnTable>
      <Name>2</Name>
      <NeededDIN>3</NeededDIN>
      <NeededDOUT>4</NeededDOUT>
    - <Suggestion>
        <Switch>1</Switch>
        <DI4>1</DI4>
        <Startaddress>2</Startaddress>
        <DO4>1</DO4>
        <Startaddress>2</Startaddress>
      </Suggestion>
    </TurnTable>
  - <Conveyor>
      <Name>11</Name>
      <NeededDIN>1</NeededDIN>
      <NeededDOUT>2</NeededDOUT>
```

Figure A.11: Documentation: Hardware suggestion

A.5 Installation guide

This installation guide assumes a running Windows environment with the installed Siemens® SIMATIC® toolkit. At least the SIMATIC® Manager and WinnCC® flexible. If the repository is already set up and a new client is added please skip the steps three to six. The installation disk of the implementation contains the following structure:

logisticSystem

FSDOKU

CODE

JavaSource

AddressChecker.bat

CleanUpCode.bat

GCPrefill.bat

HRPrefill.bat

Please follow the next instructions to install the implementation:

1. Get a license key for MetaEdit+® from the Keymanager at the institute.
2. Install MetaEdit+® with the key. MetaEdit+® will create the default working directory in the users document order. Then a restart should be done.
3. Copy the Modeldefinition-folders *logisticSystem*, *FSDOKU* and *CODE* to the working directory. The *CODE* directory is empty and the generated code will be placed here. *FSDOKU* contains the generated web based documentation and a *pdfs* folder which includes sample documentations of single objects.
4. Copy the Java classes to a favoured folder or to a new folder in the working directory.
5. Copy the four *.bat files: *CleanUpCode*, *AddressChecker*, *HRPrefill* and *GCPrefill* to the working directory. They will call the java classes so an adoption has to be done to update the location.
6. Now MetaEdit+® can be started and the repository *logisticSystem* should be available and the software is ready to use. Otherwise please check the working directory which must contain this folder (Step 3).
7. Optional network configuration: If this client uses a remote repository the program can be started after the installation: The menu contains the option Repository / Add Repository to List which allows connection to a network repository.

A.6 Detailed design document: Hardware suggestion

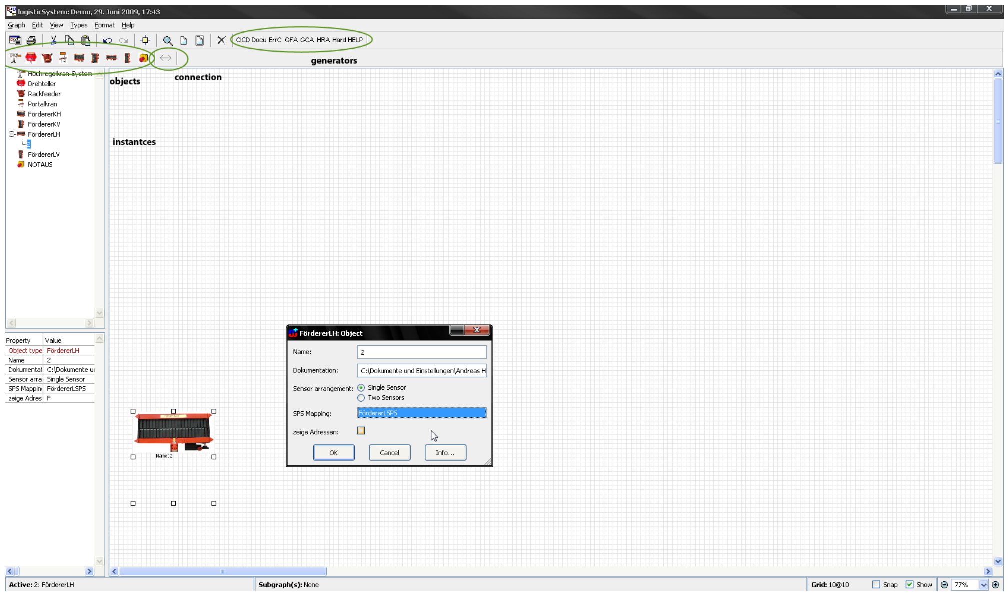
Listening A.1 shows the DDD for the hardware suggestion.

Listing A.1: DDD for hardware suggestion XML

```
<?xml version="1.0" encoding="UTF-8"?>

<!ELEMENT Switches (#PCDATA)>
<!ELEMENT Switch (#PCDATA)>
<!ELEMENT DO8 (#PCDATA)>
<!ELEMENT DO4 (#PCDATA)>
<!ELEMENT DI8 (#PCDATA)>
<!ELEMENT DI4 (#PCDATA)>
<!ELEMENT Customer (#PCDATA)>
<!ELEMENT Startaddress (#PCDATA)>
<!ELEMENT ProjectName (#PCDATA)>
<!ELEMENT NeededDOUT (#PCDATA)>
<!ELEMENT NeededDIN (#PCDATA)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT ExtraObjects (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
<!ELEMENT Suggestion (((Description, Mobile_Unit,
   GroundFloor_Unit) | (DI8, DO8, DI4, DO4, Switches) | (Switch,
    ((DI8, Startaddress, DO8) | (DI4, Startaddress, DO4)),
   Startaddress)))>
<!ELEMENT rotary table ((Name, NeededDIN, NeededDOUT,
   Suggestion))>
<!ELEMENT Conveyor ((Name, NeededDIN, NeededDOUT, Suggestion))>
<!ELEMENT High_Rack ((Name, NeededDIN, NeededDOUT, Suggestion))>
<!ELEMENT Mobile_Unit ((Switch, DO8, Startaddress, DI4,
   Startaddress))>
<!ELEMENT GroundFloor_Unit ((Switch, DI8, Startaddress, DI8,
   Startaddress, DI4, Startaddress))>
<!ELEMENT Gantry_Crane ((Name, NeededDIN, NeededDOUT, Suggestion
   ))>
<!ELEMENT OverAllSums ((NeededDIN, NeededDOUT, Suggestion))>
<!ELEMENT HardwareSuggestion ((ProjectName, Customer,
   ExtraObjects+, rotary table +, Conveyor+, High_Rack,
   Gantry_Crane, OverAllSums))>
```

A.7 Sequential application generation: Screenshots



FördererLH: Object

Name: 2

Dokumentation: C:\Dokumente und Einstellungen\Andreas H

Sensor arrangement: [✓ Single Sensor] Two Sensors

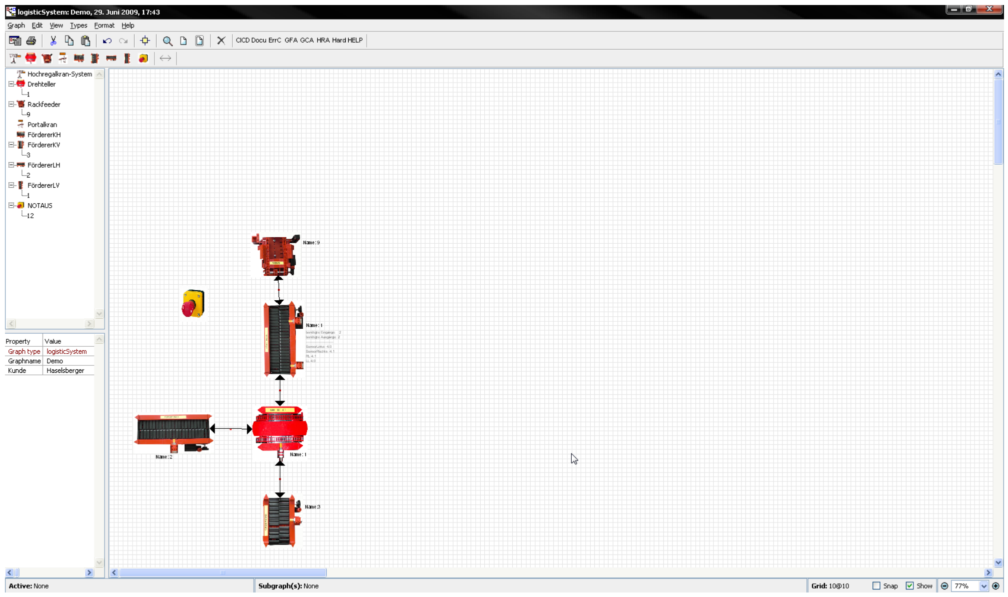
SPS Mapping: FördererLSPS

zeige Adressen: ☐

OK Cancel Info...

Property	Value
Object type	FördererLH
Name	2
Dokumentat	C:\Dokumente u
Sensor arra	Single Sensor
SPS Mappin	FördererLSPS
zeige Adres	F

Figure A.12: Application generation 1: New domain component (+parameterisation)



Property	Value
Graph type	logisticSystem
Graphname	Demo
Kunde	Haselsberger

Figure A.13: Application generation 2: Building the logistics system

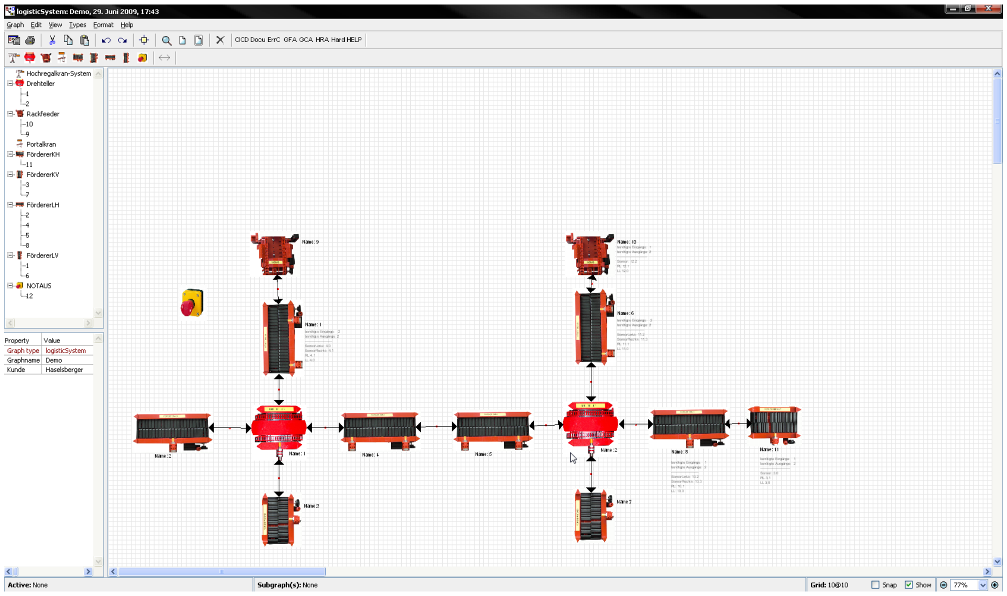


Figure A.14: Application generation 3: Building the logistics system

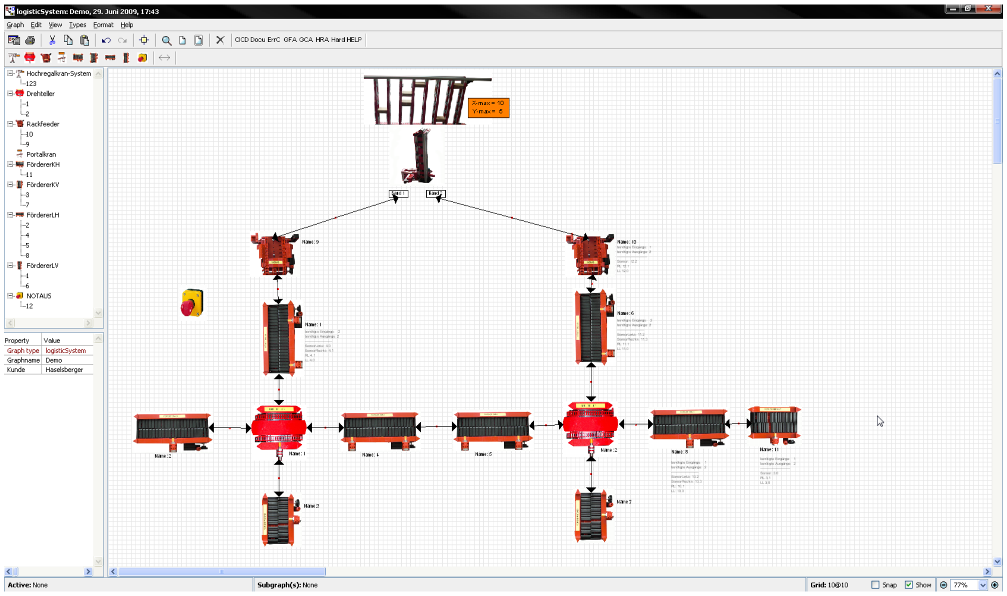
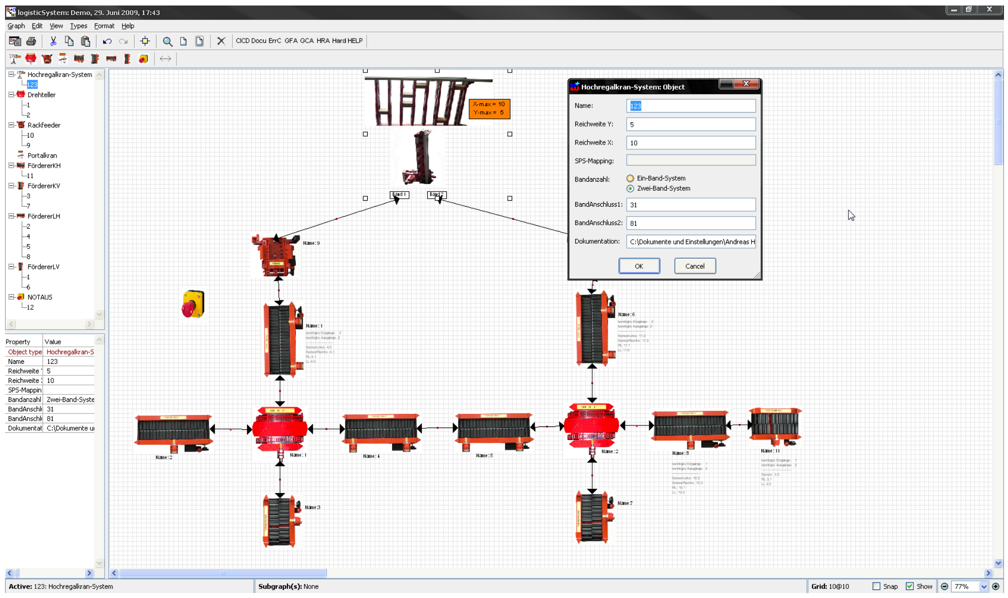


Figure A.15: Application generation 4: Building the logistics system

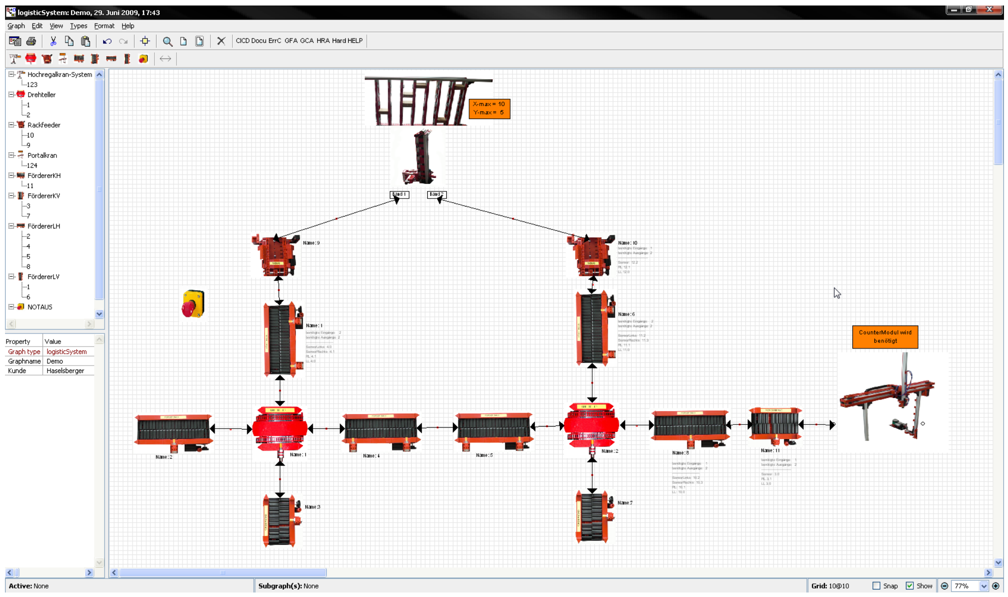


Name: 123
Reichweite Y: 5
Reichweite X: 10
SPS-Mapping:
Bandanzahl: Ein-Band-System [✓ Zwei-Band-System]
BandAnschluss1: 31
BandAnschluss2: 81
Dokumentation: C:\Dokumente und Einstellungen\Andreas H

Property	Value
Object type	Hochregalkran-S
Name	123
Reichweite '	5
Reichweite :	10
SPS-Mappin	
Bandanzahl	Zwei-Band-Syste
BandAnschl	31
BandAnschl	81
Dokumentat	C:\Dokumente u

Active: 123: Hochregalkran-System **Subgraph(s):** None **Grid:** 10@10 Snap [✓ Show] 77%

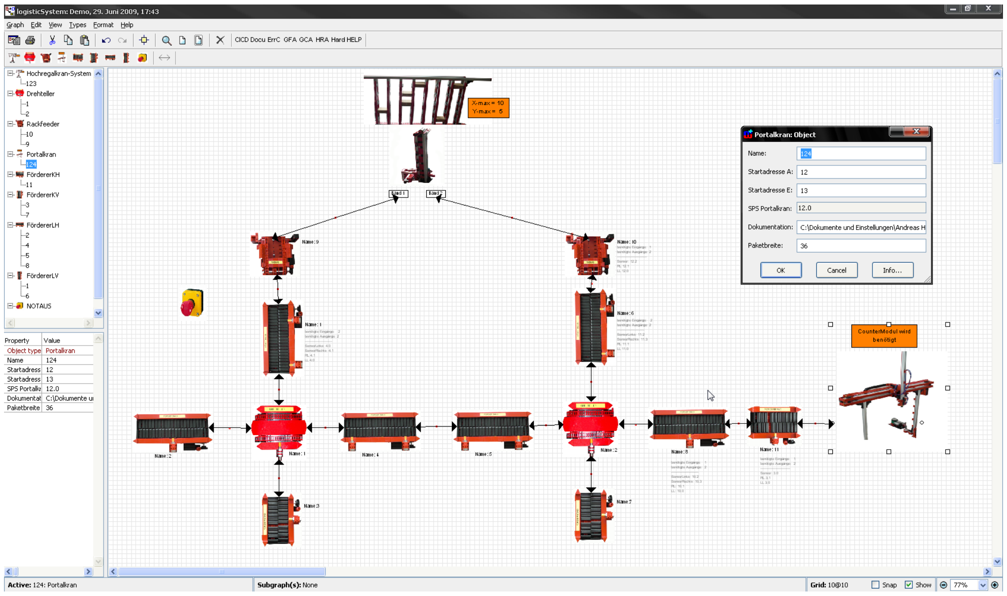
Figure A.16: Application generation 5: Parameterisation


logisticSystem: Demo, 29. Juni 2009, 17:43
Graph Edit View Types Format Help
CICD Docu ErrC GFA GCA HRA Hard HELP
Hochregalkran-System
123
Drehteller
1
2
Rackfeeder
10
9
Portalkran
124
FördererKH
11
FördererKV
3
7
FördererLH
2
4
5
8
FördererLV
1
6
NOTAUS
X-max = 10
Y-max = 5
CounterModul wird benötigt

Property	Value
Graph type	logisticSystem
Graphname	Demo
Kunde	Haselsberger

Active: None **Subgraph(s):** None **Grid:** 10@10 Snap [✓ Show] 77%

Figure A.17: Application generation 6: Building the logistics system


logisticSystem: Demo, 29. Juni 2009, 17:43
Graph Edit View Types Format Help
CICD Docu ErrC GFA GCA HRA Hard HELP
Hochregalkran-System
123
Drehteller
Rackfeeder
Portalkran
FördererKH
FördererKV
FördererLH
FördererLV
NOTAUS
X-max = 10
Y-max = 5
CounterModul wird benötigt
OK
Cancel
Info...

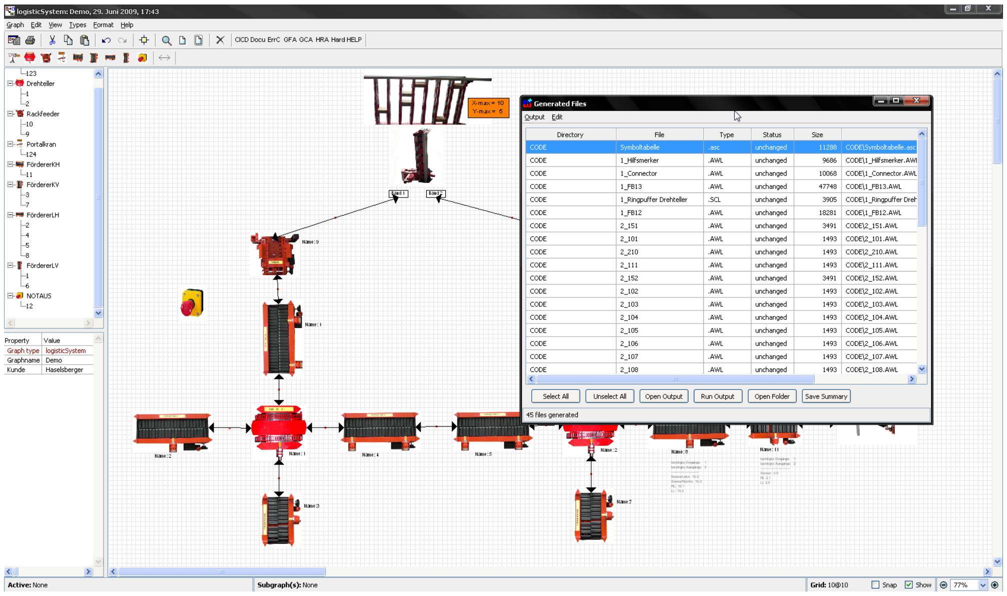
Portalkran: Object

Name: 124
Startadresse A: 12
Startadresse E: 13
SPS Portalkran: 12.0
Dokumentation: C:\Dokumente und Einstellungen\Andreas H
Paketbreite: 36

Property	Value
Object type	Portalkran
Name	124
Startadress	12
Startadress	13
SPS Portalkr	12.0
Dokumentat	C:\Dokumente u
Paketbreite	36

Active: 124: Portalkran   Subgraph(s): None   Grid: 10@10   Snap   ✓ Show   77%

Figure A.18: Application generation 7: Parameterisation


logisticSystem: Demo, 29. Juni 2009, 17:43
Graph Edit View Types Format Help
CICD Docu ErrC GFA GCA HRA Hard HELP
Output Edit
Select All
Unselect All
Open Output
Run Output
Open Folder
Save Summary

Generated Files

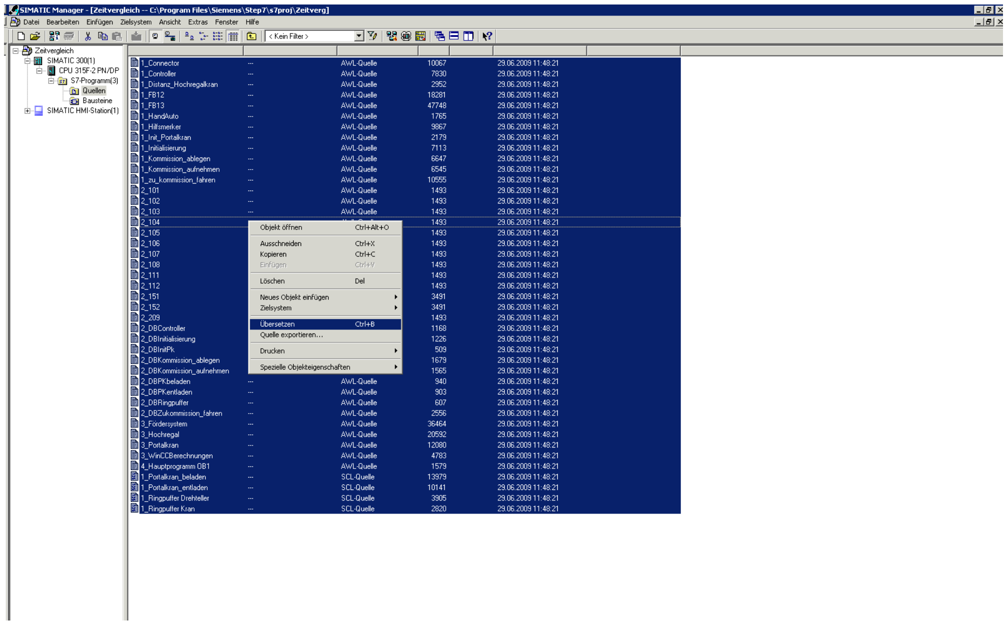
Directory	File	Type	Status	Size	
CODE	Symboltabelle	.asc	unchanged	11288	CODE\Symboltabelle.asc
CODE	1_Hilfsmerker	.AWL	unchanged	9686	CODE\1_Hilfsmerker.AWI
CODE	1_Connector	.AWL	unchanged	10068	CODE\1_Connector.AWL
CODE	1_FB13	.AWL	unchanged	47748	CODE\1_FB13.AWL
CODE	1_Ringpuffer Drehteller	.SCL	unchanged	3905	CODE\1_Ringpuffer Dreh
CODE	1_FB12	.AWL	unchanged	18281	CODE\1_FB12.AWL
CODE	2_151	.AWL	unchanged	3491	CODE\2_151.AWL
CODE	2_101	.AWL	unchanged	1493	CODE\2_101.AWL
CODE	2_210	.AWL	unchanged	1493	CODE\2_210.AWL
CODE	2_111	.AWL	unchanged	1493	CODE\2_111.AWL
CODE	2_152	.AWL	unchanged	3491	CODE\2_152.AWL
CODE	2_102	.AWL	unchanged	1493	CODE\2_102.AWL
CODE	2_103	.AWL	unchanged	1493	CODE\2_103.AWL
CODE	2_104	.AWL	unchanged	1493	CODE\2_104.AWL
CODE	2_105	.AWL	unchanged	1493	CODE\2_105.AWL
CODE	2_106	.AWL	unchanged	1493	CODE\2_106.AWL
CODE	2_107	.AWL	unchanged	1493	CODE\2_107.AWL
CODE	2_108	.AWL	unchanged	1493	CODE\2_108.AWL

45 files generated

Property	Value
Graph type	logisticSystem
Graphname	Demo
Kunde	Haselsberger

Active: None   Subgraph(s): None   Grid: 10@10   Snap   ✓ Show   77%

Figure A.19: Application generation 8: Code generation



1_Connector	---	AWL-Quelle	10067	29.06.2009 11:48:21
1_Controller	---	AWL-Quelle	7830	29.06.2009 11:48:21
1_Distanz_Hochregalkran	---	AWL-Quelle	2952	29.06.2009 11:48:21
1_FB12	---	AWL-Quelle	18281	29.06.2009 11:48:21
1_FB13	---	AWL-Quelle	47748	29.06.2009 11:48:21
1_HandAuto	---	AWL-Quelle	1765	29.06.2009 11:48:21
1_Hilfsmerker	---	AWL-Quelle	9867	29.06.2009 11:48:21
1_Init_Portalkran	---	AWL-Quelle	2179	29.06.2009 11:48:21
1_Initialisierung	---	AWL-Quelle	7113	29.06.2009 11:48:21
1_Kommission_ablegen	---	AWL-Quelle	6647	29.06.2009 11:48:21
1_Kommission_aufnehmen	---	AWL-Quelle	6545	29.06.2009 11:48:21
1_zu_kommission_fahren	---	AWL-Quelle	10555	29.06.2009 11:48:21
2_101	---	AWL-Quelle	1493	29.06.2009 11:48:21
2_102	---	AWL-Quelle	1493	29.06.2009 11:48:21
2_103	---	AWL-Quelle	1493	29.06.2009 11:48:21
2_104			1493	29.06.2009 11:48:21
2_105			1493	29.06.2009 11:48:21
2_106			1493	29.06.2009 11:48:21
2_107			1493	29.06.2009 11:48:21
2_108			1493	29.06.2009 11:48:21
2_111			1493	29.06.2009 11:48:21
2_112			1493	29.06.2009 11:48:21
2_151			3491	29.06.2009 11:48:21
2_152			3491	29.06.2009 11:48:21
2_209			1493	29.06.2009 11:48:21
2_DBController			1168	29.06.2009 11:48:21
2_DBInitialisierung			1226	29.06.2009 11:48:21
2_DBInitPk			509	29.06.2009 11:48:21
2_DBKommission_ablegen			1679	29.06.2009 11:48:21
2_DBKommission_aufnehmen			1565	29.06.2009 11:48:21
2_DBPKbeladen	---	AWL-Quelle	940	29.06.2009 11:48:21
2_DBPKentladen	---	AWL-Quelle	903	29.06.2009 11:48:21
2_DBRingpuffer	---	AWL-Quelle	607	29.06.2009 11:48:21
2_DBZukommission_fahren	---	AWL-Quelle	2556	29.06.2009 11:48:21
3_Fördersystem	---	AWL-Quelle	36464	29.06.2009 11:48:21
3_Hochregal	---	AWL-Quelle	20592	29.06.2009 11:48:21
3_Portalkran	---	AWL-Quelle	12080	29.06.2009 11:48:21
3_WinCCBerechnungen	---	AWL-Quelle	4783	29.06.2009 11:48:21
4_Hauptprogramm OB1	---	AWL-Quelle	1579	29.06.2009 11:48:21
1_Portalkran_beladen	---	SCL-Quelle	13979	29.06.2009 11:48:21
1_Portalkran_entladen	---	SCL-Quelle	10141	29.06.2009 11:48:21
1_Ringpuffer Drehteller	---	SCL-Quelle	3905	29.06.2009 11:48:21
1_Ringpuffer Kran	---	SCL-Quelle	2820	29.06.2009 11:48:21

Figure A.20: Application generation 9: Load the generated code into the Simatic® Manager

Bibliography

[ABM09] Ritu Arora, Purushotham Bangalore, and Marjan Mernik. Developing scientific applications using Generative Programming. In *Software Engineering for Computational Science and Engineering, 2009. SECSE '09. ICSE Workshop on*, pages 51–58, May 2009. (Cited on page 19.)

[ACA08] V. Alves, T. Camara, and C. Alves. Experiences with Mobile Games Product Line Development at Meantime. In *Proc. 12th International Software Product Line Conference SPLC '08*, pages 287–296, 2008. (Cited on page 16.)

[ACN$^+$08] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. FLiP: Managing Software Product Line Extraction and Reaction with Aspects. In *Proc. 12th International Software Product Line Conference SPLC '08*, pages 354–354, 8–12 Sept. 2008. (Cited on page 17.)

[BC05] Felix Bachmann and Paul C. Clements. Variability in Software Product Lines. Technical report, Software Engineering Institute, 2005. (Cited on page 14.)

[BDKB07] M. Bergert, C. Diedrich, J. Kiefer, and T. Bar. Automated PLC software generation based on standardized digital process information. In *Proc. ETFA Emerging Technologies & Factory Automation IEEE Conference*, pages 352–359, 25–28 Sept. 2007. (Cited on page 37.)

[Bec98] Beckhoff. Einführung in IEC-1131-3 Programmierung. Technical report, Beckhoff Industrie Elektronik, 1998. (Cited on page 31.)

[Beu08] D. Beuche. Modeling and Building Software Product Lines with Pure::Variants. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 358–358, Sept. 2008. (Cited on page 17.)

[BF01] M. Bonfe and C. Fantuzzi. Object-oriented approach to PLC software design for a manufacture machinery using IEC 61131-3 norm languages. In *Proc. IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, volume 2, pages 787–792, 8–12 July 2001. (Cited on page 36.)

[BFG$^+$01] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In van der Linden [vdL02b], pages 13–21. (Cited on page 14.)

[BG01] J. Bezivin and O. Gerbe. Towards a precise definition of the OMG/MDA framework. In *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, pages 273–280, Nov. 2001. (Cited on page 19.)

[Bol06] W. Bolton. *Programmable Logic Controllers, Fourth Edition.* Newnes, 2006. (Cited on pages 29, 30, 33 and 34.)

[Bos05] Jan Bosch. Software Product Families in Nokia. In Obbink and Pohl [OP05], pages 2–6. (Cited on page 16.)

[CJNM05] Paul C. Clements, Lawrence G. Jones, Linda M. Northrop, and John D. McGregor. Project Management in a Software Product Line Organization. *IEEE Software*, 22(5):54–62, 2005. (Cited on pages 10 and 11.)

[CRR09] Lan Cao, Balasubramaniam Ramesh, and Matti Rossi. Are Domain-Specific Models Easier to Maintain Than UML Models? *Software, IEEE*, 26(4):19–21, July-Aug. 2009. (Cited on page 21.)

[Cza04] Krzysztof Czarnecki. Overview of Generative Software Development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *UPP*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2004. (Cited on page 19.)

[Dav87] Stanley M. Davis. *Future Perfect.* Addison Wesly, 1987. (Cited on page 5.)

[DRGN07] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, and Thomas Neumayer. Integrated tool support for software product line engineering. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 533–534, New York, NY, USA, 2007. ACM. (Cited on pages 17, 26 and 79.)

[DSF07] O. Djebbi, C. Salinesi, and G. Fanmuy. Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 301–306, Oct. 2007. (Cited on pages 26 and 79.)

[Ecl08] Eclipse. openarchitectureware in Eclipse. http://www.eclipse.org/gmt/oaw/, 2008. (Cited on page 22.)

[Ecl09a] Eclipse. GEMS: Generic Eclipse Modeling System. http://www.eclipse.org/gmt/gems/, 2009. (Cited on page 22.)

[Ecl09b] Eclipse. GMT: Generative Modeling Technologies. http://www.eclipse.org/gmt/, 2009. (Cited on page 22.)

[EV05] Jacky Estublier and Germán Vega. Reuse and variability in large software applications. In Michel Wermelinger and Harald Gall, editors, *ESEC/SIGSOFT FSE*, pages 316–325. ACM, 2005. (Cited on pages 17 and 19.)

[fEA08] Information Technology for European Advancement. ITEA2 Homepage, 2008. (Cited on page 6.)

[Fre00] G. Frey. Automatic implementation of Petri net based control algorithms on PLC. In *Proc. American Control Conference the 2000*, volume 4, pages 2819–2823, 28–30 June 2000. (Cited on pages 36 and 37.)

[fSis09] Institute for Software integrated systems. GME: Generic Modeling Environment. http://www.isis.vanderbilt.edu/Projects/gme/, 2009. (Cited on page 21.)

[FW06] G. Frey and F. Wagner. A Toolbox for the Development of Logic Controllers using Petri Nets. In *Proc. 8th International Workshop on Discrete Event Systems*, pages 473–474, 10–12 July 2006. (Cited on page 36.)

[GH09] Weber Jörg Günther and Wolf Hannes. Lagerverwaltung. Bachelor-Thesis, Graz University of Technology, 2009. (Cited on pages 39, 53, 54 and 85.)

[GHE08] Christian Gerber, Hans-Michael Hanisch, and Sven Ebbinghaus. From IEC 61131 to IEC 61499 for distributed systems: a case study. *EURASIP J. Embedded Syst.*, 2008(2):1–8, 2008. (Cited on pages 35 and 36.)

[Gro09] Object Management Group. Object Management Group website. http://www.omg.org/, 2009. (Cited on page 19.)

[GVDV08] Gerardo de Geest, Sander Vermolen, Arie van Deursen, and Eelco Visser. Generating Version Convertors for Domain-Specific Languages. *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 197–201, Oct. 2008. (Cited on page 21.)

[HGHV07] M. Hirsch, C. Gerber, H.-M. Hanisch, and V. Vyatkin. Design and Implementation of Heterogeneous Distributed Controllers According to the IEC 61499 Standard - A Case Study. In *Proc. 5th IEEE International Conference on Industrial Informatics*, volume 2, pages 829–834, 2007. (Cited on page 36.)

[Hol08] Inc Holobloc. Resources for the New Generation of Automation and Control. http://www.holobloc.com/, 2008. (Cited on page 35.)

[Hon09] Honeywell. DOME: Domain Modeling Environment . http://www.htc.honeywell.com/dome/, 2009. (Cited on page 22.)

[HP07] Kwan Hee Han and Jun Woo Park. Development of Object-Oriented Modeling Tool for the Design of Industrial Control Logic. In *Proc. 5th ACIS International Conference on Software Engineering Research, Management & Applications SERA 2007*, pages 353–358, 20–22 Aug. 2007. (Cited on page 36.)

[Ins] Software Engineering Institute. Product Line Hall of Fame. (Cited on page 16.)

[Ins08] Software Engineering Institute. SPLE. http://www.sei.cmu.edu/productlines/index.html, 2008. (Cited on pages 6 and 8.)

[Jaa02] A. Jaaksi. Developing mobile browsers in a product line. 19(4):73–80, 2002. (Cited on page 16.)

[KAG$^+$07] Uirá Kulesza, Vander Alves, Alessandro Garcia, Alberto Costa Neto, Elder Cirilo1, Carlos J. P. de Lucena, and Paulo Borba. Mapping Features to Aspects: A Model-Based Generative Approach7. In *Early Aspects: Current Challenges and Future Directions*, Lecture Notes in Computer Science, pages 155–174. Springer Berlin / Heidelberg, 2007. (Cited on page 13.)

[Kas08] Wolfgang Kastner. Speicherprogrammierbare Steuerungen. Technical report, Lecture notes, Technical university Vienna, 2008. (Cited on pages 29 and 30.)

[KMML07] T. Kosar, M. Mernik, and P.E. Martinez Lopez. Experiences on DSL Tools for Visual Studio. *Information Technology Interfaces, 2007. ITI 2007. 29th International Conference on*, pages 753–758, June 2007. (Cited on page 21.)

[KP09] Steven Kelly and Risto Pohjonen. Worst Practices for Domain-Specific Modeling. *Software, IEEE*, 26(4):22–29, July-Aug. 2009. (Cited on page 18.)

[Kru08] C.W. Krueger. The BigLever Software Gears Unified Software Product Line Engineering Framework. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 353–353, Sept. 2008. (Cited on page 17.)

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain Specific Modeling Enabling full code generation.* WILEY Interscience, 2008. (Cited on pages 17, 19, 20, 21 and 83.)

[LCP$^+$00] Bass L., Clements, P., Donohoe, P., McGregor, J., and Northrop L. Fourth Product Line Practice Workshop Report. Technical Report CMU/SEI-2000-TR-002 (ESC-TR-2000-002), Software Engineering Institute. Carnegie Mellon University, 2000. (Cited on pages 8, 26 and 79.)

[Lei09] Andrea Leitner. A software product line for a business process oriented IT landscape. Master's thesis, Graz University of Technology, 2009. (Cited on pages 26 and 79.)

[LP07] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proc. 11th International Software Product Line Conference SPLC 2007*, pages 151–162, 2007. (Cited on page 15.)

[MED08] MEDEIA. Medeia. http://www.medeia.eu/, 2008. (Cited on pages 37 and 38.)

[Met08] MetaCase. Domain-Specific modeling with MetaEdit+, 2008. (Cited on pages 23 and 25.)

[MGM05] G. Music, D. Gradisar, and D. Matko. IEC 61131-3 Compliant Control Code Generation from Discrete Event Models. In *Proc. IEEE International Symposium on Mediterrean Conference on Control and Automation Intelligent Control*, pages 346–351, 2005. (Cited on page 31.)

[Mic08a] Microsoft. Domain-Specific Language Tools . http://msdn.microsoft.com/en-us/library/bb126235(VS.80).aspx, 2008. (Cited on page 21.)

[Mic08b] Microsoft. Domain-Specific Language Tools Changes. http://msdn.microsoft.com/en-gb/library/bb932387.aspx, 2008. (Cited on page 21.)

[Nor02] L.M. Northrop. SEI's software product line tenets. 19(4):32–40, 2002. (Cited on pages 9, 10, 11 and 12.)

[Nor04] Robert L. Nord, editor. *Software Product Lines, Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004, Proceedings*, volume 3154 of *Lecture Notes in Computer Science*. Springer, 2004. (Cited on page 105.)

[Nor07] Linda Northrop. Fourth Product Line Practice Workshop Report. Technical report, Software Engineering Institute. Carnegie Mellon University, 2007. (Cited on pages 2, 9, 13 and 16.)

[NTB$^+$08] Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. Applying visualisation techniques in software product lines. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visuallization*, pages 175–184, New York, NY, USA, 2008. ACM. (Cited on page 15.)

[OP05] J. Henk Obbink and Klaus Pohl, editors. *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*. Springer, 2005. (Cited on pages 100 and 105.)

[ope08] PLC open. PLC open. http://www.plcopen.org, 2008. (Cited on page 31.)

[ope09] openArchitectureWare.org. openarchitectureware. http://www.openarchitectureware.org/, 2009. (Cited on page 22.)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering*. Springer, 2005. (Cited on pages 5, 6, 7, 8, 9, 10, 11, 12, 14 and 15.)

[PK07] Risto Pohjonen and Steven Kelly. Interactive Television Applications using MetaEdit+. Model-Driven Development Tool Implementers Forum (MDDTIF07), 2007. (Cited on pages 24 and 25.)

[PM06] Klaus Pohl and Andreas Metzger. Variability management in software product line engineering. pages 1049–1050, 2006. (Cited on page 14.)

[RBKS07] M. Regensburger, C. Buckl, A. Knoll, and G. Schrott. Model Based Development of Safety-Critical Systems Using Template Based Code Generation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 89–92, Dec. 2007. (Cited on page 22.)

[Sch07] Gernot Schmölzer. *A Model-based Software Product Line Architecture for Data-intensive Systems*. PhD thesis, Technical university Graz, 2007. (Cited on pages 5, 8, 10 and 12.)

[Sen07] Paulus Sentosa. Generation of Text Editors for Custom Domain Specific Language on the Eclipse Platform. Master's thesis, Hamburg University of Technology, 2007. (Cited on pages 22 and 23.)

[Siv08a] S. Sivonen. DSML for Developing Repository-Based Eclipse Plug-Ins. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 356–356, Sept. 2008. (Cited on page 23.)

[Siv08b] Sanna Sivonen. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins. *VTT Publications 680 VTT, Espoo, Finland.*, 2008. (Cited on pages 21, 23, 25 and 26.)

[SMBU07] D. Sellier, M. Mannion, G. Benguria, and G. Urchegui. Introducing Software Product Line Engineering for Metal Processing Lines in a Small to Medium Enterprise. In *Proc. 11th International Software Product Line Conference SPLC 2007*, pages 54–62, 2007. (Cited on pages 16 and 36.)

[SO96] Dieter Spath and Ulf Osmers. Virtual Reality - An Approach to Improve the Generation of Fault-Free Software for Programmable Logic Controllers (PLC). In *ICECCS*, pages 43–46, 1996. (Cited on page 37.)

[STB$^+$04] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *SPLC*, pages 34–50, 2004. (Cited on page 16.)

[TMKG07] C. Tischer, A. Muller, M. Ketterer, and L. Geyer. Why does it take that long? Establishing Product Lines in the Automotive Domain. In *Proc. 11th International Software Product Line Conference SPLC 2007*, pages 269–274, 2007. (Cited on page 16.)

[Tol06] Juha-Pekka Tolvanen. MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 690–691, New York, NY, USA, 2006. ACM. (Cited on page 25.)

[TPB$^+$08] Trinidad, P., Benavides, D., Ruiz-Cortes, A., Segura, S., and A. Jimenez. FAMA Framework. In *Proc. 12th International Software Product Line Conference SPLC '08*, pages 359–359, 2008. (Cited on page 17.)

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. (Cited on page 18.)

[vdL02a] F. van der Linden. Software product families in Europe: the Esaps & Cafe projects. 19(4):41–49, 2002. (Cited on page 6.)

[vdL02b] Frank van der Linden, editor. *Software Product-Family Engineering, 4th International Workshop, PFE 2001, Bilbao, Spain, October 3-5, 2001, Revised Papers*, volume 2290 of *Lecture Notes in Computer Science*. Springer, 2002. (Cited on page 99.)

[vdLBK$^+$04] Frank van der Linden, Jan Bosch, Erik Kamsties, Kari Känsälä, and J. Henk Obbink. Software Product Family Evaluation. In Nord [Nor04], pages 110–129. (Cited on page 6.)

[vdW99] E. van der Wal. Introduction into IEC 1131-3 and PLCopen. *The Application of IEC 61131 to Industrial Control: Improve Your Bottom Line Through High Value Industrial Control Systems (Ref. No. 1999/076), IEE Colloquium on*, pages 2/1–2/8, 1999. (Cited on pages 31, 32 and 33.)

[Ver96] A.A. Verwer. Teaching open standards for PLC programming. *Mechatronics in Education: Delivery of a New Engineering Discipline into the Workplace, IEE Colloquium on*, pages 8/1–8/7, 26 Mar 1996. (Cited on pages 29, 31 and 33.)

[WCKK06] D.M. Weiss, P.C. Clements, Kyo Kang, and C. Krueger. Software Product Line Hall of Fame. *Software Product Line Conference, 2006 10th International*, pages 237–237, Aug. 2006. (Cited on page 16.)

[Wei05] David M. Weiss. Next Generation Software Product Line Engineering. In Obbink and Pohl [OP05], page 1. (Cited on page 6.)

[Wei08] D.M. Weiss. The Product Line Hall of Fame. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 395–395, Sept. 2008. (Cited on page 16.)

[YF06] M.B.Y. Younis and G. Frey. UML-based Approach for the Re-Engineering of PLC Programs. In *Proc. IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 3691–3696, 6–10 Nov. 2006. (Cited on page 37.)

[ZLRK04] Yong Zhou, Thies Lauk-Reineke, and Christian König. Das System SPS. Technical report, Lecture notes, Technical university Braunschweig, May 2004. (Cited on page 29.)