Pure

Syddansk Universitet

Software Engineering Environment for Component-based Design of Embedded Software

Guo, Yu

Publication date:
2010

Document Version
Final published version

Link to publication

Citation for pulished version (APA):
Guo, Y. (2010). Software Engineering Environment for Component-based Design of Embedded Software.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. jan.. 2017

A Dissertation for the Degree of Doctor of Philosophy

Software Engineering Environment for Component-based Design of Embedded Software

by

Yu Guo

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Soenderborg, Denmark

2009

© 2009
Yu Guo
All Rights Reserved

Acknowledgements

First of all, I wish to thank my supervisor, Prof. Christo Angelov, this thesis would not have been possible without his expert guidance. His comments and suggestions were always extremely perceptive, helpful, and appropriate. And, I am grateful for his inspiring and encouraging me to pursue a career in embedded software research, and for enabling me to do so.

I am thankful for suggestions, comments, and contributions from my colleagues at the Mads Clausen Institute for Product Innovation. I am very grateful to Associate Prof. Nicolae Marian for his scientific advice and knowledge. My enormous debt of gratitude can hardly be repaid to my friend, Assistant Prof. Krzysztof Sierszecki, who provided many insightful discussions and suggestions throughout the Ph.D. project. I also thank the wonderful staff in the MCI for always being so helpful and friendly. For this research, implementation was essential. Many people from MCI helped with this, for which I would like to thank them wholeheartedly.

Many people participating in the MoDES project from Aalborg University helped me in various ways. I am especially grateful to Prof. Anders P. Ravn, and Associate Prof. Arne Skou for all that they have taught me during our project meetings and workshops. I am particularly thankful to my friend, István Knoll, who provided a lot of very useful input and ideas to this research work, during our collaboration.

I would like to deeply thank Nicholas Gunder and Torben Hoffmann from Motorola A/S, for their valuable input to this work from the industrial point of view.

Thanks to the Danish Council for Strategic Research for funding this project.

My sincere gratitude goes to my parents for their love, support and patience. I owe special thanks to my wife Dong, for her love, care and understanding. It would have been much more difficult for me to accomplish this work without her support.

Yu Guo
Viborg, September 2009

Abstract

The extensive and ever increasing use of embedded real-time control systems poses a serious challenge to software developers, in view of conflicting requirements concerning time to market, development costs, safety and dependability. It is hard to satisfy these requirements using conventional software technology, which is largely based on informal design methods and manual coding techniques. That is why it is necessary to develop new design methods and tools that will eventually help improve existing practices.

These considerations have motivated the development of the framework – *Component-Based Design of Software for Distributed Embedded Systems* (COMDES) and the related software design methodology, in an attempt to provide a solution to the problems formulated above. The essence of this methodology is the adoption of formal models used to systematically develop embedded software applications that are correct by construction, and ultimately – configure applications from prefabricated reusable components, with the support of appropriate tools. It combines component-based design with a model-driven software development approach, which reduces development time through design automation, enhances software quality by deriving the implementation from models, and enables software specification on a more abstract level, using an adequate modelling language.

The COMDES framework provides a domain-specific modelling language that can be used to specify relevant aspects of system structure and behaviour in the domain of distributed embedded control systems with hard real-time constraints. The framework has been further extended with a number of meta-models that have been derived from the formal specification of domain-specific design models. The meta-models can be used to represent predefined component models as well as application models in a computer-aided software engineering environment. Furthermore, component models have been realized following carefully developed design patterns, which provide for an efficient and reusable implementation. The components have been ultimately implemented as prefabricated executable objects that can be linked together into an executable application.

The development of embedded software using the COMDES framework is supported by the associated integrated engineering environment consisting of a number of tools, which support basic functionalities, such as system modelling, validation, and executable code generation for specific hardware platforms.

Developing such an environment and the associated tools is a highly complex engineering task. Therefore, this thesis has investigated key design issues and analysed existing platforms supporting model-driven software development that can be used as a foundation of a component-based software engineering environment, in order to develop a viable toolset for the COMDES framework. As a result, the Eclipse platform has been chosen to implement the software engineering environment, due to its strong support for tool development by integrated model-driven development frameworks such as the Eclipse Modelling Framework, Graphical Modelling Framework, etc.

Based on theoretical investigations and hands-on experiences, this thesis focuses on the practical aspects of building the COMDES software engineering environment on the Eclipse platform, with an emphasis on technologies and tools concerning both model-driven and component-based development of embedded software.

Preliminary experiments have indicated that the COMDES framework and its development toolset may offer a better approach to embedded software development than previous technologies, and in particular – speed up the development cycle and improve the quality of software for distributed real-time embedded control systems.

Abstrakt

Den voksende brug af indlejrede realtids kontrolsystemer stiller store udfordringer til softwareudviklere set i lyset af modstridende krav, som time-to-market, udviklingsomkostninger, sikkerhed og pålidelighed. Det er svært at møde disse krav ved brug af konventionel softwareudviklings-teknologi, som stort set er baseret på uformelle designmetoder og manuel kodning. Det er derfor nødvendigt at udvikle nye designmetoder og -værktøjer, som på sigt vil kunne være med til at forbedre den eksisterende praksis på området.

Disse overvejelser har motiveret til udvikling af et værktøj til komponentbaseret design af software for distribuerede indlejrede systemer – Component-based Design of Software for Distributed Embedded Systems (COMDES) – og den relaterede software design metode i forsøget på at levere en løsning til problemstillingerne skitseret ovenfor. Kernen i denne teknologi er anvendelsen af formelle modeller. Ved at bruge disse modeller, kan indlejrede softwareapplikationer udvikles systematisk, konstrueres korrekt ved implementationen og endeligt, at det bliver muligt at konfigurere applikationer fra færdigbyggede og genbrugelige komponenter, under anvendelse af passende værktøjer. De kombinerer komponentbaseret design med den modeldrevne softwareudviklingsmetode, som mindsker udviklingstiden ved automatisering af visse designprocesser, forbedrer softwarens kvalitet ved automatisk at generere implementationen fra modeller og den muliggør specifikation af software på et mere abstrakt niveau, ved at bruge et passende modelleringssprog.

I COMDES indgår et domæne-specifikt modelleringssprog som kan bruges til at specificere relevante aspekter af systemets struktur og funktionalitet indenfor domænet distribuerede indlejrede kontrolsystemer med hårde realtids krav. Frameworket er yderligere udvidet med et antal metamodeller, udledt af den formelle specifikation af domænespecifikke designmodeller. Metamodellen kan bruges til at repræsentere såvel foruddefinerede komponentmodeller som applikationsmodeller for computerunderstøttet softwareudvikling. Derudover blev komponentmodeller implementeret, der følger præcist udviklede designmønstre, som muliggør en effektiv og genbrugelig implementation. Endelig blev komponen-

terne implementeret som præfabrikerede eksekverbare objekter, der kan afvikles og linkes sammen i en eksekverbar applikation.

Udvikling af indlejret software ved brug af COMDES frameworket er understøttet af det tilhørende udviklingsmiljø (IDE), som dækker over et antal værktøjer der understøtter basale funktioner som systemmodellering, validering, kodegenerering og kompillering for specifikke hardware-platforme.

Konstruktion af sådanne udviklingsmiljøer og de tilhørende værktøjer er en meget kompleks udviklingsopgave. I denne afhandling bliver derfor designspørgsmål af central vigtighed undersøgt og eksisterende platforme, der understøtter modeldrevet softwareudvikling, der kan bruges som fundament til et komponentbaseret softwareudviklings-værktøj, analyseret med henblik på udvikling af et anvendeligt sæt værktøjer for COMDES.

Eclipse blev valgt som platform for implementeringen af softwareudviklingsværktøjet, da den med det integrerede modeldrevne udviklingsframework, meget godt understøtter udvikling af værktøj som f.eks. Eclipse Modeling Framework, Graphical Modeling Framework, osv.

Baseret på teoretiske undersøgelser og praktisk erfaring, fokuserer denne afhandling på praktiske aspekter vedrørende udviklingen af COMDES softwareudviklingsværktøjet på Eclipse platformen og med vægt på teknologier og værktøjer til både modeldrevet og komponentbaseret udvikling af indlejret software.

Foreløbige eksperimenter indikerer, at COMDES og det tilhørende udviklingsværktøj muliggør en bedre tilgang til udvikling af indlejret software end tidligere teknologier, og mere konkret, den kan gøre udviklingsprocessen hurtigere og samtidig bidrage til en højere produktkvalitet ved distribuerede indlejrede realtidskontrolsystemer.

List of Publications Developed During the PhD Project

Publications in Refereed Conference Proceedings

- Kebin Zeng, Yu Guo and Christo Angelov. Graphical Model Debugger Framework for Embedded Systems. Accepted for presentation to *the 13th DATE Conference and Exhibition: Design, Automation and Test in Europe (DATE2010)*, Dresden, Germany, March 8-12, 2010.

- Christo Angelov, Krzysztof Sierszecki and Yu Guo. Formal Design Models for Distributed Embedded Control Systems. In *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)*, Denver, Colorado, USA, October 2009.

- Yu Guo, Krzysztof Sierszecki and Christo Angelov. Model-Driven Development of Domain-Specific Applications: Tool Support. In *Proceedings of 11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering (SPLST 2009 and NW-MODE 2009)*, Tampere, Finland, August 2009.

- Yu Guo, Krzysztof Sierszecki, and Christo Angelov. COMDES Development Toolset. In *Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008)*, pages 233-238, Malaga, Spain, September 2008.

- Christo Angelov, Xu Ke, Yu Guo, and Krzysztof Sierszecki. Reconfigurable State Machine Components for Embedded Applications. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2008)*, pages 51-58, Parma, Italy, September 2008.

- Yu Guo, Krzysztof Sierszecki, and Christo Angelov. A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008)*, pages 1315-1320, Turku, Finland, July/August 2008.

- Yu Guo, Feng Zhou, Nicolae Marian, and Cristo Angelov. Hardware-in-the-Loop Simulation of Component-Based Embedded Systems. In *Proceedings of the 8th International Workshop on Research and Education in Mechatronics (REM 2007)*, Tallinn, Estonia, June 2007.

- Nicolae Marian and Yu Guo. Model-Based Design of Embedded Software. In *Proceedings of the 7th International Workshop on Research and Education in Mechatronics (REM 2006)*, Stockholm, Sweden, June 2006.

Publications in Conference Proceedings

- Yu Guo, Torben Hoffmann, and Nicholas Gunder. Autocoding State Machine in Erlang. In *Proceedings of the 14th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2008.

Other Publications in Refereed Conference Proceedings

- Zdravko Karakehayov and Yu Guo. Parallel Embedded Systems: Where Real-Time and Low-Power Meet. In *Proceedings of the ISCA's 21st International Conference on Parallel and Distributed Computing and Communications Systems (PDCCS 2008)*, New Orleans, Louisiana, USA, September 2008.

Internal Project Reports

- Yu Guo. *Production Cell case study*, Real-Time Systems Project Report, Mads Clausen Institute, University of Southern Denmark, December 2007.

- Xu Ke and Yu Guo. *Case Studies in Component-based Design for Distributed Embedded Systems*, Distributed Embedded Systems Project Report, Mads Clausen Institute, University of Southern Denmark, June 2006.

Contents

List of Figures

In the history of the human race, the making and usage of tools mark the spot where human is distinct from animal. Benjamin Franklin is credited with first defining the human being as a "tool-making animal." The artificial tools were made of different materials – from stone, bronze, iron, etc., to artificial material, composite material, etc., and even materials existing in a virtual form – program code. Meanwhile, the purpose of using tools has shifted from food gathering and food production to tool production, machine production and software production. After spending hundreds, and even thousands of years in exploration and practice, we have been stepping into the age of making tools that produce software to control machines used in various human activities.

This thesis tells a story of making such tools for the purpose of software production, as a tiny drop in the ocean of human history ...

Chapter 1

Introduction

1.1 Distributed real-time embedded control systems

As human beings, we not only use traditional tools to help us with daily activities, but also try to develop new kinds of tools, machines and intelligent devices that will make it possible to reduce and ultimately, replace manual work. Nowadays, the majority of tools, machines and smart devices are controlled by so-called embedded systems.

Embedded systems span all aspects of modern life and there are many examples of their use, such as telecommunications applications, consumer electronics, household appliances, transportation systems, medical equipments, and so on. Such applications employ increasingly embedded systems to provide flexibility, efficiency and advanced features.

In addition to hardware such as microprocessor, memory, I/O ports, etc., the implementation of embedded system's functionality relies predominantly on software – embedded software is quickly gaining importance as embedded applications grow in numbers and complexity, because software can reduce the cost of hardware when it comes to mass production. Embedded software is usually written for special-purpose hardware, sometimes using a small real-time operating system such as VxWorks, Nucleus, eCos, and so on. As a consequence, the operation of embedded systems is not solely dependent of their hardware, but also the embedded software. The combination of these two factors determines how an embedded system behaves.

As one of important characteristics, embedded system industry is "embedded" into other industries, as embedded systems are designed to do some specific task for specific industrial applications, rather than to be a general-purpose computer

for multiple tasks. Therefore, the industry or application domain that the embedded system is tailored for plays a key role during the development of embedded software.

Most of the industries deploy embedded systems for the purpose of control, and most of the control systems found in nature are feedback control systems as defined in Control Theory. Such systems sense the environment to determine the desired output (reference value) and the actual output of an object of control, use the difference between the two values to generate a control signal, and act on the object of control using the generated control signal (see Fig. 1.1).

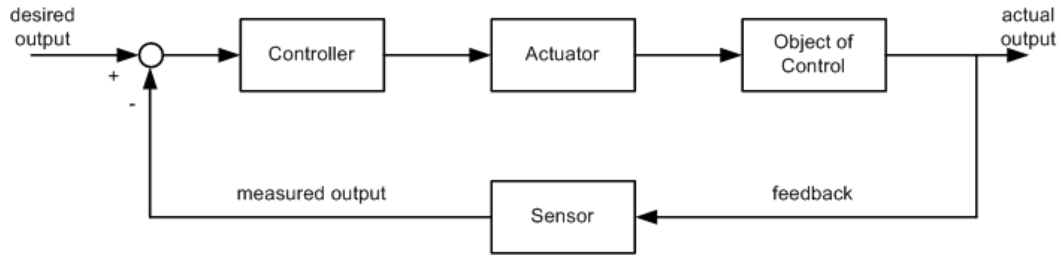


Figure 1.1: A control system

As the prices of modern small microcontrollers are going cheap, it's very common to implement control systems as embedded systems. Embedded control systems are found in cars, airplanes and houses, information and communication devices such as digital TV and mobile phones, and autonomous systems such as service or edutainment robots.

Embedded systems are not always standalone devices. Many embedded systems consist of small, computerized parts, meaning that a central control computer is substituted by several embedded systems (nodes) with local signal-processing and control functions. In such a system, the microcontrollers of various nodes do not share memory. Instead, nodes communicate with one another through communication networks, such as high-speed buses or telephone lines. Such a system is called a distributed embedded system, as the overall control task is distributed over a network of nodes and each of them performs only a part of the task. The benefits of introducing distribution of embedded systems are obvious: better performance through parallel execution of programs; increased reliability by using redundant nodes; less restrictive localization of nodes by the introduction of networking; easier modification or enhancement by addition or replacement of nodes.

Many embedded control systems are reactive systems that must respond quickly to events taking place in the external environment. Such systems are subject to

a "real-time constraint" – i.e., the control signal generated by the system must be provided before a specific time interval – usually called *deadline*, after receiving a stimulation event. The completion of an operation after its deadline may lead to a critical failure of the entire system. A real-time embedded control system is usually used in mission-critical applications involving surroundings or human lives, which are of growing importance for modern society. The operation of such an embedded system depends not only upon its logical correctness but also upon the time in which it is performed. As a consequence, embedded software for such a real-time system is quite complicated.

Things are becoming even more complicated when combining a distributed system and a real-time system together in a control application. With the continuing advances in the computational power, and reductions in the costs of high-performance processing and memory elements, more and more devices such as intelligent sensors, actuators with substantial processing capabilities are used in a range of distributed embedded real-time control applications. The networked embedded systems must run autonomously, while simultaneously communicating and responding to unanticipated combinations of events at run-time.

As distributed real-time embedded control applications grow more complex, so is the embedded computing software! As an integral part of embedded systems, the embedded software systems are characterized by functional and non-functional aspects. Embedded system developers have to consider requirements of both aspects, together with physical constraints. Functional requirements specify the expected functionalities or features, while non-functional requirements specify mainly performance (such as timing, memory etc). For the same functional requirements, non-functional properties can vary depending on a large number of factors and choices, including the overall system architecture and the characteristics of the underlying platform [1].

Timing, as one of the non-functional aspects, is of utmost significance to the real-rime application. Hence, it is not only necessary to ensure that the functional behaviour of the system is correct but also – its timing requirements are met. However, developing software for complex, distributed real-time control systems has become more difficult, because of the ever increasing complexity of applications and the gap between software development productivity and complex hardware availability [2]. Consequently, software development cost and time are quite high.

To sum up, according to Thomas A. Henzinger, "an embedded system is an engineering artefact involving computation that is subject to physical constraints. The physical constraints arise through two kinds of interactions of computational processes with the physical world: (1) reaction to a physical environment, and (2)

execution on a physical platform. [...] embedded systems consist of hardware, software, and an environment" [1]. The development of embedded systems is getting increasingly complex as they tend to become more and more distributed and real-time in nature. As a result, according to Colin Atkinson, "the methods and technologies that have traditionally been used to develop embedded systems are starting to reach the limits of their scalability" [3].

Today's embedded engineer must be familiar with a wide range of technologies for describing both hardware and software. Translating from the requirements of a problem, specific to an industry, to a solution requires a deep understanding of the technologies that comprise an appropriate solution. Furthermore, end-users expect the result to be fast, available, scalable, etc. The expectations can be met by introducing adequate software engineering approaches into embedded software development.

Several methods exist that can be used to improve software development productivity and thus bridge the gap between the present levels of hardware and software technology. One of them, component-based development, is quite attractive in the domain of embedded systems, as it advocates the idea of quickly creating software by assembling applications from reusable software components.

1.2 Component-based development of embedded software

The idea of *Component-Based Development* (CBD) is in fact quite old. By saying old, we do not mean it is as old as decades ago when the notion of component was introduced into the area of software programming. By saying old, we mean thousands of years old!

At the end of the Warring States Period of China (5th century BC to 221 BC), the state Qin, as one of the Seven Warring States, easily defeated the other six major states as well as a dozen of minor states in a couple of years, and effectively unified China in 221 BC. According to records from historical books (e.g. Strategies of the Warring States), her army was not large enough to defeat all armies from the six states. One of the possible reasons why Qin was transformed from a backward state into one that surpassed the other six states was found out lately by archaeologists – the weapons that the army was equipped with.

Some of their weapons as well as parts of weapons were discovered by archaeologists at different ancient battlefields where her army fought against their opponents. The weapons include crossbows, long swords, dagger-axes, halberds, etc. To people's surprise, the pieces of these weapons (see Fig. 1.2 for an ex-

ample), dating from that period and excavated from different areas, are exactly identical to each other and interchangeable, which makes it possible for individual soldiers to replace broken parts quickly and easily with new ones.

Figure 1.2: An interchangeable trigger from a crossbow

The fact tells us that tools for fighting had already been "componentized" around thousands of years ago! Such an approach helped the army react as fast as possible to changes, and maintained it as the most dominant force in China.

Based on experience, a human being really gets benefits by breaking a complicated product down into smaller pieces that have well-defined interfaces. Such activity makes the product flexible, because it is possible to easily replace a component when the product requirements change. Reuse of components in new products accelerates development and lowers cost. Furthermore, sharing a common interface between components helps multiple organizations collaborate more easily.

A product can be assembled from smaller components. Such a practice has been repeated by our great ancestors for thousands of years in various already mature development activities. As a relatively new industry commenced in the middle of the 20th century, what people are searching and researching now for software development is not quite different from what others have done before. Researchers in this area are trying to break complicated software down into interchangeable pieces (components), in order to improve productivity. Before the con-

cept of "Software Componentization" was introduced, most of software developers already designed and wrote software products out of multiple pieces of previously written code. They just needed to replace those pieces with new implementations for another product. Modular programming as a software design technique was introduced in the late 1970s, by Niklaus Wirth. This technique, which is still widely used in practice, increases the extent to which software is composed from separate parts, called modules, thus improving maintainability. Douglas McIlroy is considered as the person who formulated the idea that software should be componentized, i.e. built from prefabricated components, at the NATO conference on software engineering in Garmisch, Germany, 1968, with a publication entitled *Mass Produced Software Components* [4].

Unfortunately, according to Clemens Szyperski, "components are well established in all other engineering disciplines, but until recently were unsuccessful in the world of software" [5]. Software products, unlike traditional products, are hard to maintain. Software producers have found that most of their time is spent on maintaining the software rather than developing it. As one of the reasons, software products are subject to changes. Once a requirement has been changed, it usually takes producers long time to modify the product, because developers have to gather all knowledge of designs and code in order to decide where to modify. Moreover, after the modification, they will have to spend time to ensure the changed parts do not influence other parts of the system. In case of any difficulty of reaction to changes, their opponents will have enough time to kick them out of the market.

The occurrence of CBD seems promising to give a solution of the problem. "The objective of CBD," according to Katharine Whitehead, "is to improve the software development process by assembling new software from prebuilt software components rather than developing it from scratch" [6].

Katharine Whitehead, in her book "Component-based Development: principles and planning for business systems", gives a definition of software component:

> "A software component is a separable piece of executable software that makes sense as a unit, and can interoperate with other components, within some supporting environment. The component is accessible only via its interfaces and is capable of use as-is, after any necessary installation and configuration procedures have been carried out. In order for it to be combined with other components, it must be possible to obtain details of its interface" [6].

She addresses "separability" as the first principle of defining components. It must be possible to separate a component from its context and use it in another

context. She considers a software component as a black box with invisible internal implementation, thus any access to the component has to be through its interface. The implementation of a component can be changed without affecting its use by other components, provided that the new component offers the same functionality as previously.

She specifically emphasizes a software component as being a prebuilt entity, rather than a source unit that might be assembled into a program and then compiled; consequently, components are not assembled before compilation but afterwards. The assembly of components then involves a configuration process.

Another definition given by Clemens Szyperski in his book "Component Software: Beyond Object-Oriented Programming" presents an idea that is close to the statement discussed above:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies, which can be deployed independently and is subject to the third party composition" [5].

Both of the definitions implicitly designate the concept of reuse as an important characteristic property of a software component. A software component should be designed and implemented so that it can be reused when composed with other components. In other words, it needs to be well separated from its environment and other components. An implementation of a software component is invisible behind its interface that defines the component's access point. Users of the component know no details beyond the interface and its specification. The definition of the interface is of key importance to the definition of a component. The component can be reused without relying on anything but their interfaces and specifications.

Once individual software components have been constructed, they should be able to cooperate usefully. A framework copes with how components can be composed into a large system. A descriptive definition of component framework is given by Clemens Szyperski as follows:

> "A component framework is a software entity that supports components conforming to certain standards and allows instances of these components to be 'plugged' into the component framework" [5].

This definition points that an important role of a framework is to regulate the interaction between component instances.

Nowadays, there are a number of mature and sophisticated technologies including components and frameworks developed for various purposes, such as Object Management Group's CORBA, Sun's Enterprise Java Beans and Microsoft's COM, which are major component players in the world of Web and Desktop applications. Moreover, plug-in technique for the Firefox web browser, and the Eclipse development platform, etc., also illustrate good practices of using components.

However, the benefits of using CBD, such as reductions in development time and cost, are much more difficult to realize in embedded systems development, as an important design challenge for such complex real-time embedded systems is to satisfy non-functional requirements, while ensuring proper functional behaviour, when building new applications from existing components. Development of embedded software meeting such combined requirements is time- and manpower consuming, especially when the requirements are changing from time to time.

The creation of component-based software for real-time embedded control applications is now high on agenda of the research community. For one example, in the area of industrial automation, the IEC 61131-3 [7] standard defined by the International Electrotechnical Commission has received worldwide industrial acceptance for development of programmable logic controllers (PLCs).

Under the IEC 61131-3 standard, a PLC application is divided into a number of software components that are written in any of the languages proposed in the standard. For instance, *Function Block Diagram* is used for specifying signal processing and data flows in application programs composed of function blocks. Function block diagrams can be combined with *sequential function charts* to express the interaction of control algorithms and application logic. The entire software required to solve a particular control problem can be formulated as a configuration that is specific to a particular type of control system, including the arrangement of the hardware, i.e. processing resources, memory addresses for I/O channels, etc.

IEC 61131-3 is the first real endeavour to standardize programming languages for industrial automation. Compared with traditional programming systems, it appears to be a major step forward. However, it has also certain limitations, e.g. strictly periodic execution of application programs. This has stimulated the development of the follow-on standard IEC 61499, which employs event-driven function blocks and function block networks.

Koala [8][9] is another example of a component-based technology, which has been developed and used by Philips. It is tailored for development of software in consumer electronics such as televisions, video recorders, etc., which are often resource constrained since they use cheap hardware to keep development costs

low. A Koala component is a piece of code that can interact with its environment through explicit interfaces only. As a consequence, a basic Koala component has no implicit dependencies on other Koala components. Also, a Koala component is said to be hardware-independent, since hardware dependency is encapsulated in particular components. Interaction with the environment, including the underlying hardware-dependent services is exclusively via interfaces.

A system using the Koala components is built on top of a small real-time kernel with preemptive scheduling, which separates high frequency from low frequency tasks. It is possible to specify the order of tasks using precedence relations and mutual exclusion. But, there is no support for timing properties.

The technology has been proven suitable for consumer electronics software. However, the weak point of Koala lies in the lack of tools supporting efficient development on a large scale. Koala developers must conform to rules that can be violated, unless checked automatically [10].

There are also a number of other component technologies for embedded systems [10][11][12] such as Prediction-Enabled Component Technology (PECT) [13] developed by Carnegie Mellon University, Rubus developed by Arcticus Systems AB [14], PECOS [15] developed as a collaborative project between industrial and research partners, AUTOFOCUS [16][17][18] developed at the Technical University of Munich, and so on. However, there is currently no widely accepted component technology for embedded software development, due to the fact that requirements vary considerably in different application domains.

Research on software components and frameworks for this kind of system has been carried out in the Mads Clausen Institute for Product Innovation, University of Southern Denmark [19][20]. It addresses the problems of component-based design including systematic definition of software components and their composition, such that components can be used to adequately specify the structure and behaviour (i.e. functional and timing behaviour) of an embedded software system.

To sum up, as with componentization activities humans did thousands of years ago, the contemporary organizations who first master the art of component-based embedded systems development will be able to react to changes of requirements quickly, because the approach can improve productivity concerning software development, by breaking down a large problem into sub-problems and solve those separately. Software components can be reused, thus it is not necessary to reimplement functionalities in a different application within the same domain.

Component technology indisputably provides us with a better handle on complexity than previous technologies. Nevertheless, the growing size of applications and the demands for shorter time-to-market still leave many issues open for further

consideration. One of the issues is how to produce the software components as well as the application composed from them? Do we have to create them one by one manually? Or could we produce them automatically? What kind of technology can be applied to realize the benefits of component-based development?

Obviously, the productivity can be further increased if automation is introduced into the embedded software development process. However, this issue goes beyond the scope of pure component-based development. A joint consideration of component-based development and model-driven techniques can help us address the above problem.

1.3 Model-driven software development

Emperor Taizong of Tang (599-649) once said, "Using copper as a mirror allows one to keep his clothes neat; Using history as a mirror allows one to see the future trends." Before we discuss software and embedded systems, let us have a look at history and see what happened in the printing industry, in order to understand how the component-based approach is related to the model-driven approach. Before the technique for printing text was invented, how were books produced? All books were copied out by hand. If each page is seen as a component of a book, people had to manually replicate each component as many times as the number of book copies. This was not only a long and expensive process, but it also meant that every book was different. Although handwriters were conscientious in their work, the errors in copies of books were still inevitable. The more times a book was copied the more mistakes were made.

People must have been tired of copying books, which could be the reason why the printing technique was invented. For example, with the woodblock printing technique, people prepared a wooden pattern (Fig. 1.3), whereby the areas to show 'white' are cut away, leaving the characters or image to show in 'black' at the original surface level. It is only necessary to ink the block and bring it into firm and even contact with the paper or cloth, in order to achieve an acceptable print.

Usually, each block was created for one page of a book. If a page is a *component* of a book, the woodblock for the page can be seen as a *model* of the component. Therefore, the "model-driven book development" approach has been widely adopted in the printing industry. Compared to handwriting, the benefits of using woodblocks are obvious: 1) woodblock printing is faster and the book printing process can be automated; 2) all books from one set of woodblocks are identical and 3) the content of the book are kept on a set of woodblocks which is

independent of the media they are printed on.

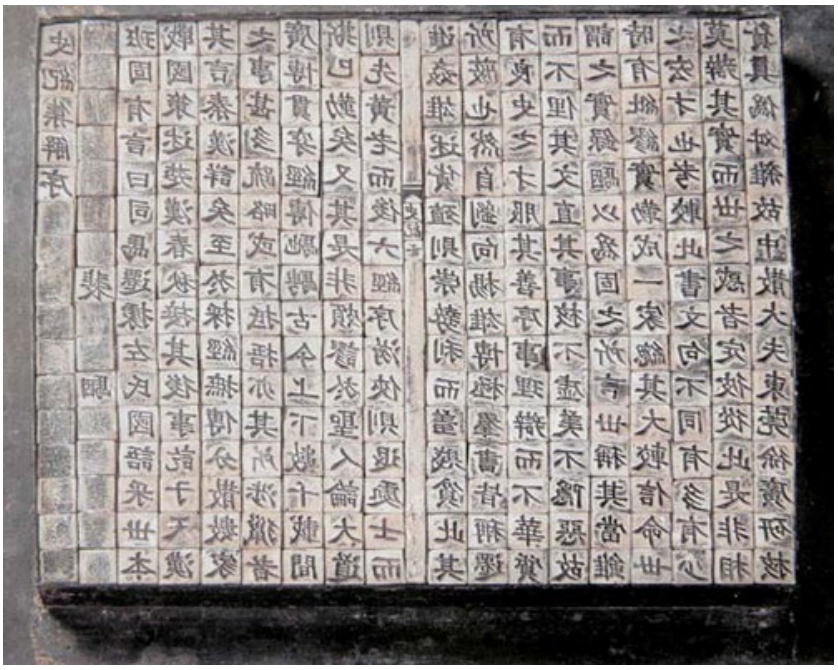


Figure 1.3: A wooden block

This technique, together with the new printing process, made the mass production of relatively cheap books possible for the first time. The effects of introducing models in the printing industry were dramatic: reliable maps could be printed for explorers, musicians could reproduce their work for others, and scholars could spread their new ideas.

Similarly, the relationship of a model to components can also be seen as a statue mold to statues or a cookie cutter to cookies, etc., which give certain shapes according to a pattern. Components can be created in many different ways, but shaping or forming them around, or on models is one of the ways to quickly create identical components. If a component is something in the real word, then the model is a conceptual representation or abstract description of the thing.

The way that people are creating software is not quite different from the way people copied books before the printing age came. The only changes are that books are replaced by software products, pages by software components, and handwriters by programmers.

In the field of embedded software, people are still handwriting software products! Even though some parts of software have been componentized and can be generated, most of software has to be written manually line by line, only with the

help of "copy" and "paste" commands. The titans of software industry can produce excellent software products mainly because they are maintaining a huge army of programmers, a finest development process as well as a management system, or a monopoly position within the companies' own areas, rather than adopting an approach based on system modelling and automatic code generation. The situation in small-scale companies is even worse, since they have neither the monopoly position nor the excellent development process and management system. In order to lower the cost of development, outsourcing is widely used in practice. However, it is especially difficult to outsource embedded software, as its execution hardware and external environment are required on the spot to test the developed software product. Consequently, months or years of delay from design to implementation are inevitably introduced.

The techniques of describing a software system have been limited as well. When source code is the only way to describe software, then the only people who can be intimately involved in its production are the programmers. Description of software is still on a relative low level of abstraction.

In fact, the history of software development has seen a steady lifting of the level of abstraction at which developers operate. Assembly language has been largely replaced by higher-level languages such as C. And the C itself has been marginalized by C++, .NET or Java with the widespread adoption of object-orientation. *Model-Driven Software Development* (MDSD) is perhaps the next evolutionary step in software technology, as it tries to separate a problem's solution from its technical detail.

MDSD [21][22] has received much attention as a way of improving the efficiency of software development. In a word, MDSD is a methodology advocating the use of models in software development. The goal of MDSD is to increase development speed through automation, enhance software quality by deriving implementation from model, and enable software "programming" on a more abstract level, using an adequate modelling language.

Model-Driven Architecture (MDA) [23][24][25][26] is a particular approach to model-driven development based on the use of the Object Management Group's (OMG) modelling technologies. OMG's version of MDSD helps to manage the growing complexity of software development by defining corresponding layers of abstraction. The focus of MDA is on standardization of notations, such as the Meta-Object Facility (MOF) [27], the Unified Modelling Language (UML), etc. The UML has been successful and started to be adopted in industry. MDA is "an approach to system development, which increases the power of models in that work. It is model-driven because it provides a means for using models to

direct the course of understanding, design, construction, deployment, operation, maintenance and modification" [23].

Model is a core concept of MDA. Models provide a higher-level of abstraction than the code, in which software components are usually written. Consequently, a problem can be described by models in a way that avoids delving into technological detail. A model can be represented in one or more notations. A notation determines how to represent a model as a specific set of symbols, and these may be either graphical or textual. MDA defines a model as follows:

> "A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modelling language or in a natural language" [23].

Platform is also a core concept in the context of MDA, MDA defines a platform as follows:

> "A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented" [23].

Based on the two concepts, MDA makes a distinction between *Platform-Independent Models* (PIMs) and *Platform-Specific Models* (PSMs). A platform independent model focuses on the operation of a system while hiding the details necessary for a particular platform. It reflects the part of the complete specification that does not change from one platform to another. A platform-specific model combines the specifications in the PIM with an additional focus on the details of using a specific platform by a system.

As a significant step of MDA, *Model transformation* "is the process of converting one model to another model of the same system" [23]. Typically, in the context of MDSD, a transformation occurs from PIM to PSM, which can be done manually, with computer assistance, or automatically.

To express a model, a modelling language is considered necessary. A modelling language is an artificial language that can be used to convey information or knowledge about systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. The flowchart, introduced in 1930s, that represents an algorithm or process using boxes and arrows of various kinds, is maybe the most popular modelling language adopted in designing or documenting a process or program.

As one of the OMG standards, the UML is a widely recognized and used modelling standard nowadays. It is a graphical modelling language for visualizing, specifying and constructing the artefacts of a software-intensive system. The UML offers a standard manner of representing system blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components. UML is designed to be compatible with the object-oriented software development methods. A set of diagrams are used as partial graphical representations of a system's model in UML, such as use case diagrams, class diagrams, sequence diagrams, etc.

As a general-purpose modelling language, UML is used for a wide variety of purposes across a broad range of domains. However, UML is often criticized as being large and complex, because it contains many diagrams and constructs that are redundant or infrequently used. Furthermore, learning and adopting UML is also problematic, especially when required of engineers, who are familiar with notations and concepts from a specific business domain, lacking the prerequisite skills.

UML includes a profile mechanism that allows it to be constrained and customized for specific domains and platforms. By using concepts like stereotypes, tagged values and constraints, the standard UML can be extended to fit a particular domain. For instance, in order to use UML in the domain of real-time embedded system, special UML profiles have been developed such as MARTE [28], which provides additional capabilities for model-driven development of real-time and embedded systems. However, basic UML knowledge is still required from the embedded system engineers.

Instead of trying to support all possible software problems, a Domain-Specific Language (DSL) concentrates on a specific subset of problems and directly addresses the problems faced by the end user in a particular area, such as telecommunication systems, real-time systems, control systems, GUI generation, etc. Domain-Specific Modelling (DSM) is a software engineering methodology for designing and developing systems, most often IT systems such as computer software. It involves systematic use of a DSL to represent the various facets of a system and to build domain models that describe a bounded field of interest or knowledge. DSM languages tend to support higher-level abstractions than general-purpose modelling languages, and they require less effort and fewer low-level details to specify a given system.

Having a modelling language designed for a specific domain allows a tight fit with exact needs of the domain, thus reducing the time needed by developers to

learn the modelling language, since it offers terms and concepts that are familiar to the developers. By focusing on the specific problem domain, such languages map more closely to the desired solution, which is thus easier to develop, optimize and use. In the domain of real-time embedded control application, a DSL can be designed as well.

A domain-specific language is usually defined by another language. In the context of MDSD such a language is often called *meta-model*. A meta-model is a special kind of model that specifies the abstract syntax of a modelling language. Consequently, the language for defining a meta-modelling language is called *meta-meta-model*. In the context of MDA, meta-models are expressed using a meta-meta-model standard – Meta-Object Facility. Essentially, it is a domain-specific modelling language designed for defining modelling languages. A meta-model of a DSL is an instance of the MOF, and a model built using the DSL is an instance of the meta-model.

The MOF framework for meta-modelling is based on an architecture with four meta-layers. These layers are illustrated in Fig. 1.4. The topmost layer in this architecture (meta-meta-model, MOF) defines an abstract language and framework for specifying, constructing and managing technology neutral meta-models. It is the foundation for defining any modelling language. All meta-models defined by MOF are positioned at the M2 layer, such as UML for specifying or visualizing software systems. The models of the real world, represented by concepts defined in the corresponding meta-model at the M2 layer are at the M1 layer. As an illustration, a vending machine model is defined in UML at the M1 layer. Finally, at the M0 layer are things from the real world, which can be considered as instances of model elements. For example, the vending machine UML model can be used to generate software code running in as many as vending machines in the real world.

In this study, meta-models at the M2 layer will be created in the MOF language, which can be used to describe domain-specific models used to specify distributed real-time embedded control systems. Tools based on these meta-models will be created to facilitate M1 layer models creation as well as transformation. Eventually, code will be generated and executed on a hardware platform at the M0 layer.

To summarize, the model-driven software development approach enables the domain application developers to focus on the design solution using platform-independent models, without being constrained by any implementation-related considerations. After a complete and fully validated domain model has been created, tools can be used for converting the model into a platform-specific model

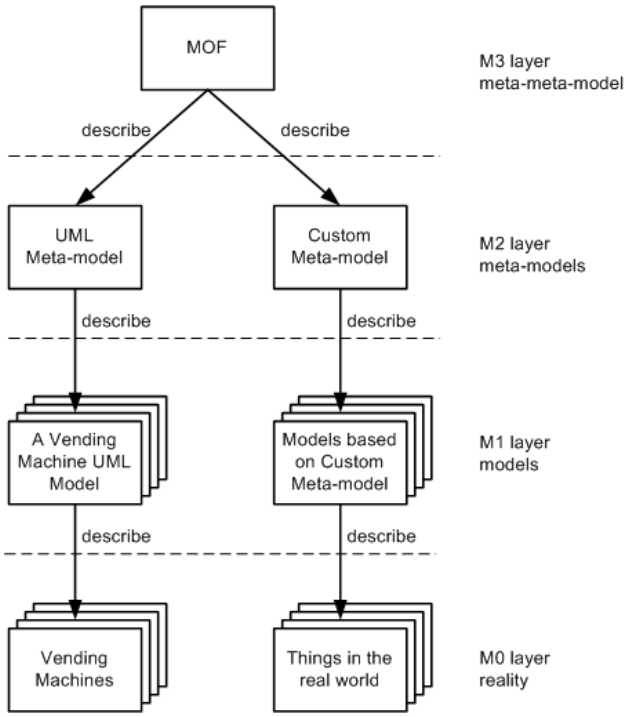


Figure 1.4: MOF architecture with four meta-layers

by integrating implementation details. That model is consequently used for generating the source code. In case of any change or bug removal, the models are updated and new code will be generated again. As a result, a domain expert who gets trained on the tools will be able to build the model himself, while the software experts are only used for developing new tools thus making the MDSD concept more and more viable. Therefore, it is not necessary to maintain a large number of programmers to write code for the application and consequently, the cost of development can be reduced.

However, such an excellent idea does not seem to have been fully adopted by software producers yet. According to Urban Roth, an IBM Rational sales specialist who gave a talk about MDSD from an industry perspective at the Mads Clausen Institute in 2009, less than 15% of the companies in Nordic countries have started to introduce the MDSD approach in the software development process. In order to get more knowledge of the situation regarding MDSD usage in industry, a simple survey has been made. A question: "Is the Model-Driven Development approach company-widely used to develop software products?" has been posted to the software development community. It was answered by 16 people having various software related positions in different industries, such as developer, architect, sales

manager, consultant, etc.

From the answers a conclusion can be made: software people believe that the MDSD is the technology of the future, because the use of models is quite analogous to what happened in other engineering disciplines such as mechanical engineering and electronics, which brought quite a lot of benefits. However, there are obstacles preventing the MDSD approach from being popular: on the one hand, it is hard to generate a complete application from the model; on the other hand, tool support is inadequate.

In particular, the popularity of this approach is limited by the insufficient maturity of the toolsets available. In most cases, the toolset is not there yet to support full-scale MDSD development. There are few tools that provide all MDSD features. Some model-based tools (e.g. IAR visualSTATE) deal with only a part of the development process and generate only the skeleton of the application, whereas the rest of the code has to be added manually.

However, the generation of complete applications is possible when MDSD is combined with a component-based framework, because in such a case, the domain-specific part (usually written manually) is implemented as a repository of reusable, prefabricated components.

Component-Based Development increases the productivity of software development due to the reuse of components. CBD focuses on the decomposition of the software, rather than the automation of the development process, which is addressed by MDSD. Thus, the combination of MDSD and CBD will lead the industry towards interoperable, reusable, portable software components based on standard models. With adequate tool support, automatic generation will also reduce the number of errors in the resulting programs, compared to manual coding, thus improving software quality and dependability.

1.4 Research motivation and goals

The widespread use of real-time embedded control systems poses a serious challenge to software developers, in view of severe and conflicting requirements that are related to issues as diverse as economy of production, time to market, safety and dependability. These systems are characterized by a number of domain-specific properties, such as tight interaction between software, hardware and the external environment, complex computation dedicated to certain tasks, reliable operation under hard real-time constraints and restricted operational resources. The above requirements cannot be met by currently used software technology, which is largely based on informal design methods and manual coding techniques.

As development of software for such systems is a challenging task, a new approach towards software development is needed, i.e. industrial production of software for embedded applications. This is a hot topic of research in the Software Engineering community, which can be largely characterized by keywords such as model-driven and component-based design of embedded software. The essence of this methodology is the adoption of formal models (frameworks) that are used to systematically develop software applications, and ultimately – configure applications from prefabricated software components, with the support of appropriate tools.

This methodology can be further enhanced by the introduction of automation during various phases of software development, resulting in systems that are easier to develop, integrate and maintain. Furthermore, discontinuity gaps typically arising in the process of development, whereby the implementation is not always consistent with the specification, can be eliminated since the implementation can be derived from, or generated directly from specification models.

A modelling language for specifying systems in the application domain is thus required. However, a shift in focus away from general-purpose modelling languages to domain-specific modelling languages provides a solution for the problem of language incompatibility between the application domain and technical solutions. With domain-specific modelling languages, application developers or domain experts specify an application using modelling techniques that belong to, or are even unique for the domain. Such a model bridges the gap between specification in the application domain and the implementation in the technology domain. Moreover, the DSL approach actually lifts the development task to higher levels of abstraction from which the desired implementations can be generated automatically. Consequently, domain experts can easily and quickly specify and generate or configure embedded software that can be understood by more people than just embedded programmers.

The combination of component-based and model-driven development will increase the productivity of embedded software development and the portability of the software. It can lead the embedded software industry into a win-win situation, as the two approaches have complementary advantages. On the one hand, the well-known problem, i.e. the impossibility to generate the whole software using MDSD, due to the lack of business logic, can be solved with the support of the CBD approach, where prefabricated software components carrying the business logic can be reused in a new application model. As a result, no code needs to be generated for these reused component models. On the other hand, the MDSD, as a software engineering paradigm, promises to reduce manual development work

by producing automatically software components and applications from models.

To reach the win-win situation in the domain of real-time control systems, a comprehensive component-based framework, reflecting the true nature of the domain, has to be created first. Developing such a framework, including a domain-specific modelling language, as well as component models and design patterns, is a highly complex engineering task, which is currently in the focus of attention of many research groups.

Research is currently going on at the Mads Clausen Institute, University of Southern Denmark, aimed at developing executable models and libraries of software components for embedded applications, which has so far resulted in the development of the COMDES framework (Component-based Design of Software for Distributed Embedded Systems) and design patterns for reusable and/or reconfigurable components such as function blocks, actors etc., as well as a timed-multitasking kernel architecture providing an operational environment for this type of system [19][29][30][31][20].

However, the developed software design methods can be efficiently used provided there is a computer-aided software engineering environment, consisting of a set of tools, supporting various stages of the software development process.

The goal of the project is to study the problems related to real-time embedded software design, using both CBD and MDSD approaches, and provide solutions and the associated software development environment that will hopefully contribute to the development of embedded applications. Accordingly, the research tasks of the project can be formulated as follows:

- Prototype meta-modelling of the COMDES DSL has been tried out during several projects, and has been partially defined [29][30]. These meta-models largely focused on the structure and behaviour of signal-processing components, execution and interaction of components, and composition of applications. However, to make the COMDES framework useful, the definitions of only DSL and components are not sufficient. It is also necessary to address issues like reuse, transformation to platform, allocation to networked platform, automatic generation of all executable code, etc. This means that the meta-model of the COMDES framework has to contain enough information, so that it can be used to specify domain-specific structure and heterogeneous behaviour for real-time embedded control systems. Then, it should be possible to systematically translate models into source code, and the translation can be efficient and complete. The transformations between model and model, model and code need to be considered as well.

- In accordance with the development process of the COMDES framework,

necessary tools supporting the process have to be identified and the functionalities of each tool have to be defined, such as system specification, component specification, component code generation, application configuration. Another issue is their integration into a complete environment and development of methodology providing a consistent representation and interplay between domain-specific modelling techniques, as well as an interchangeable representation of data used by various tools. The domain-specific development environment finally captures specifications and automatically generates or configures the target applications in the real-time embedded control engineering domains.

- Technologies required for building the tools as well as the engineering environment need to be studied and compared, in order to cover all aspects of the COMDES software development process. A prototype of the engineering environment integrating the tools has to be implemented. The environment itself must be implemented as exchangeable modules (plug-ins) within a higher-level framework, such as Eclipse or GME that will be used as a software bus integrating constituent tools. These and other relevant issues will be studied in the project with the ultimate goal of developing a prototype version of a toolbox supporting embedded software development under the COMDES framework.

A number of success stories in the traditional Software Engineering domain have proved that CBD is an important step along the road of improving software development productivity. We are trying to make this technology facilitate the development of embedded software as well. Furthermore, we strongly believe that the embedded software industry will increasingly adopt models as first-class artefacts, and tools automating the steps between models and executable code should be built to support this kind of development in a specific application domain.

To form an effective basis for the development of real-time embedded systems, this project is about building an integrated environment for modelling such systems with components, and for automatic configuration of the application and generation of its implementation. The study is expected to contribute to several fields of interest:

- To the embedded software development world, it presents a methodology for the development of distributed embedded control systems, which combines open architecture and predictable behaviour under hard real-time constraints. The embedded system is composed from autonomous system agents (actors), which are configured from trusted prefabricated components.

- To the component-based development society, it provides not only an implementation of reusable and reconfigurable components for embedded software, but also a framework allowing for applications to be composed of instances of these components. The components and application can be specified in a high-level abstraction as models from which implementations can be generated automatically.

- To the model-driven software development world, it provides meta-models describing the component-based framework including DSL, components, runtime environment, and platform, etc., specific to distributed real-time embedded control systems. Accordingly, a set of tools based on the meta-models that automatically transform embedded software models into code are specified and implemented. Furthermore, the study can also be seen as an illustrative example of how to adopt the MDSD approach in embedded software development.

- To the Eclipse community, since the toolset is ultimately implemented in the Eclipse platform, it provides a demonstration of building a domain-specific modelling environment, combined with a component-based software development environment, using the Eclipse projects.

Finally, the tools built in this project will hopefully help embedded control application developers build systems quickly and correctly, and inspire other researchers and developers to create similar tools that rescue programmers from the tedious manual coding work.

The remainder of the thesis proposes a solution to the problem formulated above, which is presented in more detail in the following chapters.

Chapter 2 presents design issues concerning tools and technologies that can be used to develop a software development environments (toolset) supporting component-based development of embedded software. A number of tools have been carefully studied in terms of both the functionality offered and the technology used in order to choose a suitable platform for toolset development.

Chapter 3 presents COMDES (Component-based Design of Software for Distributed Embedded Systems) – a domain-specific framework for embedded control systems, and the associated software design process. The framework contains a domain-specific language that can be used to specify the structure and behaviour of distributed real-time embedded control systems using prefabricated executable components.

Chapter 4 presents a detailed discussion of a set of platform-independent models that are used to specify components and systems under the COMDES frame-

work, which can be seen as an implementation of design methods presented in Chapter 3. These are illustrated using COMDES graphical notations with accompanying meta-models and constraints formally describing the domain models.

As a framework for real-time applications, COMDES needs a run-time environment to fulfill all the timing requirements of applications. Therefore, a meta-model of a real-time operating system is described in Chapter 5, which can be seen as a platform-specific model of a COMDES application. Furthermore, a specification is given to show how to transform a platform-independent model to a platform-specific model.

The presented COMDES models have been used to develop a number of component design patterns, i.e. possible implementations of the models. Chapter 6 presents design patterns for COMDES executable components of the *Basic*, *Composite*, *State Machine* and *Modal Function Block* kinds as well as *Signal Drivers*, and also discusses patterns of higher-level components such as *Actors*. These implementation patterns can be used to fulfill the functional requirements of a COMDES application.

The COMDES design method is essentially concerned with the models of embedded systems and model transformations in a computer-aided environment. Chapter 7 presents the transformation processes – from models to executables, as well as a prototype version of the envisioned software engineering environment for configuration of applications from prefabricated components. The Eclipse platform is used to host the environment.

The COMDES framework and its components have been experimentally validated in a number of real-time control case studies. Chapter 8 presents experiments developed during the Ph.D. project: a simplified version of well known Production Cell Case Study, using the developed toolset for the purpose of demonstration.

Finally, Chapter 9 concludes the thesis by discussing the problems addressed during the execution of the project, and outlines possible directions of future investigation.

This thesis has been developed under the Danish national research project – Model Driven Development of Embedded Systems (MoDES), carried out jointly by three academic partners: SDU/MCI (Mads Clausen Institute, University of Southern Denmark), AAU/CISS (Centre for Embedded Software Systems, Aalborg University), DTU/IMM (Department of Informatics and Mathematical Modelling, Technical University of Denmark), as well as a number of industrial partners, i.e. PAJ Systemteknik, Center for Software Innovation, Danfoss A/S, Skov A/S, etc. It has been funded by the Danish Council for Strategic Research.

Chapter 2

Model-driven Software Development Frameworks

We are pursuing the goal of creating a development environment comprising a comprehensive set of tools that will be used to develop embedded real-time software. Users of the tools will be able to model a system, check the models, and generate the software through one or more transformation steps. To obtain such tools, the simplest way could be: grab a bunch of programmers, start coding from scratch as soon as possible, and wait for a couple of years. During the process, outsourcing could be a choice to lower the cost.

However, this is not something that a Ph.D. student with only three years of project time can afford. Fortunately, there are other tools and development platforms available in the world of MDSD, which can be used as foundations to assist and speed up the development of the tools that we need. Such tools or environments typically provide a meta-modelling language – a set of generic concepts that are abstract enough such that they are common to most domains. They can significantly lower the cost of obtaining tool support for a DSM language by providing assistance for meta-model creation and generating parts of the implementation i.e. editors for the modelling language. As a result, we need only to focus on the meta-model constructs that are tailored for the domain characteristics.

This chapter first introduces a number of existing system development technologies for embedded real-time control systems mainly from the tool point of view, and then provides a brief overview of typical modelling technologies and engineering methodologies for MDSD with respect to CBD that could be used to create domain-specific tools.

2.1 Examples of embedded system development tools

Presently, there are numerous component-based and model-based software development technologies for embedded real-time control systems available in both the academic and commercial domains [32]. Some of them are quite sophisticated and have been validated in industrial applications. This section presents a brief overview of tool support for several software design methods that are typical examples in their areas of application: industrial control systems specified in terms of function block diagrams and other languages defined in standard IEC 61131-3; embedded applications specified in terms of hierarchical and concurrent state machines, and finally – component-based development of software for distributed embedded applications.

2.1.1 IEC 61131-3 standard

The automation industry has adopted the IEC 61131-3 standard, which defines a number of special-purpose languages for programmable logic controllers (PLCs) used for automation of electromechanical processes, such as control of machinery on factory assembly lines, amusement rides, or lighting fixtures. This kind of controller is usually designed for multiple inputs and output arrangements, extended temperature ranges, immunity to electrical noise, and resistance to vibration and impact. A PLC is an example of a real-time system since output results must be produced in response to input conditions within a bounded time, otherwise unintended operation will result.

Under IEC 61131-3, a PLC can be programmed using one or more languages defined in the standard. IEC 61131-3 is the only global standard for industrial control programming. It harmonizes the way people design and operate industrial controls by standardizing the programming interface, which allows people with different backgrounds and skills to create different elements of a program during different stages of the software lifecycle: specification, design, implementation, testing, installation and maintenance. The standard includes the definition of the Sequential Function Chart language, used to structure the internal organization of a program, and four interoperable programming languages: Instruction List, Ladder Diagram, Function Block Diagram and Structured Text. Via decomposition into logical elements, each program is structured, increasing its reusability, reducing errors and increasing programming and user efficiency.

Presently, there are a number of vendors who offer IEC 61131-3-based devel-

opment environments for control applications, such as MULTIPROG from KW-Software GmbH, OpenPCS from SYS TEC electronic GmbH, SIMATIC STEP 7 from Siemens AG, CoDeSys from 3S-Smart Software Solutions GmbH, etc. Due to the fact that these development environments are designed for the purpose of commercial use, they typically contain a number of tools providing comprehensive support for a broad variety of tasks that are typically executed in an automation project:

- The toolset basically provides a programming environment supporting all the five programming languages according to definition in IEC 61131-3.
- It usually comes with an offline simulator to execute and debug programs no matter if any control hardware is available. The simulator can run on the same computer as the programming system. As a result, an application can be debugged before it goes to the real process and thus save valuable time when it comes to integration.
- It offers a compiler that generates code containing controller-specific binary instructions that can be directly executed.
- It provides a library of loadable function blocks for control tasks. For example, a standard PID control algorithm is implemented as a function block that can be graphically configured with the appropriate parameters, and be used wherever closed-loop control tasks are needed, e.g. temperature control, pressure control, flow control as well as fill-level control, etc.

Apart from the above mentioned essential features, an IEC 61131-3 development environment usually also supports features like deployment management, controller online updates, version control, system diagnostics, etc. which are helpful and meaningful to industrial users.

This kind of environment can be illustrated with the CoDeSys Automation Suite – a comprehensive software development toolset for industrial automation systems. All common automation tasks solved by means of software can be realized with the CoDeSys Suite based on the widely used controller and PLC programming system of the same name. Matching the IEC 61131-3 standard, it supports all standard programming languages but also allows for the inclusion of C-routines.

CoDeSys v3.3 consists of a number of tools for PLC software development. The basic toolset includes:

- CoDeSys V3 – the IEC 61131-3 programming system supporting all of the programming languages defined in the IEC standard (see Fig. 2.1)

- Gateway Server – for the communication between the CoDeSys programming system and compatible controllers
- CoDeSys SP Win V3 – a SoftPLC under Windows NT/2000/XP with soft real-time properties

The development environment contains the PLC programming system with complete online and offline functionality, compilers as well as additional components for configuration, visualization etc. Optional tools are available for application in specific domains, i.e. motion control. Communication between the development and the device layer is based on the CoDeSys Gateway Server.

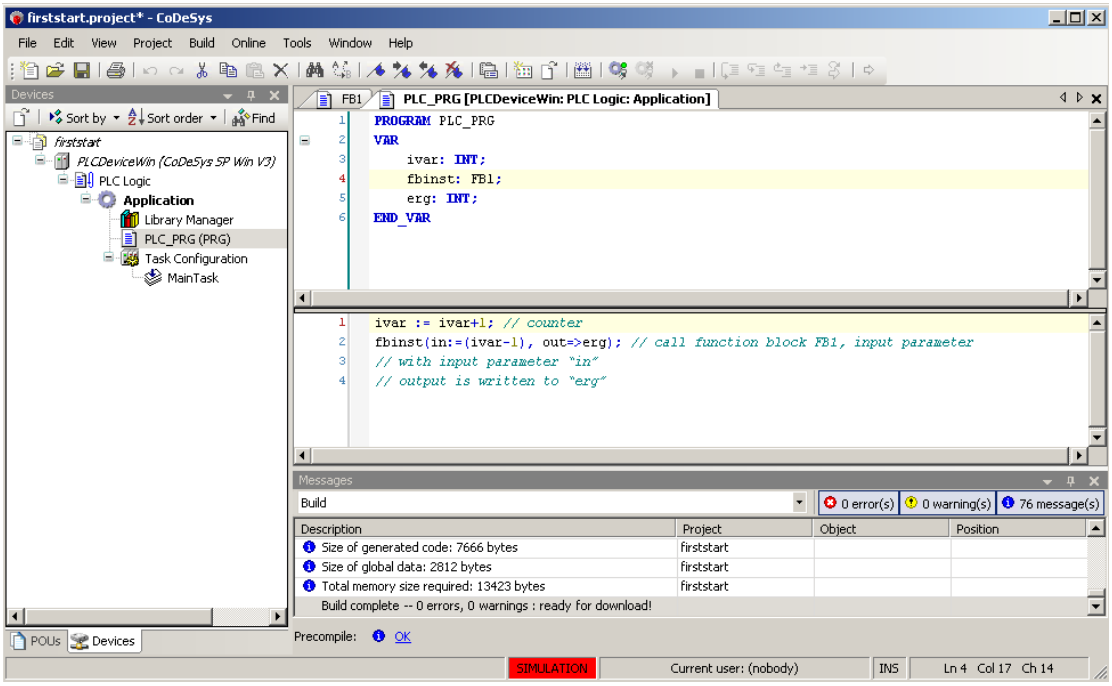


Figure 2.1: CoDeSys programming system

Apart from programming, the toolset also supports deployment by organising a project to include the programming objects that make up a PLC program together with the resource objects necessary to run one or several instances of the program (application) on certain target systems (PLCs, devices).

Programming objects (POUs) such as programs, functions, function blocks, methods, interfaces, actions, data type definitions etc., are not device-specific but they might be instantiated for use on a device. For this purpose, program POUs must be called by a task of the respective application. The execution of programming objects is controlled by one or several tasks. Real preemptive multitasking

can be realized, but the user must explicitly take care of the synchronization problems.

A device object represents a specific (target) hardware object, such as controller, fieldbus node, bus coupler, drive, I/O-module, monitor, etc. Each device is defined by a device description that specifies the properties of a device concerning configurability, programmability and possible connections to other devices. It must be installed on the local system in order to be available for allocation. It is possible to run an application on a simulation device which is per default available within the programming system. So, no real target device is needed to test the online behaviour of an application.

A standard library is provided, containing all functions and IEC 61131-3-compatible function blocks that are required as standard POUs for an IEC programming system. Several other libraries are shipped supporting additional functionalities, e.g. a utility library containing a collection of various blocks that can be used for bit/byte manipulation, auxiliary mathematical functions, signal generators and analogue signal processing, etc.; communication-specific libraries allowing an IEC application to remotely access devices in accordance with the type of fieldbus used; and a basic library providing function blocks for motion control, i.e. controlling the motion of a single axis as well as the synchronized motion of two axes.

The tool supports code generation through integrated compilers and the use of machine code results in short execution times. However, machine code will not be generated until the application project gets downloaded to the target device (PLC, simulation target). Therefore, no code generation is done when the project is compiled via the build commands. The build process is done to check the project for syntactical errors.

The development environment itself is built as a component-based system, where the functionality available in the environment depends on the currently used plug-ins (components). In addition to the essential system plug-ins, customer specific plug-ins can be optionally created, such as various editors, code generators, communication drivers, etc.

2.1.2 IAR visualSTATE

IAR visualSTATE is a set of highly sophisticated and easy-to-use development tools for designing, testing and implementing embedded applications modelled with hierarchical and concurrent state machines (UML Statecharts). It provides advanced verification and validation utilities and generates very compact C/C++ code that is 100% consistent with system design. As a commercial tool, it also

provides automatic documentation generation with comprehensive information.

The visualSTATE toolset comprises the following fully integrated tools allowing for development and test of embedded applications based on Statecharts diagrams:

- Navigator – a graphics-based project management tool for the overall handling of visualSTATE projects, from model design over test and simulation, to code generation and documentation of visualSTATE projects
- Designer – a graphics-based application for designing Statecharts diagrams using the UML notation
- Verificator – a powerful test tool for dynamic formal verification of models created with the Designer
- Validator – a graphics-based application for simulating, analysing, and debugging models created with the Designer
- RealLink – a tool for testing a model in a target application
- Coder – the tool can automatically generate code on the basis of models created with the Designer
- Documenter – a tool allowing for creating an up-to-date documentation report on a visualSTATE project, including design, tests, and code generation

In addition, it can be fully integrated with the existing IAR Embedded Workbench, a fully integrated C/C++ compiler and debugger toolset, enables true state machine debugging on hardware, including direct graphical feedback in various levels of detail.

As a tool for design tasks dealing with functional behaviour, it has a graphics-based editor for designing Statecharts diagrams using the UML notation (see Fig. 2.2). An embedded application can thus be designed by drawing objects, events, and actions etc, based on the concept of hierarchical state machines that can capture concurrent behaviour inside one state machine. The concept also incorporates both Mealy semantics, in which actions are associated with transitions between states, and Moore semantics, in which actions can take place within states.

The design philosophy of visualSTATE can be expressed as follows:

- Map events in the environment, like device driver input or interrupts, to state machine events.

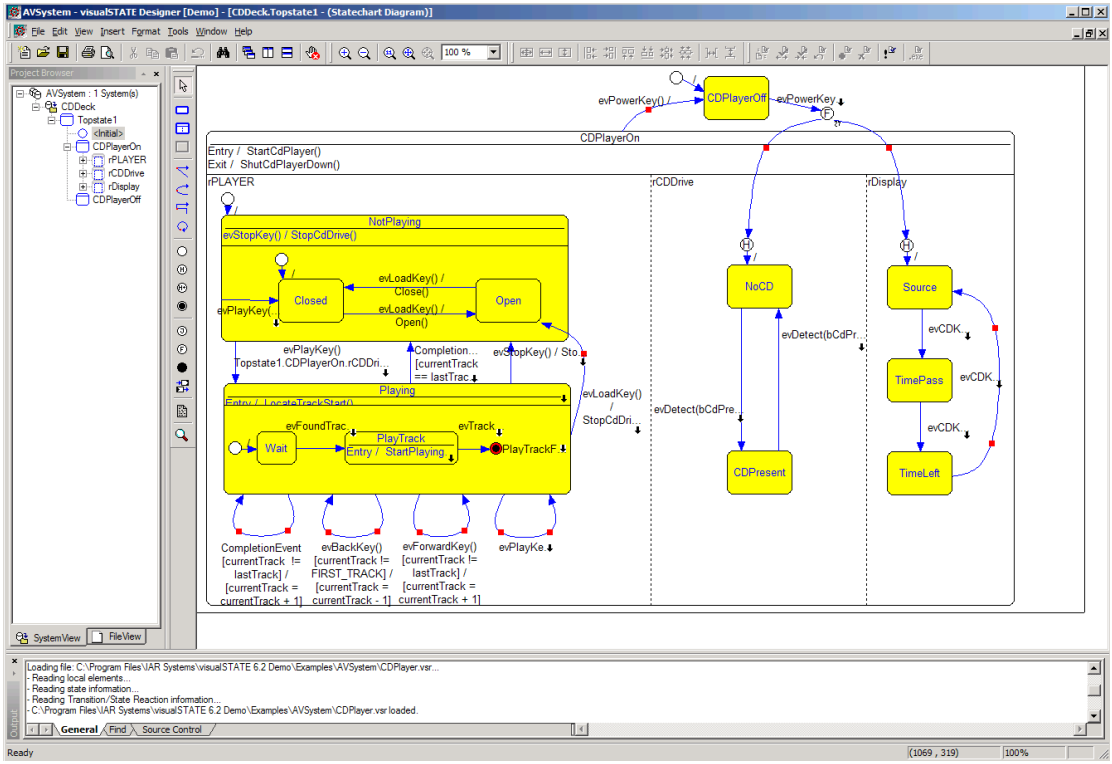


Figure 2.2: A state machine model in visualSTATE

- Capture the discrete system logic in states, events, transitions and actions using UML diagrams.

- Map actions to functions or device drivers interacting with the environment.

By separating the core system logic from the environment, it is possible to generate extremely tight C code for the state machine logic. As a result, it is easy to port a design to different hardware platforms with minor modifications involving events and actions. The generated code can run on any target, regardless of Operating System (OS). There are no requirements on any OS services, since a design model can express concurrency among state machines. If an OS is present, it is very easy to integrate a visualSTATE model into the run-time environment.

Formal verification can be performed on the model, thereby discovering possibly erroneous behaviour at an early stage. The visualSTATE verification is an example of model checking, which is a set of different methods of algorithmically verifying that a model satisfies a specification, such as absence of deadlock, unreachable states or unused events etc.

There is also a validation tool available to ensure at an early stage of design

that the application behaves as expected, even before the hardware exists. It offers operations such as stepping through states, setting breakpoints, animation, etc. The functionality includes graphical animation of the state diagram when executing, the possibility to set breakpoints at the state machine level instead of the C level, as well as trace and log functionality.

In summary, visualSTATE is an integrated development environment for developing, testing, and implementing embedded applications based on Statecharts diagrams. It includes several tools, which help increase productivity and quality when developing embedded software. Its design model is especially suited to dealing with the discrete-event behaviour of systems.

2.1.3 AutoFocus

AutoFocus [16][17][18] has initially been developed at the Technical University of Munich (TU Munich) and further development has been carried out by TU Munich and Validas. It provides a graphical software development environment for distributed embedded systems. The main focus of AutoFocus is on the modelling, simulation, validation and code generation facilities for component-based embedded applications. Furthermore, it supports version control, test management and test visualization.

The tool supports a modelling language that comprises a set of concepts used to describe distributed systems, i.e. components, ports, channels, control states, etc. These concepts are based on the idea of a system being made up of a network of communicating components. A simplified representation of the AutoFocus meta-model is shown in Fig. 2.3 [16], using the UML class diagram notation as the meta-modelling language.

As the main building blocks of systems, components encapsulate data, internal structure, and behaviour. Components can communicate with their environment via well-defined interfaces. Components are concurrent: each one of them runs sequentially; however, in a set of components, each component's run is independent of the other components' runs. A global system clock drives all components in a system, and each component carries out one operation per system clock cycle. Components can be hierarchically structured, i.e., consist of a set of communicating sub-components.

Control states and transitions define the control state space and the flow of control inside a component. Each transition connects two distinct controls states (or one control state with itself, in case of a loop transition) and carries a set of annotations i.e. guards, assignments, etc. determining its firing conditions.

Ports are used for the communication between a component and its environ-

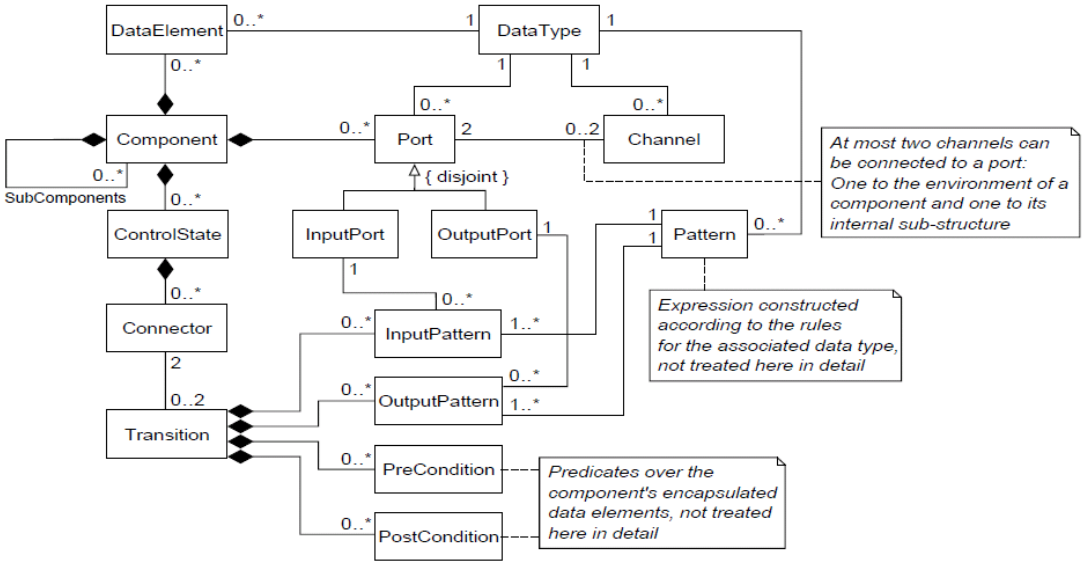


Figure 2.3: Basic modelling concepts of AUTOFOCUS: the meta-model

ment. Components read data from input ports and send data to output ports. Ports are named and typed, allowing only specific kinds of values to be sent/received to/from them. There are two kinds of ports: so-called immediate ports and delayed ports. Immediate ports pass along the value that is written to them immediately, that is, within the same system clock cycle, whereas delayed ports propagate values written to them not before the next system clock cycle.

Channels connect component ports. Channels are unidirectional, named, and typed. They define the communication structure of a system.

AUTOFOCUS uses a signal-based communication. Communication is performed in a synchronized manner: all components communicate in a time-driven round-based scheme with a global time-rate for all components. Within each round, events for occurrence of a message as well as the non-occurrence can be detected.

The above concepts are sufficient to describe a large class of systems. Based on these concepts, AUTOFOCUS provides four different views of the system. Each of those views concentrates on different aspects of the system, i.e. system structure, functional behaviour, etc. The integration of the views on a common semantic basis [17] leads to an integrated formal specification of the system.

System Structure Diagrams (SSDs) (Fig.2.4) describe the static aspects of a distributed system via a network of interconnected components exchanging data over channels. It provides both the topological view of a distributed system and the syntactic interface of each individual component. System structure diagrams

are represented graphically, where rectangular vertices symbolize components and arrow-shaped edges stand for channels.

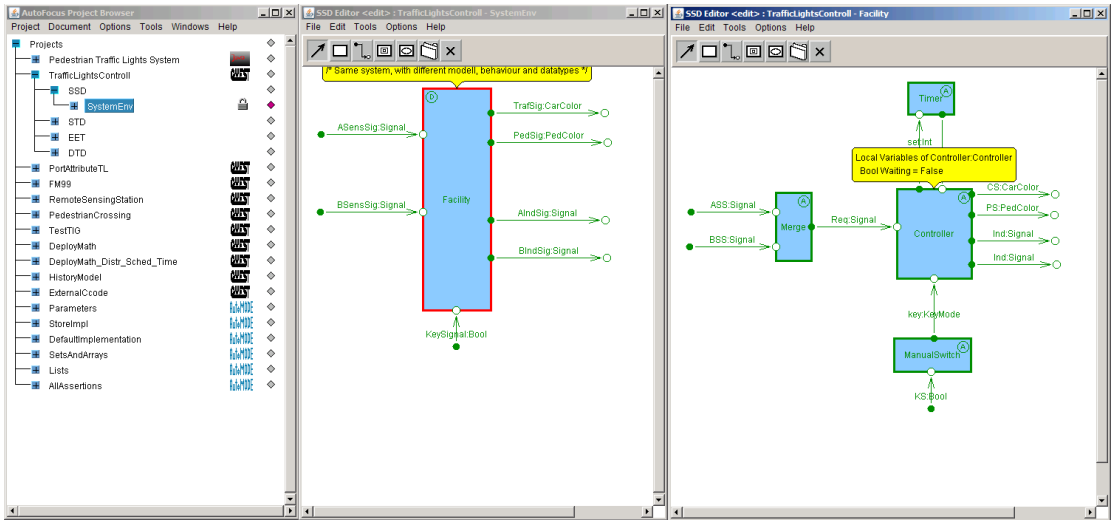


Figure 2.4: AutoFocus system structure diagrams

The types of the data processed by a distributed system are defined in a textual notation. It allows the experienced developer to define his own data types and corresponding functions while designing of the model. The data types defined here may be referenced by other views, for example as channel data types in SSDs, or by local variables of components.

State transition diagrams (STDs) are used to describe dynamic aspects, i.e. the behaviour of a distributed system and its components. Each system component can be associated with an STD, and each state of a state transition diagram can have an STD as a substructure. These diagrams are similar to Statecharts diagrams, known from the UML modelling language. Graphically, they are represented as graphs with labelled oval nodes as states and arrows as transitions.

Besides STDs, extended event traces (EETs) may also be used to describe the behaviour of distributed systems by exemplary runs associated with a component-based view of the system. The interaction view can be represented as extended event traces describing the communication within the system, and between the system and the external environment. The notations used are similar to message sequence charts (MSCs) or sequence diagrams.

Architecture. The AutoFocus tool is designed as a distributed environment with a common repository containing all project-related documents and multiple clients connected over a network [18]. The core of an AutoFocus client

is the project browser, displaying all projects, their documents, and versions available in the repository. By selecting a document, the project browser requests the document from the repository and opens a window using the appropriate editor.

Editors. AutoFocus provides different editors for the graphical description techniques. All of these editors use an identical user interface concept with mouse-based user interactions to facilitate fast editing of SSDs, EETs and STDs. Hierarchical diagrams, which are a core concept of the AutoFocus system model, are fully supported by the editors.

Library. Various predefined libraries of basic components are available. If there are no suitable components for a particular area of application, it is also possible to build a dedicated library containing the required components.

Syntactic checks. Abroad variety of syntactic consistency checks is offered by the AutoFocus tool, such as: Do all components have a refinement or behaviour? Are all the ports connected and are all the channels bound? Do all components have a unique name? Are the transitions connected?

Simulation. Semantic diagram consistency is supported by the tool by simulating the system. It is possible to run the simulation using the AutoFocus GUI on the host computer, without any real embedded hardware. It allows developers to view the SSDs and the STDs and to track the signals graphically. In the simulation process the cycle-time can be adjusted, so that the simulation can be viewed at the speed desired. The simulation also visualizes the state of the whole system including active automaton states and values of variables and channels, which offer a good overview on how the entire system reacted to input signals.

Code generation. From the models created with the AutoFocus tool Java, Ada or ANSI-C code can be generated with corresponding code generators. The generated code can be run on various platforms, where Java, Ada or C code may be compiled and executed. Using cross compilers the ANSI-C code may be adjusted to run on almost any embedded system.

2.1.4 Summary

Based on the introduction given so far, it can be clearly seen that the embedded development tools basically concern the description of system in high-level languages based on a number of concepts or models, and transformation of the system description into low-level code. Advanced development tools usually place the emphasis on modelling in a domain-specific language rather than coding in a general-purpose language.

The tools from the IEC 61131-3 family have been accepted by the automation industry, thus the functionalities provided by the tools directly reflect industrial

requirements. The visualSTATE is a model-driven development tool employing state machines as the underlying modelling concept. It contains tools not only for modelling and code generation, but also for analysis, such as verification and validation (simulation). Unfortunately, it does not provide adequate support for components, and reuse is possible only at the source code level.

AutoFocus is a good example of an environment integrating the model-driven and component-based development approaches, whereby a system is decomposed into a number of components that have a well-defined meta-model as foundation. Users start software development by formally modelling the system, using certain graphical and textual notations. While modelling a system, components that are saved in libraries can be reused at model level. Apart from code generation, checking and analysis against models are supported as well. Both system structure and functional behaviour can be modelled in this environment. However, the modelling concepts do not deal with timing behaviour explicitly.

Such tools reveal typical features of a modern embedded software development environment, concerning model-driven and component-based development: domain concepts used as foundation, graphical modelling of components and systems, library of component models, analysis of the system in order to validate the model, code generation out of the validated model, deployment on various hardware platforms, etc. However, developing domain-specific modelling tools supporting all these features is a very complicated task. It can be eventually made easier by using design methods and software engineering environments (platforms) specifically targeted at tool development.

The rest of this chapter will discuss software technologies and development platforms that can be used to create comprehensive domain-specific tools and toolsets, using a model-driven software development (MDSD) approach.

2.2 Required aspects and features of MDSD platforms

As depicted in Fig. 2.5, when implementing the MDSD developers typically start with creating a meta-model of the domain-specific language used to specify the PIM. The DSL is used to model the structure and behaviour of the target system on an abstract level. It is also necessary to create a meta-model for the PSM, in case the PSM and the PIM are written in different DSLs. These abstract PIM models will then successively be transformed into more specific models, and transformations between PIM and PSM have to be precisely specified. Textual artefacts such as code, configuration scripts, documentation, etc., will be finally

generated, resulting in the desired system. Thus, developers have to deal with a lot of different meta-models, and they need tool support in defining, transforming, and generating meta-models.

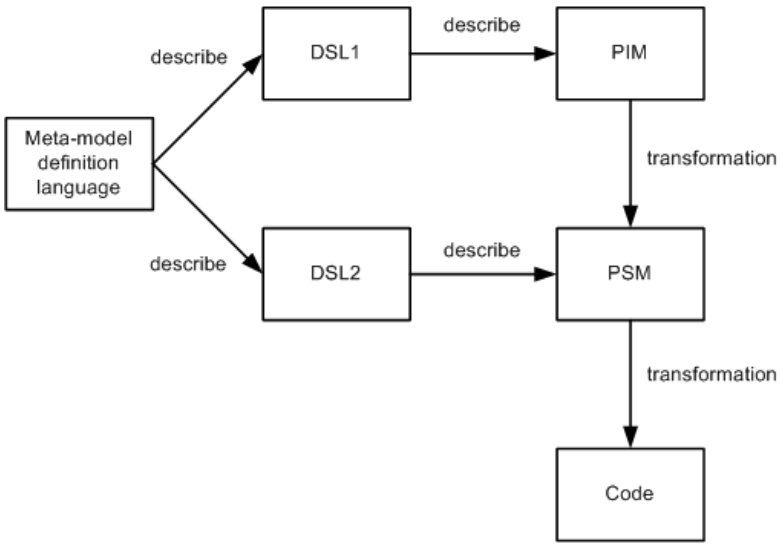


Figure 2.5: DSL development process

In terms of the process, languages like meta-model definition language, GUI definition language, constraint definition language, transformation definition language and code generator definition language are necessary to construct a DSL, as well as the corresponding tools.

The meta-model definition language (meta-meta-model) is used to build a meta-model that defines the structure of the DSL involving the vocabulary of domain concepts in the language, how these concepts can be combined to create domain-specific models, etc. The meta-model definition language is usually rich enough to describe modelling languages for a wide variety of domains.

The GUI definition language defines the way that models are presented, such as using graphical notations or textual notations. In other words, the meta-model definition language defines the abstract syntax of a DSL, while the GUI definition language defines the concrete syntax.

Although the structure of models has been restricted by a meta-model of the DSL, when model instances are created by developers, they could still be incomplete or incorrect. Hence, certain constraints on the meta-models have to be specified, so as to make sure that their *static* semantics is correct. In this context, static semantics means the set of rules that specify the well-formedness of domain models, such as multiplicity constraints, aggregation constraints, etc., that are

typically defined in a constraint definition language.

The meaning of the domain concepts in a DSL is given by *dynamic* semantics, which guides the execution and transformation of the models. As DSL adopts concepts from the problem domain, it should be designed intuitively clear, so that a domain expert would easily know the meaning of the language elements. Denotational semantics and operational semantics are the main approaches that can be used to formalize the meaning of DSL constructs [33]. Model transformation usually implements the DSL's semantics.

Model transformations are an integral part of the model-driven software development approach. Transformations between models that conform to corresponding meta-models can be implemented in a model-to-model transformation language. If the target of a transformation is text or code, it can be viewed as a special case of model transformation – model-to-text transformation. A language that specifies a code generation process can save countless hours and reduce tedious coding. Furthermore, since the eventual goal is to extract information from the model data in some way, facilities are needed to parse models based on meta-models before doing any transformation.

For example, the Model Driven Architecture [23], which has been proposed by the Object Management Group, provides support for MDSD by offering a conceptual framework with an emphasis on standardization. Within the scope of MDA, the OMG standard provides Meta Object Facility as a language for the definition of meta-models and thus, a DSL can be based on MOF. The GUIs for UML can be used as GUIs for the DSL too, provided that the DSL extends the UML meta-model using profiles. A particular instance of the meta-model is exported as an XMI (XML Metadata Interchange) document, which is accepted by most of the MDA tools as an input format. UML class diagrams do allow the specification of some basic rules, for example, the multiplicity of associations. For more complex semantic specifications, however, UML employs Object Constraint Language (OCL) expressions to specify constraints. Together with UML 2.0, action semantics is a way of specifying algorithmic behaviour, i.e. the implementation of class operations. Query/View/Transformation (QVT) is one of the OMGs standards used to describe transformations between source and target models. Although there is no standard for code generation, a number of technologies exist on the market parsing UML-based models and transforming them into code.

Presently, there are several notable DSL development tools from academic research that meet the requirements discussed above. However, there is one more thing to consider. While the MDSD approach focuses on different aspects of software development other than CBD, some meta-modelling approaches disregard

the means for componentization and reusability at the model level. However, in the context of CBD, component reuse is a natural requirement. When taking both CBD and MDSD approaches into account, some DSL development tools explicitly provide means for modelling components, and afterwards, allow for the use and reuse of the defined component models when creating application models. Therefore, support for componentization and reusability at the model level by the fundamental DSL development tools should also be a point of consideration.

The following sections provide a brief overview of MDSD tools that can be used as a foundation for building specialized tools constituting an engineering environment for component-based development of embedded software. These tools are evaluated in terms of the aspects discussed above.

2.3 GME

The Generic Modelling Environment (GME) [34][35] is a powerful tool developed by Vanderbilt University Institute for Software Integrated Systems (ISIS), supporting graphic definition of domain-specific modelling languages and the capability to generate domain-specific graphic modelling environments. GME allows the users to define a meta-model paradigm using an extended UML class diagram notation with constraints written in OCL. Based on the meta-model paradigm, GME generates a domain-specific modelling environment, whereby entities defined in the paradigm are available to graphically construct domain-specific models.

The tool is used mostly in the context of the Model-Integrated Computing (MIC) approach [36] that addresses the problems of designing, creating, and evolving information systems by providing domain-specific modelling environments, including model analysis and model-based program synthesis tools. MIC places models in the centre of the entire life cycle of systems, including specification, design, development, verification, integration, and maintenance. MIC facilitates MDSD by providing a technology for the specification and use of domain-specific modelling languages (DSMLs), a fully integrated meta-programmable MIC tool suite, and an open integration framework supporting formal analysis tools, verification techniques and model transformations in the development process. Moreover, MIC not only deals with automatic applications synthesis from the model, but also emphasizes model analysis. Following the MIC approach, one will be able to develop a model integrated computing environment for a given problem.

The syntactic definitions of a DSL are modelled using the MetaGME language (Fig. 2.6 [34]). The MetaGME offers a set of generic concepts such as Project, Folder, Model, Atom, Connection, etc., to create a meta-model. The static se-

mantics are specified with constraints using the OCL. Thus, complex constraints such as relationships constraints, rules for the containment hierarchy and property values can be represented in GME. Meanwhile, a built-in constraint manager is provided to enforce all domain constraints during model building. This constraint manager can be invoked explicitly by the user, and it is also invoked when event-driven constraints are present during modelling.

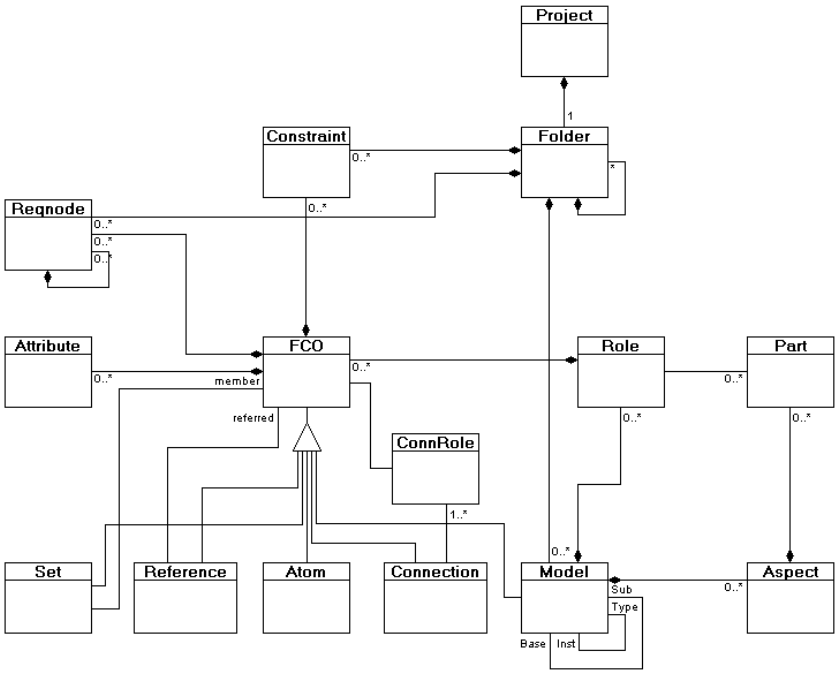


Figure 2.6: GME modelling concepts

An editor for domain-specific modelling is automatically configured once its meta-model has been defined and imported into the tool, which requires no GUI definition language. Additional methods are available for customizing GME for implementing domain-specific visualization of the models. But this kind of customization capability is limited without modification of the source code of the tool.

GME is good at creating diagrams describing the static structure of a system such as class diagram and UML component diagram, but it is hard to make diagrams representing the dynamic behaviour of a system like message sequence chart, UML activity diagram, etc., without largely modifying the tool.

GME itself does not provide any transformation definition language. But there is a Graph Rewriting and Transformation (GReAT) language [37][38] – a graphical language for the specification of graph transformations between domain-specific modelling languages built in GME. GReAT uses meta-models to specify the abstract syntax of the input and the target models, and sequenced graph rewriting rules for specifying the transformation itself.

GME provides several techniques, i.e. The Builder Object Network (BON), for programmatic access to the GME model information. Access to the objects, and to the relationships between them, is available through methods that act on these objects. The obtained information can be further used for generating program code or system configuration.

GME offers no code generation definition language. However, the architecture of GME can be extended through its Add-on mechanism. External tools such as code generator can register to receive some events and will be automatically invoked when the events occur.

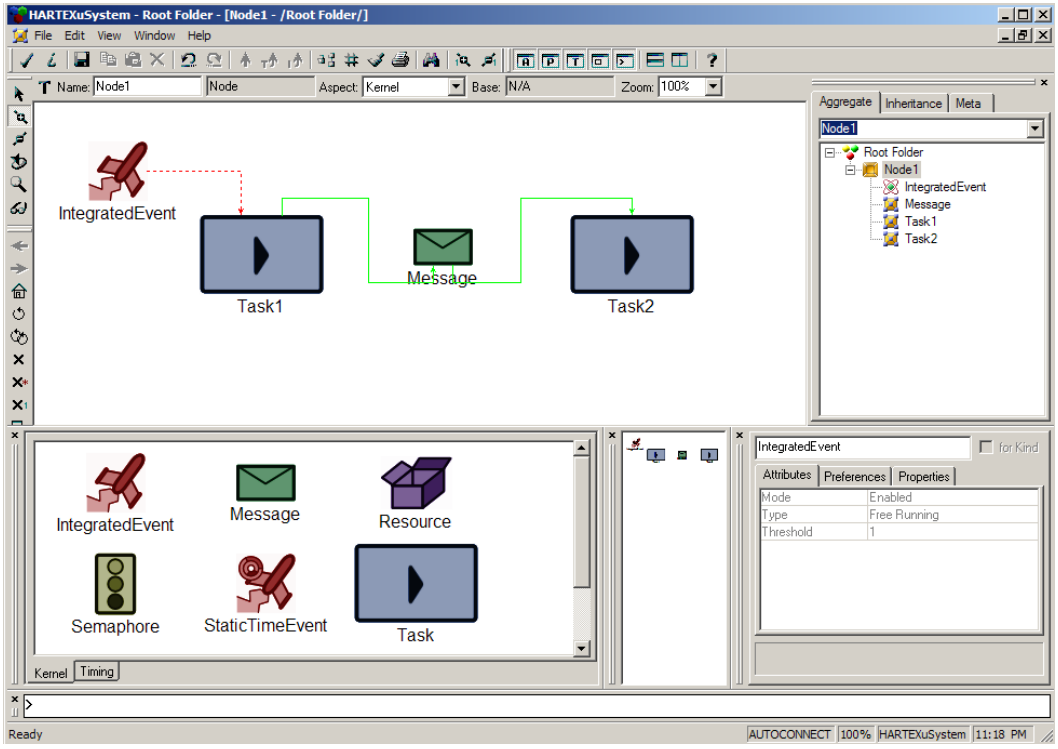


Figure 2.7: A real-time kernel model in GME

GME supports model reuse by providing concepts of types and instances. By default, a model created from scratch – based on a metatype – is a type. An

instance of a model can be created from a type. Therefore any modification in a type model propagates down to all its instances. In this way, it is possible to create libraries of type models that can be used in multiple applications. Based on predefined and verified models residing in these libraries, the developer is able to create new instances in his or her project without losing the connection to the prototype model. Thus, further enhancements and corrections in the original model can be easily propagated to all of its instances automatically. Types, instances and the creation of instances from types are supported by the GME modelling engine, thus such models and mechanisms need not be constructed in meta-models.

The tool GME (current version 7.6.29) is available at the website of the Institute for Software Integrated Systems (`http://www.isis.vanderbilt.edu/projects/tools`). It is based on Microsoft COM technology and runs only on the Windows platform. The source code is available for download too, so it is possible to make customization with any language that supports Microsoft COM technology (C++, Visual Basic, C#, Python etc.).

2.4 Cadena

Cadena [39] is an Eclipse-based extensible integrated modelling and development framework for component-based systems, developed by the Laboratory for Specification, Analysis, and Transformation of Software (SAnToS) in the Computing and Information Sciences Department of Kansas State University. Cadena models are type-centric in that multi-level type systems are used to specify and enforce a variety of architectural constraints relevant to the development of large-scale systems.

Cadena meta-modelling capabilities can be used to formally capture the definition of widely used component models such as the CORBA Component Model (CCM), Enterprise Java Beans, NesC, etc. Cadena meta-modelling can also be applied to specify new component models, including domain-specific component models that are tailored to the characteristics of a particular domain or underlying middleware capabilities.

The Cadena Architecture Language with Meta-modelling (CALM) is the meta-modelling language on which Cadena is based. It employs a three-tiered meta-modelling approach to define a modelling language for a given component framework (Fig. 2.8 [40]), where a model in a particular tier defines the language or vocabulary of entities that can be used in constructing models in the tier below it. Constraints like multiplicity of associations can be specified when building the meta-model.

The style tier is used to define structures of the architecture elements according to domain properties (i.e. the vocabulary from a domain) by specifying domain-specific languages for building types of components, interfaces, and connectors, etc. It describes a meta-model of the architectural elements that can be used in the construction of a system. Meta-models for PIM and PSM can be built in the style tier in CALM. Having component kinds defined in this tier guarantees that component types and instances conform to the vocabulary specified by the style, enables precise specification of components used in the underlying component framework, and enables precise specification of domain-specific component modelling languages.

Once an architecture or platform has been described in a style, those domain-specific languages can be used in the module tier to define component and interface types within a particular architecture. These component types and interface types conform to the component kinds and interface kinds described within the style. Types serve as a template from which a set of component instances can be generated; changes in a component type will propagate to all of its instances. The scenario tier is used to allocate instances of declared component and connector types. Scenarios contain instances of component types, instances of other scenarios (as nested scenarios), and connectors, which tie the instances together.

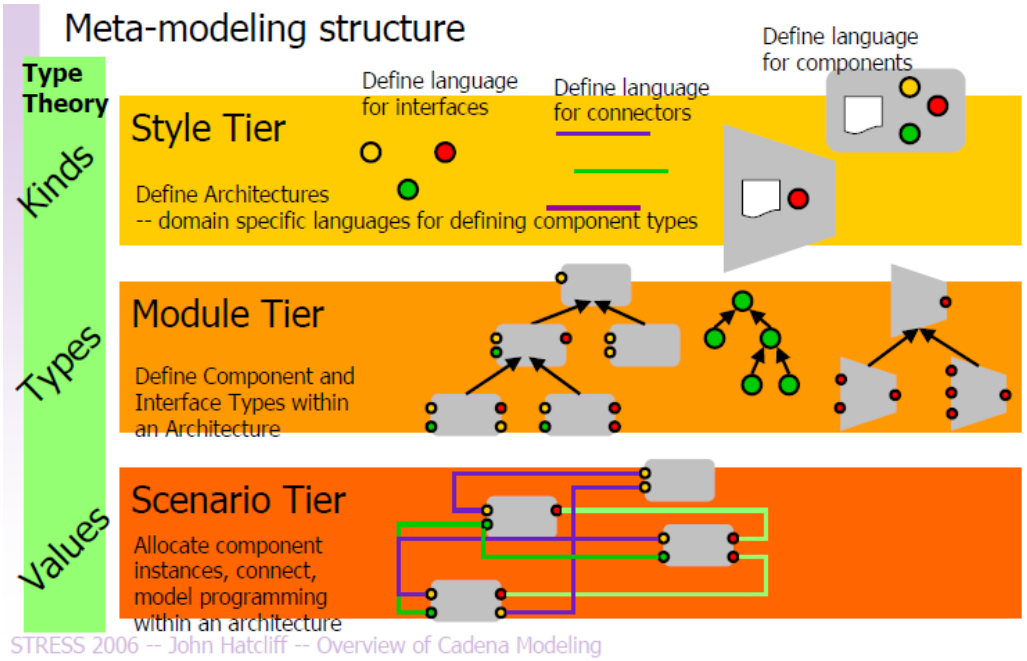


Figure 2.8: Cadena's three tiered framework

The Cadena tool supports both definition of meta-models and models in a form-based or graphical-based GUI. It can in turn be used for creating models that conform to the newly defined meta-models. A GUI definition language is not necessary if a new GUI for a given domain-specific modelling is not required. Means for constraint definition, transformation definition and code generation are not offered by Cadena. It supports only multiplicity and multiplexity as constraints when defining a meta-model. CALM models can be connected to analysis and code generation facilities via Cadena plug-ins. Some of Cadena plug-ins serve as model interpreters that realize the semantics of CALM models. Cadena core APIs are available to parse a model (scenario). In addition to that, the Cadena object model itself is implemented using the Eclipse Modelling Framework (EMF). As a consequence, every Cadena model can also be parsed by EMF APIs too.

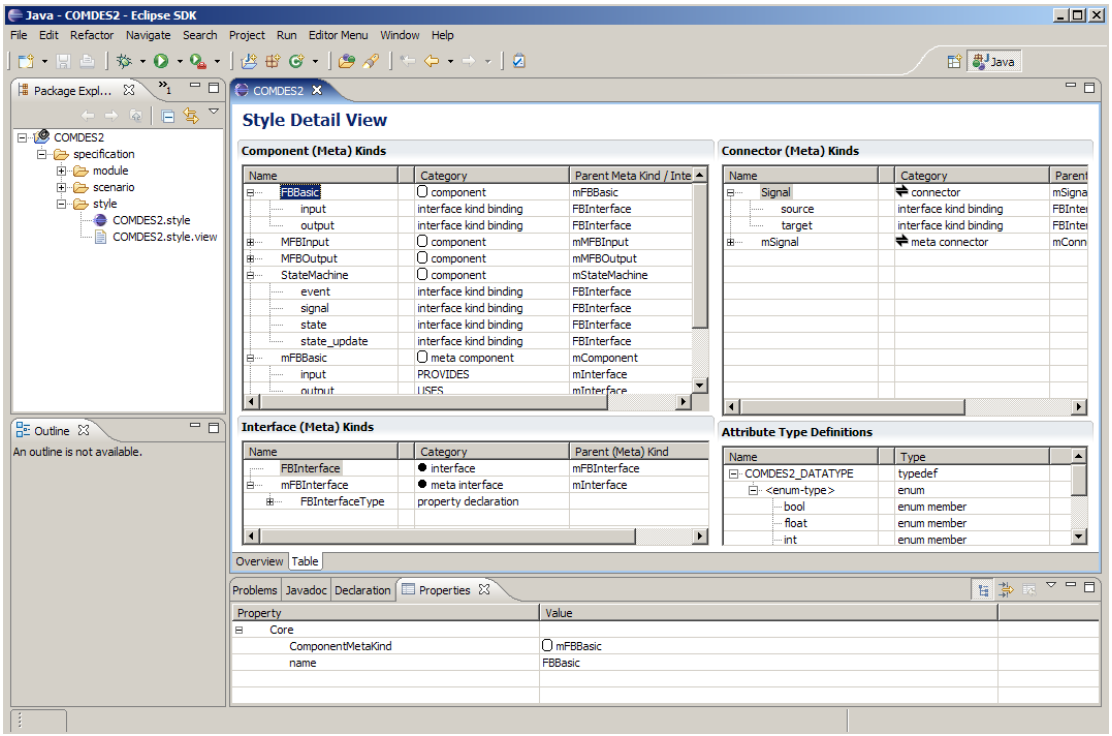


Figure 2.9: A meta-model in Cadena

The Cadena is built using the Eclipse environment and framework, leading to easy enhancement and extension of the features that Cadena currently provides. This enables developers to build model-driven engineering environments that include facilities for editing component implementations, model-level configuration,

code generation, simulation, verification, and creation of system builds, etc.

As a MDSD tool, Cadena is directly related to CBD, which makes Cadena different from other MDSD tools that are not dedicated to the CBD approach. Its type system matches quite well the inherent reuse requirement of component-based systems. As a result, the reuse of components in Cadena becomes easy, since the component type system has been integrated into the development process following the three tiers, and in turn does not have to be meta-modelled. Without such a type system, component kind, type, instance as well as relations among them have to be explicitly constructed in meta-models. Meanwhile, implementations of such a type mechanism must be provided.

To summarize, CALM/Cadena is "a rigorous type based framework for modelling multiple component middleware platforms, systematically organizing and transitioning between platform definitions, and creating customized development environments that leverage domain knowledge and automate development process steps to enable early design decisions for entire product lines"[41].

The tool (currently v. 2.x) is available at the website of the Cadena project (`http://cadena.projects.cis.ksu.edu`). It is an Eclipse-based tool, thus requiring Eclipse 3.2 or 3.3 and J2SE 5.0 series Java Virtual Machine.

2.5 MOFLON

MOFLON [42] is an integrated, standard-compliant meta-modelling environment developed by Real-Time Systems Lab, Technische Universität Darmstadt. With MOFLON, developers can create well-structured meta-model of a DSL and generate the complete application logic in Java code. The generated code features tailored and reflective access interfaces, an event mechanism, constraint checking, and XMI import and export facilities.

MOFLON has an OMG's MOF 2.0 standard-compliant visual language as a meta-modelling language to specify the abstract syntax of a domain-specific modelling language (Fig. 2.10). Furthermore, OCL 2.0 is employed to specify constraints that cannot reasonably be expressed visually. Consequently, the static semantics of a meta-model can be expressed more precisely.

Unlike other tools that usually implement the semantics through a model transformation or code generation process, MOFLON provides a language – Story Driven Modelling (SDM) that allows for the specification of the dynamic semantics of domain-specific languages. When specifying the operations of a MOFLON class, the actual behaviour is described in a story diagram [43] in terms of activities and transitions using a graphical editor, rather than being written as a piece

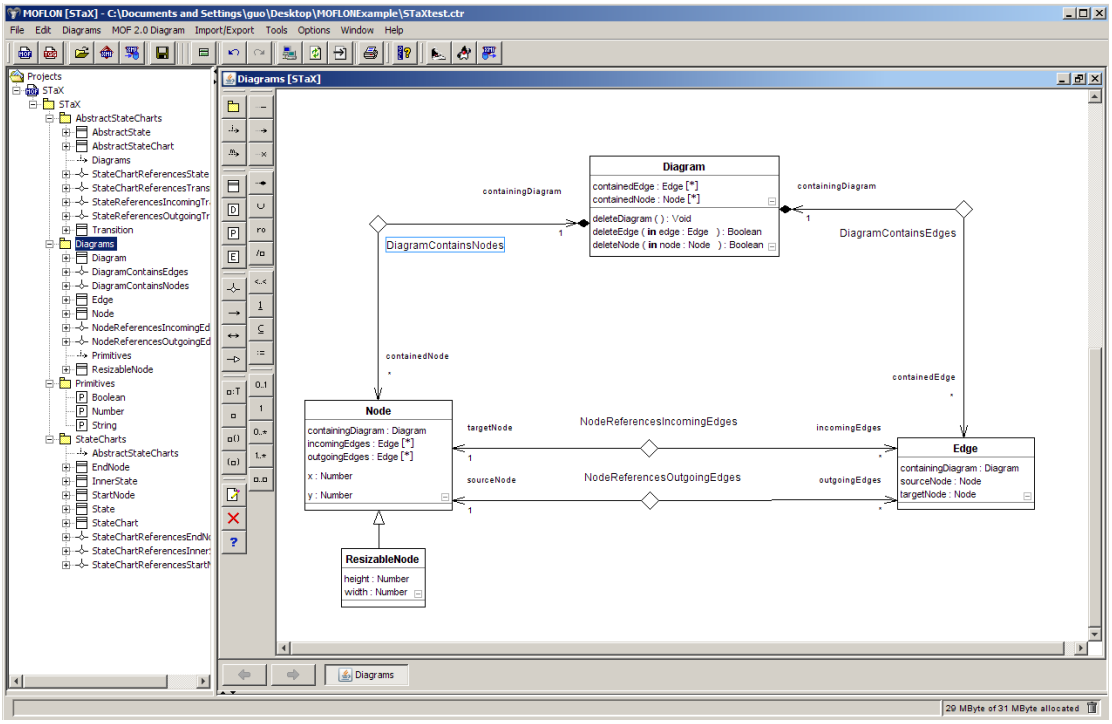


Figure 2.10: A meta-model for a state machine design language

of source code (Fig. 2.11).

From all specifications MOFLON generates Java code, which conforms to Sun's Java Metadata Interface (JMI) standard – a standard for metadata management. This code can then be utilized in order to analyse, transform, and integrate models that conform to the domain-specific meta-models. JMI defines both a tailored and a reflective interface for meta-models. Reflective interfaces are independent from a considered MOF specification and can be used for generic access of a compliant model and generic exploration of the underlying meta-model. Tailored interfaces allow for convenient typed access to models based on the specified meta-model.

MOFLON is more about meta-modelling, model analysis, transformation and integration; hence, it does not support a concrete syntax. An editor for the DSL model has to be provided by developers manually, with the help of the Java code generated from meta-models.

Model-to-model transformation in the world of MOFLON is also referred to as model integration. An approach based on Triple Graph Grammars (TGG) [44] allows for the visual and declarative specification of model-to-model transformation rules. TGGs explicitly maintain the correspondence of two models by means of correspondence links that map elements of one model to elements of the other

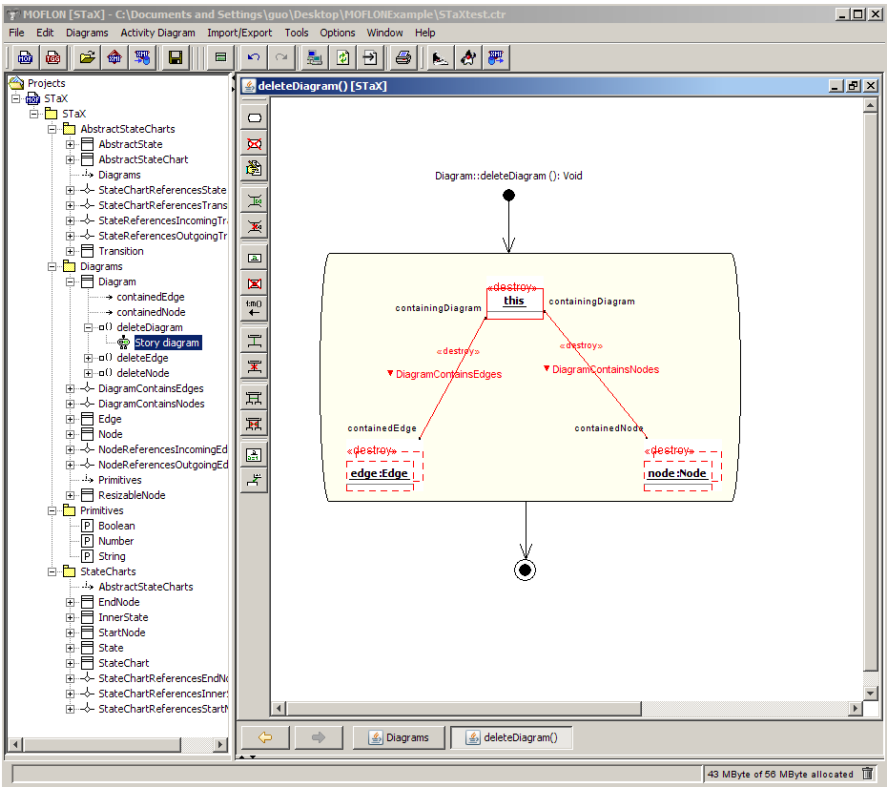


Figure 2.11: A story diagram specifying an operation that deletes a diagram and its contained elements

model and vice versa.

As a meta-modelling environment, MOFLON enables its users to visually specify the domain-specific meta-model of a considered system. However, by design it has no concern of component-based development, and thus offers no facility for component model reuse like those provided by GME and Cadena. But models for component type and instance can be specified with the MOF, and the typing could be either specified in a story diagram or implemented, based on generated Java code.

Within the MOFLON environment, the above described technologies – MOF 2.0, OCL 2.0, SDM, and TGG have been combined into a solution for a meta-modelling and specification language [45]. With these technologies, meta-models and constraints of a DSL can be created, models can be transformed and the complete application logic can be generated in JMI-compliant Java code. A DSL solution for modelling, model analysis, transformation, and integration can be developed based on the generated code combined with suitable parsers, tool adapters

and user interfaces.

The tool (currently, v. 1.3.1) is available at `http://www.moflon.org` for Windows, Linux and Mac platforms. The Java run-time environment is also necessary to install and run the tool. The source code of the tool can be accessed as well.

2.6 MetaEdit+

MetaEdit+ [46][47] is an environment for developing domain-specific modelling languages and code generators. The research behind the genesis of MetaEdit+ was carried out at the University of Jyväskylä.

MetaEdit+ facilitates modelling language definition with form-based tools based on the GOPPRR meta-meta modelling language (a language for defining modelling languages). The language provides a set of meta-modelling concepts like Graph, Object, Property, Port, Relationship, etc. Based on these concepts, it is also possible to create a meta-model graphically. Because there is no need for any compilation of code, the language definer can test the modelling language under construction while building it. For defining the modelling constraints, instead of providing means to program the constraints of the problem domain into the language, MetaEdit+ offers a wide variety of rule templates to choose from. Some constraints are defined by simply choosing the templates from dialog boxes rather than programming them. The modelling environment automatically enforces the constraints during modelling, thus supporting the modeller in making correct designs. The templates are limited and can express constraints like types, occurrence, connectivity, uniqueness, setting default values and defining a regular expression to validate the input values, etc.

MetaEdit+ provides a symbol editor that allows DSL developers to design their own visual representation for the modelling language [48]. A graphical notation can be drawn with the editor, thus a GUI definition language is not necessary. A feature, that distinguishes MetaEdit+ from other tools is that an objects appearance can also be defined conditionally, so that it changes automatically if one of its own, or other object's property values, roles or relationships are changed. MetaEdit+ offers three editors for modelling – a diagram editor, matrix editor and table editor, allowing the users to choose and switch to whichever format is more convenient. This feature makes the tool already best-suited for graph-like languages (Fig. 2.12 [49]).

In MetaEdit+, generators are defined using the MERL scripting language. It provides powerful means for navigating through the model structures (multiple models and different modelling languages) accessing the design data according to

the meta-model. MERL can access multiple models and generate multiple files, set protected regions into generated files, and access external files and tools during generation.

MetaEdit+ provides an application-programming interface (API), which can be used to access meta-model as well as model elements and data programmatically in real-time. The API also supports model import and export as XML (Extensible Markup Language).

The tool does not provide a model-to-model transformation definition language, but the transformation can be implemented via the MetaEdit+ API, which can be used to read, create, and update models [50]. Capabilities of extending the tool with other tools are limited; the MetaEdit+ can only execute external commands via generators. It does not support reuse of models.

MetaEdit+ (currently v. 4.5) is a commercial tool, which is available for all the major platforms including Windows, Linux, and Mac OS. A 31-day evaluation version can be downloaded from `http://www.metacase.com`. As a commercial tool

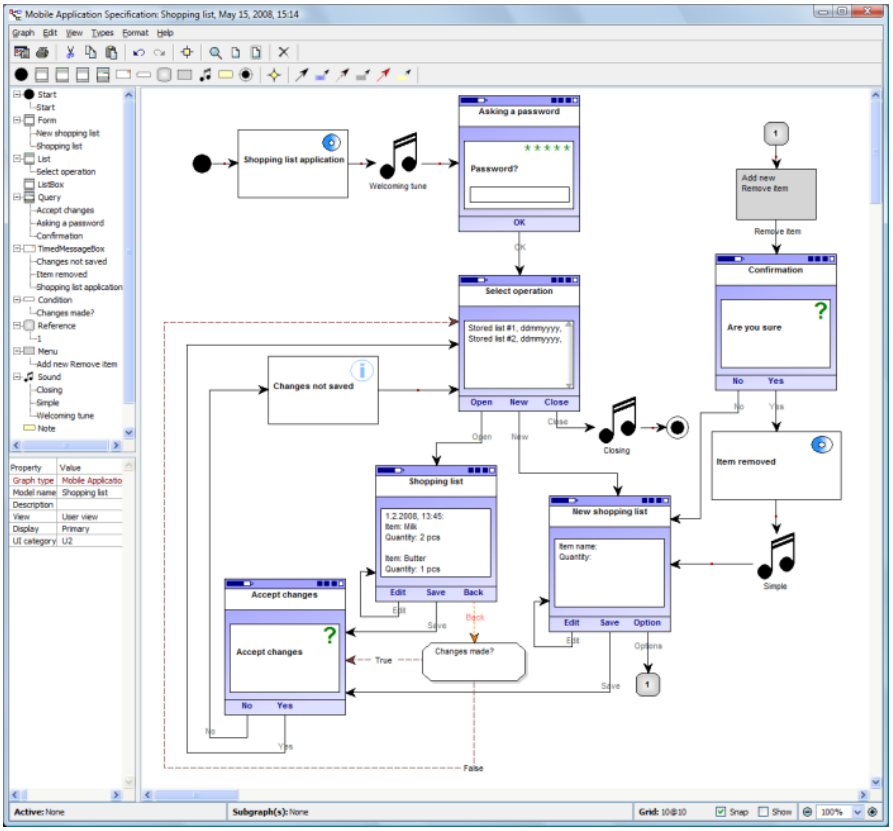


Figure 2.12: A mobile application specification language in MetaEdit+

whose version 1.0 was released back in 1993, it offers practical and mature features that are useful for daily development work, such as multi-user environment, versioning of models, model animation while running code, automatic tracing of generated code back to model elements, etc.

2.7 Eclipse modelling project

The Eclipse platform can provide the technology foundation for a DSL development environment. It offers many projects related to domain-specific modelling tool design. There is a broad spectrum of solutions, which offer a variety of capabilities.

The Eclipse Modelling Framework (EMF) is a modelling framework for that platform. It provides facilities to create a meta-model, with the support of a meta-modelling language – the Ecore meta-model that is aligned closely with the eMOF (essential MOF).

Implementation classes in Java can be generated from Ecore models. These classes provide a tailored API for building instances of the meta-model. Domain-specific models that conform to the Ecore models can be also parsed by the generated tailored API or EMF reflective API. EMF also comes with a couple of additional generators that generate editors and a generic editing framework for editing the models. The EMF contains a validation component providing the capacity of constraint definition written in Java or OCL for any EMF meta-model.

The Eclipse Graphical Modelling Framework (GMF) project can be used for the rapid development of standardized Eclipse graphical modelling editors, by providing a generative component and runtime infrastructure for developing such tools. It supports the automatic generation of graphical editors for EMF meta-models. To obtain a graphical editor for a DSL based on the EMF meta-model, an additional model must be created to be used by the GMF generator, which describes how the generated editor will look like and behave. The editor can be generated out of that model. Furthermore, specific behaviour or graphics can be added using manual coding.

Since model-to-model transformation is a key aspect of MDSD, the model-to-model transformation (M2M) project delivers a framework for model-to-model transformation languages. ATL (ATLAS Transformation Language) is a model transformation language and toolkit. It provides ways to produce a set of target models from a set of source models based on EMF technology. Its model-to-model transformation engine has matured over the past few years and is in widespread use. Alternatively, as an OMG standard for model transformation,

QVT (Query/View/Transformation) defines a standard way to transform source models into target models, which is available as an M2M component too.

There is a Model-to-Text (M2T) project that focuses on the generation of textual artefacts from models. Java Emitter Templates (JET) is a generic template engine that can be used to generate source code and other output from templates.

Like MOFLON and MetaEdit+, the EMF meta-modelling environment does not provide special support for component-based development, similar to what GME and Cadena offer. Models for component instance and type need to be specified with the Ecore, and the typing has to be implemented based on generated Java implementation classes.

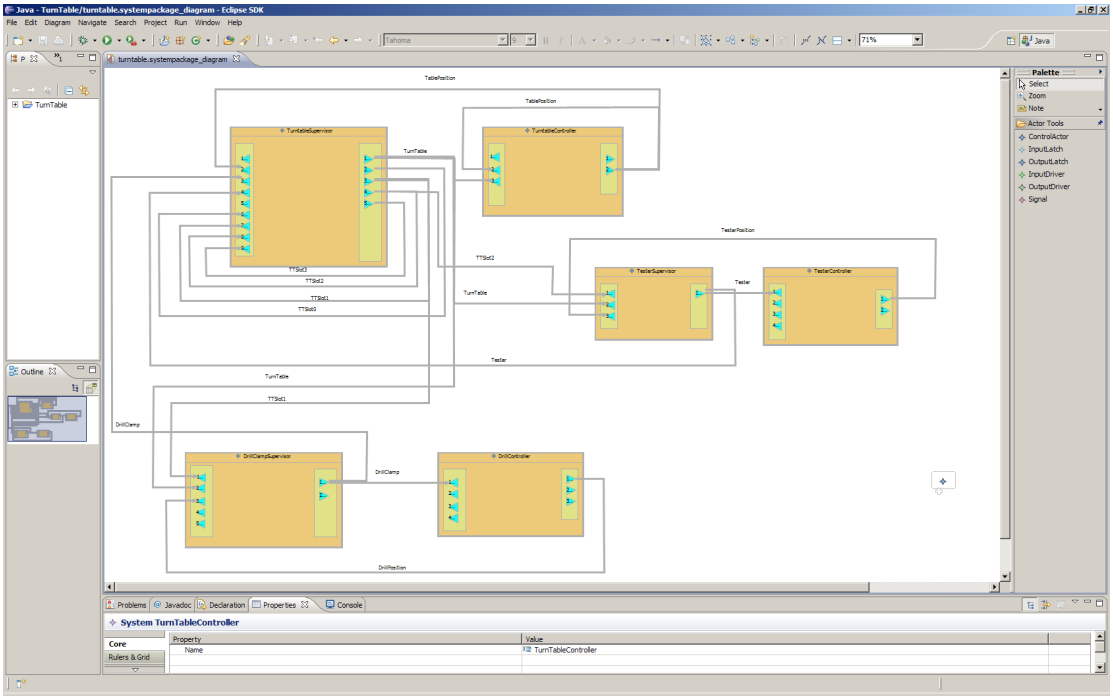


Figure 2.13: A graphical DSL in Eclipse

As an open platform, the Eclipse consists of more tools other than modelling, i.e. it provides the tools Eclipse Java Development Tools (JDT) and Plug-in Development Environment (PDE) for development tools for Eclipse. Because the Eclipse is the de-facto platform for implementation of this project, the introduced Eclipse-based technologies in this chapter will be described in more detail in the following chapters.

The Eclipse (currently v. 3.4) can be downloaded from `http://www.eclipse.org`. There is an Eclipse modelling package that contains a collection of Eclipse

Modelling Project components, including those introduced in this section. It is a Java-based tool, and can run in Windows, Linux and Mac platforms.

2.8 Summary

There are also other publications comparing similar tools and technologies, used to construct a DSL and related software development tools. In [51], Sivonen has compares several DSL development tools and technologies, in order to build a tool for a DSL and a code generator. Adopting the MDSD approach to develop the tool, the survey is focused only on meta-modelling languages, constraint languages, as well as generator definition languages. The above paper also compared the availability of the development tools such as license, documentation and support, etc. It does not consider other criteria, listed in our survey, that are instrumental for a development environment when using MDSD for component-based development.

In [45], fifteen integrated solutions and technology families for DSL construction are compared covering categories like OMG standards, family of schema-based graph/model transformation environments, classic meta-CASE solutions, as well as text-based approaches. The work covers a broad range of criteria i.e. abstract syntax, concrete syntax, static semantics, dynamic semantics, model analysis, model transformation, model integration, acceptability, scalability, tool availability and expressiveness. However, they do not mention the model reuse aspect, which is quite important when developing a MDSD solution for component-based development.

A summary of the features of the presented MDSD development environments is given in Fig. 2.14. The survey of the above tools, in view of the defined requirements, is based not only on the reviewed publications, but also on our experience gained while using these tools, to build a prototype toolset supporting the COMDES framework.

Tools like GME, Cadena, MetaEdit+ and Eclipse are actually integrated meta-modelling development environments as they support two distinct tasks: 1) They support the definition of meta-models, constraints, concrete syntax and editors for user-defined DSLs; 2) They make the newly-defined DSLs available in the tool, allowing the application developers to use the user-defined DSLs. Such kind of tool is very convenient when constructing a DSL, because once a meta-model of the DSL has been built, modelling editors based on the meta-model can be easily configured or generated, resulting in easy testing of the meta-model.

Compared to GME, Cadena, MetaEdit+ and Eclipse, MOFLON lacks any support for the definition of the concrete syntax of modelling languages. MOFLON

	Metamodeling	GUI	Constraint	Dynamic semantics	M2M	Parser	M2T	Model reuse	Environment extensibility
Cadena	CALM	Build-in	No	Transformation	No	Cadena core APIs	No	Kind, Type, Instance	Cadena plug-in
GME	MetaGME	Built-in	OCL	Transformation	GReAT	BON	No	Type, Instance	Add-on mechanism
MOFLON	MOF2.0	No	OCL	SDM	TGG	Tailored API, Reflective API	No	No	No
MetaEdit+	GOPPRR	Symbol Editor	Rule templates	Transformation	No	MetaEdit+ API	MERL	No	No
Eclipse	ECORE	GMF	OCL, Java	Transformation	ATL, QVT	Tailored API, Reflective API,	JET, 3rd party	No	Plug-in

Figure 2.14: Tools for building a component-based development environment

only generates a Java implementation conforming to a meta-model, which can only be used as a foundation of building a modelling editor; this makes it harder to use when testing and tuning the meta-model. Furthermore, MOFLON is not an extensible environment, meaning that tools for processing models (i.e. code generator) built in MOFLON cannot be integrated into the MOFLON environment itself. GME, Cadena and Eclipse not only allow the user to use the newly defined DSL, but can also be extended with customized tools, on which an entire development environment, dedicated to a user-defined DSL, can be based.

Cadena is particularly concerned with component-based development, being able to deal with component reuse by design. Its three-tier architecture featuring kind, type and instance of a component, can be directly used to model the reuse aspect of a component-based framework. The instantiation is taken care of by the tool. It has a graphical editor supporting modelling, but currently the editor is not so convenient and flexible to use, and customization of the editor (i.e. customized shape of component) is not provided by the tool. Moreover, only the cardinality constraints of relationships are supported in Cadena. In the future, when this tool is further developed and a facility supporting more complex constraints definition is provided, it could become the foundation of our COMDES toolset, as it is based on the Eclipse and EMF framework, and tools for model-to-model and model-to-text transformation can be implemented using projects from the Eclipse world.

MetaEdit+ is a commercial meta-modelling product, which offers a Symbol Editor facilitating the customization of visual modelling effects and a promising code generation tool for easy automatic synthesis of code and documentation. However, the meta-modelling process in MetaEdit+ is not as straightforward as that in GME or Eclipse EMF, and moreover, constraints definition is limited.

GME was the tool used to prototype the COMDES domain-specific language

in its early stage of development [52]. GME enables a powerful meta-modelling capability by providing a number of unique meta-modelling concepts, such as sets, references and aspects, etc.; in addition, the OCL language is fully implemented. Automatic synthesis of programs is also supported by GME through user-defined plug-ins and the BON API.

A deficiency of GME is that it is difficult to dynamically change the graphical representation of models due to the adopted (fixed) Model-View-Controller architecture [52]. The customization capability is limited without modification of the source code of the tool. For example, connection class is used to express a relationship between two objects in GME and the connection can only have attributes. No other classes, such as atoms and models can be contained in the connection. Attributes are values of predefined simple types, such as integer, string, Boolean and enumeration. Consequently, if a connection is used in the meta-model to express an association, the associations modelled using the connection primitive will be visualized as a line between the objects, and the line can only have defined attributes. However, in certain cases, a model line needs to contain other models with complicated structures. And there is no way for meta-modelling to satisfy this kind of requirement, without modifying the tool.

Among the tools discussed in this chapter, the Eclipse platform is the one we selected to implement the software engineering environment for COMDES. One of the reasons is that it supports strong meta-model definition, GUI definition and flexible constraint definition languages. A complex graphical DSL usually requires several kinds of diagram to specify an embedded system. Moreover, in each diagram models need to be checked against constraints, which is sometimes not possible or cumbersome using only OCL. The Eclipse provides a validation framework that allows for constraints to be checked using Java. Based on the EMF generated meta-model implementation classes, parsing models is quite easy using either tailored API or reflective API. Besides JET, there are other third party tools available for code generation. Although model reuse is not supported by default, this feature can be implemented at the meta-model level (see Chapter 4 – Section 4.3.1). With the support of various modelling projects, some baseline of tools can be generated automatically, if the meta-model of a DSL has been defined. Tools can be easily built and integrated into the Eclipse platform through its plug-in mechanism. Moreover, it allows for collaboration between heterogeneous development tools by providing a tool integration solution. Specifically, ModelBus [53] is a model-driven tool integration framework that can be used to build a seamlessly integrated tool environment for a given development process.

The presented platforms support a broad spectrum of solutions for building

MDSD tools with respect to CBD. But after the introduction of the COMDES framework and its development process in the following chapters, it will be seen that building the COMDES development environment is a complex problem, which cannot be directly addressed by the presented technologies, as the corresponding platforms are not directly targeted at the COMDES development process. Therefore, to achieve our goal, we need a number of technologies and tools as foundations, which will be described in more detail after the the COMDES framework and development method have been introduced.

Chapter 3

Domain-Specific Modelling Language: the COMDES Framework

Nowadays, embedded software development is still dominated by conventional design methods and manual coding techniques. However, these are not able to cope with continuously growing demands for high quality of service, reduced development and operational costs, reduced time to market, as well as ever growing demands for software safety and dependability. In particular, software safety is severely affected by design errors that are typical for informal design methods, as well as implementation errors that are introduced during the process of manual coding.

This situation has stimulated the development of new software design methods based on formal design models (frameworks) specifying system structure and behaviour, which can be verified and validated before the generation of the program code [10][1]. Furthermore, model-driven development can be combined with component-based design, whereby design models are implemented by means of reusable and reconfigurable components. Thus, embedded applications can be configured using repositories of prefabricated and validated components (rather than programmed), whereby the configuration specification is stored in data structures containing relevant information such as component parameters, input/output connections, execution sequences, etc. Hence, it is possible to reconfigure applications by updating data structures rather than reprogramming and reloading the entire application.

The main problem that has to be addressed with this method is to develop a *comprehensive*, yet *intuitive* and *open* framework for embedded systems. There are a considerable number of frameworks developed in the traditional Software

Engineering domain that employ components with operational interfaces as well as various types of port-based objects, e.g. actor frameworks [54][55][56][14][57]. However, it can be argued that the architecture of the framework (i.e. models used to specify component functionality, interfacing and interaction) should be derived from areas such as Control Engineering and System Science, taking into account that modern embedded systems are predominantly control and monitoring systems. This approach has been used for some time with industrial control systems, whose software is built from component objects *(function blocks)* that implement standard application functions and interact by exchanging signals. Accordingly, function blocks are 'softwired' into function block networks that are mapped onto real-time control tasks, e.g. standards *IEC 61131-3* [7] and *IEC 61499* [58].

Unfortunately, this is a relatively low-level approach, which is inadequate for modern embedded applications. These vary from simple controllers to highly complex, time-critical and distributed systems featuring autonomous subsystems with concurrently running activities (tasks) that have to interact with one another within various types of distributed transactions. The above standards do not provide modelling techniques and component definitions at this level and do not define concurrency, whereby the mapping of function block networks on real-time tasks, as well as task scheduling and interaction are considered implementation details that are not a part of the standard.

In order to overcome the above problems, the Control Engineering models must be augmented with concepts and techniques developed in the Computer Science domain (concurrency, scheduling, communication, state machines, etc.), as advocated by leading experts in the area of Embedded Software Design, e.g. [1][59]. The resulting framework must support compositionality and scalability through a well-defined hierarchy of reusable and reconfigurable components, including both actors and function blocks (FBs). On the other hand, it has to adequately specify system behaviour for a broad range of sequential, continuous and hybrid control applications.

These guidelines have been instrumental in developing the COMDES framework [19] and its follow-on version COMDES-II [31][20]. This is a domain-specific framework for time-critical distributed control applications, featuring a hierarchical component model, as well as transparent signal-based communication at all levels of specification. In COMDES-II, an embedded application is composed from actors, which are configured from prefabricated function blocks. This is an intuitive and simple modelling technique that is easy to use and understand by application experts, i.e. control engineers.

So far, COMDES-II design models have specified informally using graphical

and textual notations [31][20]. Previous research has also addressed the problem of developing meta-models of COMDES components using the modelling notations of GME [29][30]. However a complete meta-model of the framework has not been developed. This problem has been now addressed in the context of the Eclipse framework and its EMF modelling language (see Chapters 4 and 5). However, developing a complete meta-model requires a precise and unambiguous specification of design models constituting the domain-specific language of COMDES-II. This can be best accomplished by developing a formal (mathematical) specification of the framework.

The formal specification of a component-based framework such as *COMDES-II* is a complex task, which must be addressed systematically, having a clear idea of its objectives. Specifically, it is necessary to provide an answer to the following questions: *what* aspects of the framework are to be specified; *how* is that to be achieved (i.e. what techniques should be used) and ultimately – *why* is it needed, i.e. what is the purpose of the exercise?

The formal specification of an embedded application in *COMDES-II* must consider two interrelated aspects, in the context of component-based design of embedded software:

- Specification of system structure
- Specification of system behaviour

In other words, it is necessary to specify an application as a composition of prefabricated components, following precisely defined rules and constraints. This must result in a set of diagrams that provide an adequate and unambiguous description of the application, much in the same way as circuit diagrams specify hardware applications. On the other hand, it is necessary to precisely specify the operation of the application, implemented as a composition of "*trusted*" components having precisely defined behaviours in the functional and timing domains. This must be done in accordance with the modelling techniques and principles of operation of *COMDES-II*, i.e. executable components modelling integrated circuits, component hierarchy and composition, clocked synchronous model of computation, etc., allowing for the extensive use of the principle of *separation of concerns* [31][20].

One important observation is that the specification of system behaviour must be consistent with, and follow from the specification of system structure. In particular, when following a component-based approach, an application is conceived as a composition of prefabricated components; hence, it is natural to specify its behaviour as a composition of component functions. In the case of *COMDES-II*,

system structure is described by hierarchical data flow models, i.e. function block and actor networks, whereby the execution of components (function blocks and actors) is modelled by the corresponding functions – from input signals to output signals. Consequently, system behaviour can be specified by one or more composite functions representing signal transformations – from system input to system output signals, which are executed in response to timing or external events. However, the detailed specification of some component functions may require an operational specification, e.g. the state transition function of state machine components [60].

Precise specification of system structure and behaviour is a precondition for developing a domain-specific methodology that will ultimately make it possible to configure component-based embedded applications that are *correct by construction*. In particular, precise behavioural specification is a prerequisite for the development of appropriate analysis methods that will be used to assess the behaviour of the configured application. This can be done using either the original design models or equivalent analysis models derived through some kind of semantics-preserving model transformation, e.g. *COMDES – Simulink* and *COMDES – Uppaal* transformations [61].

The above guidelines have been used to develop the formal specification of *COMDES-II*, which is presented in the following sections. The rest of the chapter is organized as follows: Section 3.1 presents a top-down specification of system structure in terms of data flow models describing actors and actor interactions, as well the internal structure of actors, which are composed of prefabricated function blocks. Section 3.2 presents a bottom-up specification of system behaviour starting with function block behaviour, followed by actor behaviour and finally – system behaviour. These are defined as composite functions specifying signal transformations – from input to output signals – of function blocks, actors and the system itself, respectively. Section 3.3 discusses the implications of the proposed framework for the software development process, which is ultimately aimed at designing systems that are correct by construction. The concluding section summarizes the main features of the proposed framework and their implications.

3.1 Specification of system structure

3.1.1 *COMDES-II* design models – an introduction

In *COMDES-II*, an embedded system is conceived as a composition of active objects (actors) that communicate by exchanging labelled state messages (signals). Communication is transparent, i.e. independent of the allocation of actors onto

network nodes. Accordingly, the system structure can be represented by an actor network – a data flow model specifying system actors and the signals exchanged between them (see e.g. Fig. 3.1).

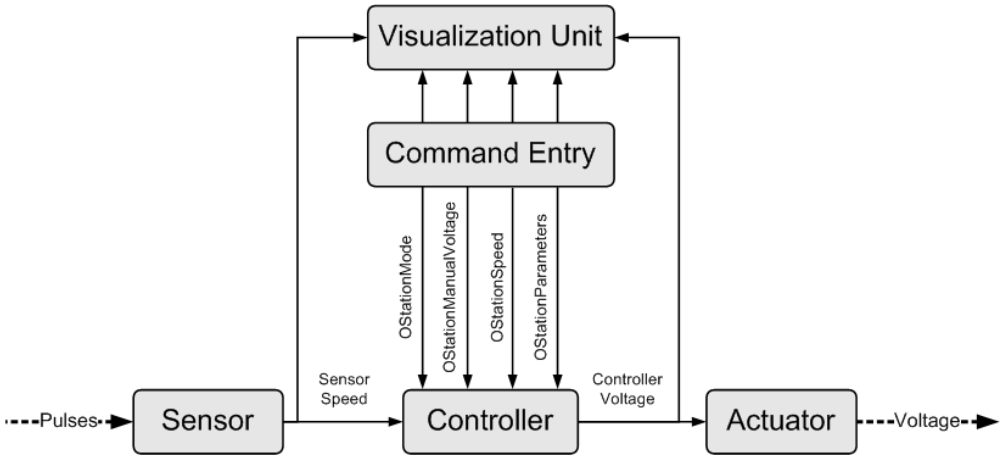


Figure 3.1: *COMDES-II* actor network – an example: *the DC Motor Control System*

An actor is modelled as an integrated circuit consisting of a signal-processing block, which is mapped onto a non-blocking (basic) task, as well as input and output signal drivers that are used to exchange signals with other actors and the outside world (see Fig. 3.2). Actor tasks are configured from function blocks and are modelled by function block networks. A function block is a *reusable executable component* that may have multiple instances within a given configuration. There are four kinds of function block: basic, composite, state machine and modal function blocks that can be used to implement a broad range of sequential, continuous and hybrid applications.

Basic function blocks have simple stateless behaviour, which is specified by functions defining signal transformations – from input signals to output signals (e.g. a PID controller function block). Complex stateful behaviour is implemented with modal function blocks (MFBs). These may be viewed as a generalization of stateless function blocks: a MFB has a number of operational modes where each mode is associated with one or more FB instances used to execute the corresponding control action. A modal function block receives indication of current mode from a supervisory state machine (SSM), whereby it executes the corresponding action, in the context of a continuous or sequential control actor, e.g. *manual/automatic* control of DC motor rotation speed (see Fig. 3.3). A function block network may be encapsulated into a *composite function block*, which can be

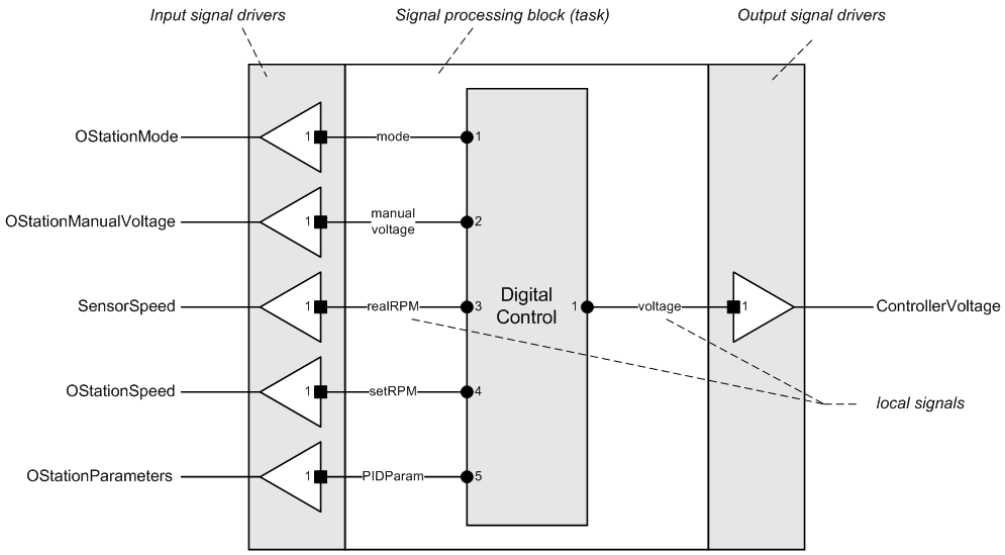


Figure 3.2: *COMDES-II Controller* actor

subsequently reused as an integral component.

Signal drivers are a special class of component – these are wrappers providing an interface to the system operational environment by executing kernel or hardware-dependent functions. Specifically, signal drivers can invoke kernel primitives to transparently broadcast and receive signals, independent of the allocation of sender and receiver actors on network nodes [62].

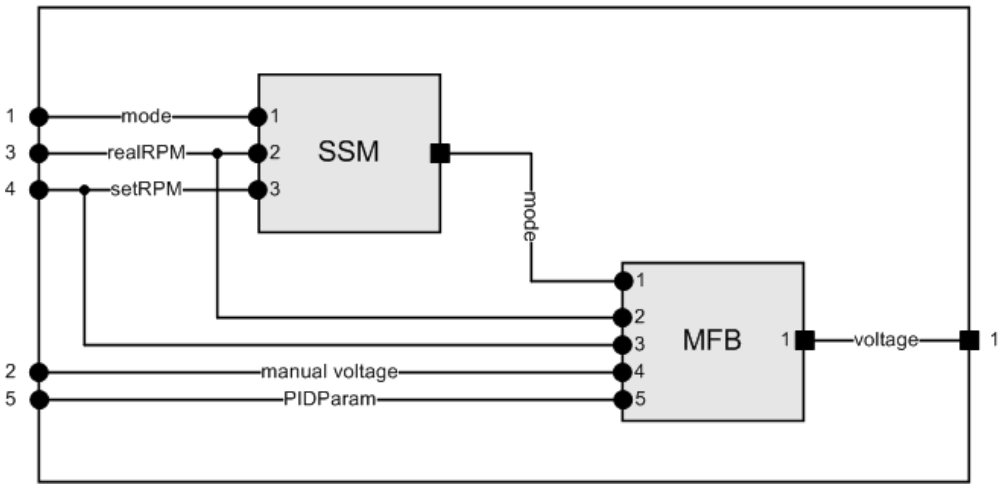


Figure 3.3: The *Digital control task* composed of state machine and modal function blocks

A detailed informal description of the above component models is given elsewhere [20]. The following discussion presents a formal specification of *COMDES-II* components and component configurations. The latter takes into account the two levels of the framework, i.e. system and actor levels, which are treated in a top-down fashion. At the top level, the system is described as an *actor network* – a data flow model involving system actors and the global signals exchanged between them, as well as a definition of the signals in terms of identifiers and constituent signal variables. At the next level, each system actor is described by a *function block network*, i.e. a data flow graph involving constituent function blocks and the internal signals exchanged.

In the broad sense, the COMDES framework consists of the four parts defined in this thesis (Chapter 4): domain-specific modelling language, executable components, run-time environment and platform. In the narrow sense, COMDES means only the domain-specific modelling language: it has two major versions, *COMDES-I* [29] [19] [63], and *COMDES-II* [30] [31] [64].

Under *COMDES-I*, the distributed embedded application is conceived as a composition of subsystems, i.e. function units, such as sensor, controller, actuator, etc., where each function unit encapsulates a number of active objects (activities) that execute separate threads of control within the function unit. However, function units are modelled as software integrated circuits that are statically allocated onto network nodes, whereby it is not possible to dynamically reconfigure a function unit or to allocate its activities onto several network nodes [31]. *COMDES-II* adopts an actor-based model, whereby a system is composed of actors encapsulating a single thread of control that communicate with each other by exchanging labelled messages (signals). This model offers greater flexibility, since actors may operate autonomously or be grouped into logical subsystems, independent of their physical allocation.

COMDES-I defines a state machine that is capable of executing function blocks and/or function block sequences, specified by function block networks, within different states/modes of operation. This is essentially a hybrid state machine, which combines the reactive and transformational aspects of component behaviour. *COMDES-II* employs a master-slave model, whereby modal continuous behaviour is specified with two types of component, i.e. a supervisory state machine, which is coupled to one or more modal function blocks. Such a model reduces the complexity of the state machine due to separation of concerns: in that case reactive behaviour is realized by the state machine component, whereas transformational behaviour is delegated to the modal function blocks. Furthermore, this model also offers the possibility of distributed allocation of the components

making control decisions and executing the corresponding control actions.

3.1.2 Distributed control system specification

A distributed embedded control system (ECS) is modelled as an actor network:

$$ECS =< \boldsymbol{A}, \boldsymbol{S}, \boldsymbol{C} >, \tag{3.1}$$

where $\boldsymbol{A}$ is the set of system actors, $\boldsymbol{S}$ is the set of system signals and $\boldsymbol{C}$ is the set of channels used to exchange signals between actors. The set of system actors $\boldsymbol{A}$ consists of environment actors $\boldsymbol{A_{env}}$ modelling the plant, and control actors $\boldsymbol{A_{con}}$ operating in a distributed system environment:

$$\boldsymbol{A} = \boldsymbol{A_{env}} \cup \boldsymbol{A_{con}}. \tag{3.2}$$

The set of system signals $\boldsymbol{S}$ can be represented as:

$$\boldsymbol{S} = \boldsymbol{S_{in}} \cup \boldsymbol{S_{com}} \cup \boldsymbol{S_{out}}, \tag{3.3}$$

where $\boldsymbol{S_{in}}$ is the subset of physical input signals, $\boldsymbol{S_{com}}$ is the subset of signals (messages) exchanged over a communication network, and $\boldsymbol{S_{out}}$ is the set of output physical signals. Furthermore, $\forall s_i \in \boldsymbol{S} : s_i =< Id_i, SV_i >$, where Id_i is a signal identifier and SV_i is a set of signal variables defined in terms of variable names and the corresponding data types:

$$SV_i = \{< name_1^i : type_1^i >, < name_2^i : type_2^i >, \cdots, < name_{ki}^i : type_{ki}^i >\}, \tag{3.4}$$

e.g. signal *OStationParameters* consisting of PID parameters, such as proportional, integral and derivative gain values (see Fig. 3.2).

The communication relationship between actors is specified in terms of channels that are defined by a source – signal – destination relation:

$$\boldsymbol{C} \subset \boldsymbol{A} \times \boldsymbol{S} \times 2^A. \tag{3.5}$$

e.g. one of the channels depicted in Fig. 3.1, which is specified by the tuple $< Sensor, Sensor\_Speed, \{Controller, Vizualization\_Unit\} >$.

In an actual implementation, control actors will be allocated to network nodes, and channels – to the network communication channel and physical I/O channels. The subsequent discussion presents the internal structure of control actors. It assumes a real-time network with predictable message latency, such as CAN, which

has been used for the experimental validation of *COMDES-II*.

3.1.3 Control actor specification

A system control actor can be defined as:

$$a_{con} =< \boldsymbol{X}, L_{in}, N_{FB}, L_{out}, \boldsymbol{Y} > \tag{3.6}$$

where: $\boldsymbol{X}$ is the set of input signals received by the actor, $\boldsymbol{X} \subset \boldsymbol{S}$, L_{in} is an input signal latch, N_{FB} is a signal-processing network of function blocks, L_{out} is an output signal latch and $\boldsymbol{Y}$ is a set of output signals generated by the actor, $\boldsymbol{Y} \subset \boldsymbol{S}$.

The input latch is used to receive input signals and decompose them into input signal variables constituting the set V, which may be viewed as *local (internal) signals* that are processed by the function block network. The latter computes output variables constituting the set W, which are used to compose the output signals generated by the output latch (see e.g. Figs. 3.2 and 3.3).

The I/O latches are composed of communication objects called *signal drivers*, denoted as D^{in} and D^{out}. In particular:

$$L_{in} = \{D_i^{in}\}, \tag{3.7}$$

where D_i^{in} is an input signal driver, generating internal input signals $V_i, V_i \subset V$, corresponding to the constituent variables of input signal s_i^{in},

$$L_{out} = \{D_i^{out}\}, \tag{3.8}$$

where D_i^{out} is an output signal driver, accepting internal output signals $W_i, W_i \subset W$, corresponding to the constituent variables of output signal s_i^{out}.

The I/O latches are activated at the release and deadline instants of the actor task. This is a basic (non-blocking) task, whose internal structure is specified as a function block network performing the transformation of input signal variables into output signal variables: $V \rightarrow W$.

The FB network is modelled by an *acyclic* data flow graph (see e.g. Fig. 3.3), which can be defined as follows:

$$N_{FB} =< B, Z, Con >, \tag{3.9}$$

where B is a set of function blocks, Z is a set of FB network variables and Con is the set of FB network connections.

A function block performs the signal transformation $X \rightarrow Y$, where X is the set of FB input variables, $X \subset Z$, and Y is the set FB output variables, $Y \subset Z$. Specifically, a function block can be defined as:

$$FB =< X, Y, P, F > . \tag{3.10}$$

where X, Y and P denote input, output and persistent variables, respectively and F is a set of functions.

Input variables X are generated by input drivers or other function blocks, $X \subset Z$. These are used together with persistent variables to compute output variables Y, $Y \subset Z$. Persistent variables P represent the internal *state* of the function block, which is retained from one execution to the next, e.g. various types of controllers, filters, etc [7]. Simple function blocks may not have internal state, e.g. arithmetic function blocks, comparators, etc. Output variables are computed by functions $f \in F$ that are defined as $y = f(x, p)$, where $y \in Y, x \in X$ and $p \in P$. A detailed description of various kinds of FB is given in Chapter 4.

The variables constituting the set Z may be viewed as *local signals* associated with the function block network:

$$Z = V \cup I \cup W, \tag{3.11}$$

where the input signal variables V are generated by input drivers and processed by function blocks; internal variables I are generated and processed by function blocks; output signal variables W are generated by function blocks and used by output drivers to compose output signals.

FB network connections are used to wire function blocks with input and output signal drivers, and with each other. The corresponding set can be specified as a union of subsets denoting input, internal and output connections: $Con = Con_{in} \cup Con_{int} \cup Con_{out}$. These are defined as source – internal signal – destination relations as follows:

$$\begin{array}{ll} Con_{in} & \subset L_{in} \times V \times B, \\ Con_{int} & \subset B \times I \times B, \\ Con_{out} & \subset B \times W \times L_{out}, \end{array} \tag{3.12}$$

e.g. the connection represented by the tuple $< SSM, mode, MFB >$ shown in Fig. 3.3.

3.2 Specification of system behaviour

3.2.1 *COMDES-II* model of computation – an introduction

System operation is specified in terms of distributed transactions executed in accordance with a model of computation known as *Distributed Timed Multitasking* [31], which is presently supported by the distributed real-time kernel *HARTEX*μ [62]. The distributed transaction involves a number of actors that execute transaction phases by invoking sequences of function blocks within the corresponding actor tasks.

Actors interact with each other by exchanging labelled state messages (signals) using dedicated communication objects (signal drivers) that provide for transparent broadcast communication between the actors involved. Control actors interact with environment actors by exchanging physical signals via signal drivers that are triggered at precisely specified time instants, resulting in the elimination of transaction jitter.

Distributed Timed Multitasking (DTM) combines the concepts of Timed Multitasking [55] and transparent *signal-based* communication. With this model, it is assumed that signal drivers are short pieces of code that are executed atomically in logically *zero* time at precisely specified time instants, which is typical for control applications. Specifically, input signal drivers are executed when the actor task is released, and output drivers – when the task deadline arrives (see Fig. 3.4) or when the task comes to an end, if it has no deadline. Consequently, task I/O jitter is effectively eliminated as long as the task comes to an end before its deadline.

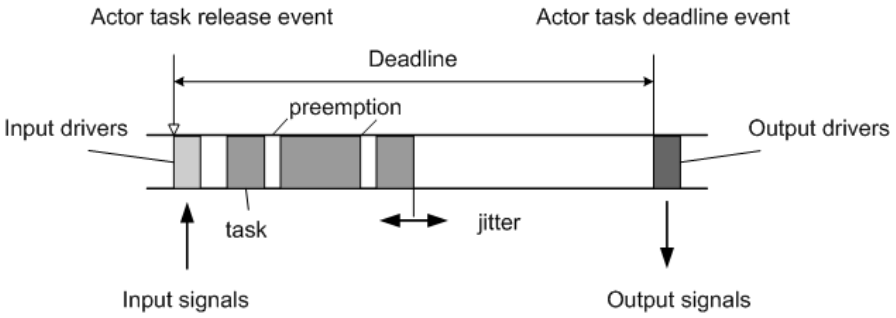


Figure 3.4: Actor execution under Distributed Timed Multitasking

Jitter-free operation can be extended to distributed systems, e.g. a phased-aligned transaction involving the actors *Sensor (S)*, *Controller (C)* and *Actuator (A)* from Fig. 3.1, which are triggered by a periodic timing event, such as a

synchronization (*sync*) message denoting the initial instant of the transaction period (T), with deadline $D \leq T$ (see Fig. 3.5). In this case, input and output signals are generated at transaction start and deadline instants, resulting in the elimination of transaction I/O jitter.

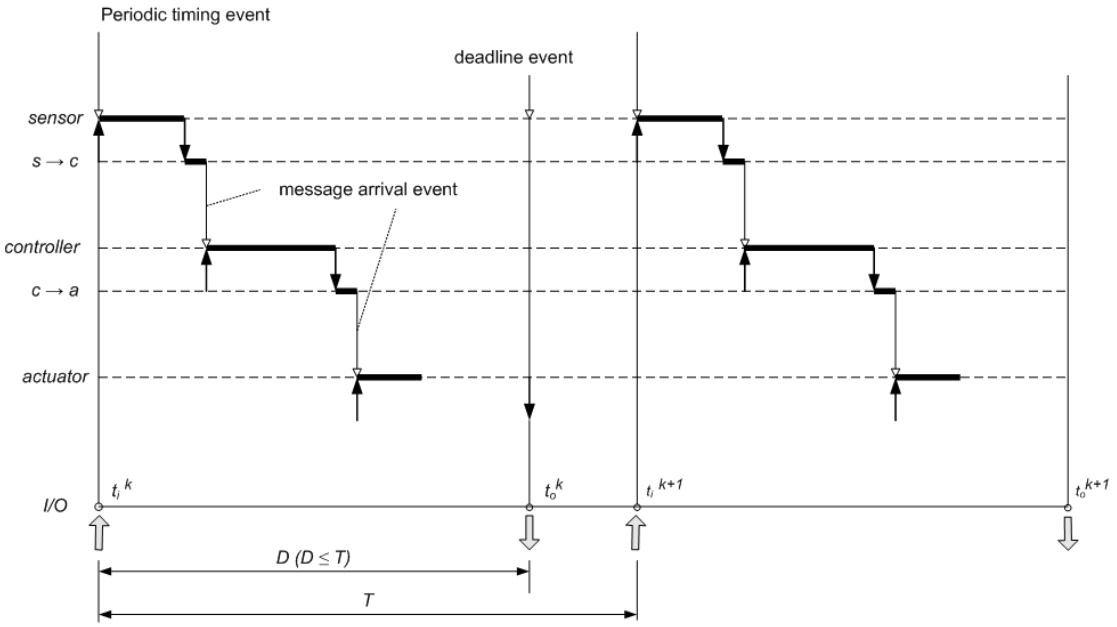


Figure 3.5: Jitter-free execution of distributed transactions

The following discussion presents a formal specification of system operation, taking into account the adopted model of computation and the model of system structure developed in the preceding section.

3.2.2 Specification of function block behaviour

Function block operation is specified with simple and/or composite functions from FB input variables $x(k)$ to FB output variables $y(k)$, $x \in X$, $y \in Y$, assuming periodic execution of system actors and constituent function blocks, which are invoked at time instants $kT, k = 1, 2, \cdots$, where T is the execution period of the host actor.

Basic function blocks implement standard signal-processing functions, such as: $y(k) = f(x(k))$ – with simple FBs implementing various kinds of mathematical operations, comparators, etc., or $y(k) = f(x(k), p(k-1), p(k-2), \cdots, p(k-l))$ – with function blocks having persistent state, where the state is defined in terms of one or more persistent state variables $p(k-1), p(k-2), \cdots, p(k-l)$, retained from

previous periods $1, 2, \cdots, l$ and updated during each period (as specified by the concrete FB algorithm, e.g. the discrete-time versions of filters, various control algorithms, etc. [7]).

A composite function block (CFB) encapsulates a FB network whose behaviour is specified with one or more functions such as: $y(k) = f(x(k))$, where f is a composite function specifying the transformation of signals from CFB inputs to CFB outputs, which is defined in terms of the functions executed by the constituent function blocks. Assuming that the CFB encapsulates a sequence of r function blocks where the output signals of a function block (except the last one) are input signals of the next one, this function can be represented as: $f = f_r \circ f_{r-1} \circ \ldots\ldots \circ f_1$, or using another notation: $y(k) = f_r(f_{r-1}(\ldots(f_1(x)))\ldots)$.

In the general case, this function will have a different expression for each particular configuration of the FB network, which has to be always specified by an *acyclic* data flow diagram. However, cycles are allowed at actor level but these are effectively broken by one-period delays due to the adopted clocked synchronous model of computation (see below).

The supervisory state machine implements the reactive aspect of actor behaviour, in separation from the transformational (signal processing) aspect, which is delegated to the modal function block. The SSM generates two output signals – s and u, meaning *state (or mode)* and *state-updated*, which are specified by the corresponding functions:

$$\begin{array}{lll} s(k) & = f(s(k-1), e(k), pr(e(k)) \text{ – a state transition function} & \\ u(k) & = \left\{ \begin{array}{ll} true & \text{– when } s(k) \neq s(k-1) \\ false & \text{– when } s(k) = s(k-1) \end{array} \right. & \text{– a Boolean function,} \end{array} \tag{3.13}$$

i.e. $u(k) = true$, when a state transition has taken place, and $u(k) = false$, when no transition has taken place.

In the above expression $e(k)$ denotes a *transition trigger*, i.e. an event specified as a Boolean expression involving binary input signals that are *present* at time kT, T is the period of the host actor, and $pr(k)$ is the priority of the event triggering the transition from $s(k-1)$ to $s(k)$.

The modal function block implements the signal processing aspect of actor behaviour by executing constituent function blocks within the corresponding modes of operation. These compute control signals $y_i, i = 1, 2, \cdots, r$, by invoking signal transformation functions $f_1, f_2, \cdots, f_r$ – from input signals to output signals. Subsets of these functions are selected for execution, depending on the *mode* (m) and *enabled* (b) input signals indicated by the state machine function block, such that:

$$
\begin{array}{ll}
\forall y_i \in A_p, & y_i(k) = f_i(x(k)), \text{ and} \\
\forall y_i \in A_q, q \neq p, & y_i(k) = y_i(k-1) \text{ – when } m(k) = p \text{ and } b(k) = true; \\
\forall y_i, & y_i(k) = y_i(k-1) \text{ – when } b(k) = false,
\end{array} \tag{3.14}
$$

where A_p is a control action, i.e. a subset of control signals y_i generated by the MFB in mode p and f_i is the function executed by the corresponding function block in order to generate the output signal y_i, $y_i \in A_p$. For instance, the control signal *voltage* of Fig. 3.3 will be generated by a PID function block if *mode* has been updated to *automatic*.

The composition of supervisory state machine and modal function block operates as a periodically executed event-driven state machine [60], whose operational semantics is informally presented below:

- The supervisory state machine is activated after the corresponding actor is triggered for execution by a periodic timing event or a signal arrival event.

- The supervisory state machine determines the current mode in response to a transition trigger (event), and if several transitions are possible, chooses the one triggered by the highest-priority event.

- In the absence of a transition trigger the state machine remains in the previous state (state/mode is not updated). A transition without a labelling signal is triggered by the clock signal, which is treated as an input signal that is always present.

- The modal function block executes a signal transformation function in order to compute one or more control signals associated with the current mode, if enabled by the *state-updated* signal.

- If the mode has not been updated, the MFB is not enabled and its output signals retain their previous values. Hence, *execute-once* semantics typical for event-driven state machines (hence no self-transition loops unless explicitly required).

The presented state machine model combines predictable execution typical for time-driven state machines with the inherent expressiveness of event-driven state machines, since diagram clutter due to self-transition loops is eliminated.

3.2.3 Specification of actor behaviour

Actors generate reactions to triggering events in the form:

$$\boldsymbol{e} \rightarrow \boldsymbol{Y} \tag{3.15}$$

where $\boldsymbol{Y}$ is the set of output signals generated by the actor in response to a triggering event $\boldsymbol{e}$. The latter may be one of the following: periodic timing event – either local timing event $\uparrow(kT)$ or global timing event $\uparrow sync(kT)$; external event $\uparrow \boldsymbol{x}_{trigger}$, where $\boldsymbol{x}_{trigger}$ is one of the actor input signals, e.g. a message arrival event or a user input event (see also the meta-model of Actor in Chapter 4) [1].

Actor output signals $\boldsymbol{y} \in \boldsymbol{Y}$ are specified by functions of input signals $\boldsymbol{x} \in \boldsymbol{X}$ that are latched by input drivers at the time of input t_{in}. With periodic actors triggered by local or global timing events $t_{in} = kT, k = 0, 1, 2, \cdots$.

Output signals are composed of output signal variables generated by the actor FB network, which has a zero *logical execution time*. Hence, the output signal variables are logically related to the input time instant kT:

$$w(k) = \psi(v(k)), \tag{3.16}$$

where ψ is a composite function – from input signal variables $v \in V$ to output signal variables $w \in W$ that constitute actor input signals $\boldsymbol{x}$ and output signals $\boldsymbol{y}$, respectively.

With simple actors having purely transformational behaviour:

$$\psi = f_n \circ f_{n-1} \circ \cdots\cdots \circ f_1, \tag{3.17}$$

where f_i are signal-transformation functions executed by the constituent function blocks, $i = 1, 2, \cdots, n$.

With complex actors built from supervisory state machines coupled to modal function blocks, each mode generates certain control signals specified by the corresponding functions, for example:

$$\begin{aligned} &w_1(k) = \psi^1(v(k)) \text{ – generated in mode 1} \\ &w_2(k) = \psi^2(v(k)) \text{ – generated in mode 2} \\ &\cdots\cdots \\ &w_l(k) = \psi^l(v(k)) \text{ – generated in mode } l \end{aligned} \tag{3.18}$$

[1] Bold letters denote actor-related variables.

In this case, for each $\psi^i, \psi^i = f_i \circ s$, where s is the state transition function and $f_i(v(k))$ is the signal transformation function executed by the modal function block when the supervisory state machine has indicated that $s(k) = i$.

In the general case:

$$\psi^i = f_i \circ s \circ g \tag{3.19}$$

where g denotes a preprocessing function. The latter is executed by a preprocessing (basic or composite) function block, generating a transition-trigger signal for the supervisory state machine (e.g. various types of arithmetic, comparators, counters, etc.)

The output variables generated by the actor task are used to compose output signals, which are latched into the output drivers at the time of output:

$$\begin{aligned} \boldsymbol{y}(t_{out}) &= \psi(\boldsymbol{x}(t_{in})), \\ t_{out} &= t_{in} + D = kT + D, k = 0, 1, 2, \cdots; 0 < D \leq T. \end{aligned} \tag{3.20}$$

Hence:

$$\boldsymbol{y}(kT + D) = \psi(\boldsymbol{x}(kT)), \tag{3.21}$$

and the actor as a whole has constant, *non-zero* logical execution time, i.e. clocked synchronous semantics [65].

In the special case of actor without deadline, it is assumed that $D = 0$, and $t_{in} = t_{out} = kT$. Hence: $\boldsymbol{y}(k) = \psi(\boldsymbol{x}(k))$, and the actor has a perfect synchronous semantics. This is the case with intermediate actors of phase-aligned transactions, where the deadline is usually associated with the last actor, which has to generate the control signal at the transaction deadline instant (see e.g. Fig. 3.5).

3.2.4 Specification of system behaviour

System operation is specified in terms of distributed transactions, such as the transaction shown in Fig. 3.5, assuming: 1) Periodic phase-aligned transactions involving non-blocking *basic* tasks, such as the one shown in Fig. 3.5, which are typical for distributed control applications [66]; 2)Non-blocking signal-based communication; 3)Distributed Timed Multitasking, which is an extension of Timed Multitasking for distributed transactions.

Under these assumptions, a periodic phase-aligned transaction with a period T_{trans} can be represented as a sequence of transaction phases, involving a number of actors, which are executed in response to a global timing event $\uparrow sync(kT_{trans})$

represented by the arrival of a synchronisation (*sync*) message generated by a sync master node:

$$\begin{aligned} &\uparrow sync(kT_{trans}) \rightarrow \boldsymbol{y}_1; && \boldsymbol{y}_1 = \psi_1(\boldsymbol{x}_1), \\ &\uparrow \boldsymbol{x}_2 \rightarrow \boldsymbol{y}_2; && \boldsymbol{y}_2 = \psi_2(\boldsymbol{x}_2), \\ &\cdots\cdots \\ &\uparrow \boldsymbol{x}_n \rightarrow \boldsymbol{y}_n; && \boldsymbol{y}_n = \psi_n(\boldsymbol{x}_n), \end{aligned} \tag{3.22}$$

where: $\boldsymbol{x}_1 = \boldsymbol{x}_{in}, \boldsymbol{x}_2 = \boldsymbol{y}_1, \cdots, \boldsymbol{x}_n = \boldsymbol{y}_{n-1}, \boldsymbol{y}_n = \boldsymbol{y}_{out}$.

Hence, transaction execution can be modelled with a composite function:

$$F = \psi_n \circ \psi_{n-1} \circ \cdots \circ \psi_1, \tag{3.23}$$

where ψ_i is the function implemented by the i-th actor, $i = 1, 2, \cdots, n$.

Taking into account Distributed Timed Multitasking, transaction execution can be represented as a transformation from input signals $\boldsymbol{x}_{in}(t_{in})$ to output signals $\boldsymbol{y}_{out}(t_{out})$, where t_{in} and t_{out} are determined by the transaction period T_{trans} and deadline D_{trans}:

$$\begin{aligned} &\uparrow sync(kT_{trans}) \rightarrow \boldsymbol{y}_{out}, \\ &\boldsymbol{y}_{out}(kT_{trans} + D_{trans}) = F(\boldsymbol{x}_{in}(kT_{trans})); D_{trans} \leq T_{trans} \end{aligned} \tag{3.24}$$

This can be explained in more detail with the example shown in Fig. 3.5, i.e. a phase-aligned transaction involving the actors *Sensor*, *Controller* and *Actuator*:

$$\begin{aligned} &Sensor : \uparrow sync(kT_{trans}) \rightarrow \boldsymbol{y}_1; \boldsymbol{y}_1(kT_{trans}) = \psi_1(\boldsymbol{x}_1(kT_{trans})) \\ &\boldsymbol{x}_1 = \boldsymbol{x}_{in}\text{: sensor signal; } \boldsymbol{y}_1\text{: process variable} \end{aligned} \tag{3.25}$$

$$\begin{aligned} &Controller : \uparrow \boldsymbol{x}_2 \rightarrow \boldsymbol{y}_2; \boldsymbol{y}_2(kT_{trans}) = \psi_2(\boldsymbol{x}_2(kT_{trans})) \\ &\boldsymbol{x}_2 = \boldsymbol{y}_1\text{: process variable; } \boldsymbol{y}_2\text{: control signal} \end{aligned} \tag{3.26}$$

$$\begin{aligned} &Actuator : \uparrow \boldsymbol{x}_3 \rightarrow \boldsymbol{y}_3; \boldsymbol{y}_3(kT_{trans} + D_{trans}) = \psi_3(\boldsymbol{x}_3(kT_{trans})) \\ &\boldsymbol{x}_3 = \boldsymbol{y}_2\text{: control signal; } \boldsymbol{y}_3 = \boldsymbol{y}_{out}\text{: actuator signal} \end{aligned} \tag{3.27}$$

In the above example:

- *Sensor* is a periodic actor with a period equal to T_{trans} and a zero deadline.
- *Controller* is effectively a periodic actor triggered by the arrival of the process

variable message $\boldsymbol{x}_2$, having zero deadline.

- *Actuator* is effectively a periodic actor triggered by the arrival of the control variable message $\boldsymbol{x}_3$, having a non-zero deadline equal to the transaction deadline D_{trans}. Hence, its outputs are latched at time instants $kT_{trans} + D_{trans}, k = 0, 1, 2, \cdots ; D_{trans} \leq T_{trans}$.

Consequently, the behaviour of the system can be represented by the function:

$$\boldsymbol{y}_{out}(kT_{trans} + D_{trans}) = F(\boldsymbol{x}_{in}(kT_{trans})), \tag{3.28}$$

where: $F = \psi_3 \circ \psi_2 \circ \psi_1$.

In the general case, the distributed system may consist of multiple subsystems executing distributed transactions with different rates of activation (multi-rate system), e.g. a multi-loop distributed control system. Accordingly, subsystem actors are allocated onto network nodes, and subsystem channels – onto the physical communication channel(s). This raises the issue of concurrent execution of transaction tasks/communications within the corresponding operational domains.

Following the adopted model of computation (Fig. 3.4), actor tasks are executed in a dynamic priority-driven scheduling environment provided by node-resident kernels, which are instances of the HARTEXμ timed multitasking kernel [62]. Communication takes place in a real-time network supporting predictable interactions, such as CAN. Transparent signal-based communication is supported by a dedicated protocol provided by the HARTEXμ kernel. With this protocol, signal drivers are executed *atomically* at precisely specified time instants that are fixed on the time axis. This makes it possible to eliminate the undesirable effects of task preemption and network communication, i.e. transaction I/O jitter, as long as transaction (end-to-end) response times are less than the corresponding end-to-end deadlines.

3.3 COMDES software development process

The presented software architecture has important implications for software safety and predictability, as well as the envisioned software development process, which is depicted in Fig. 3.6.

In particular, applications are configured from pre-validated (*trusted*) components, following strict composition rules that are derived from the syntax and static semantics of the framework. The behaviour of software components and applications is rigorously specified via a hierarchy of formal models that constitute

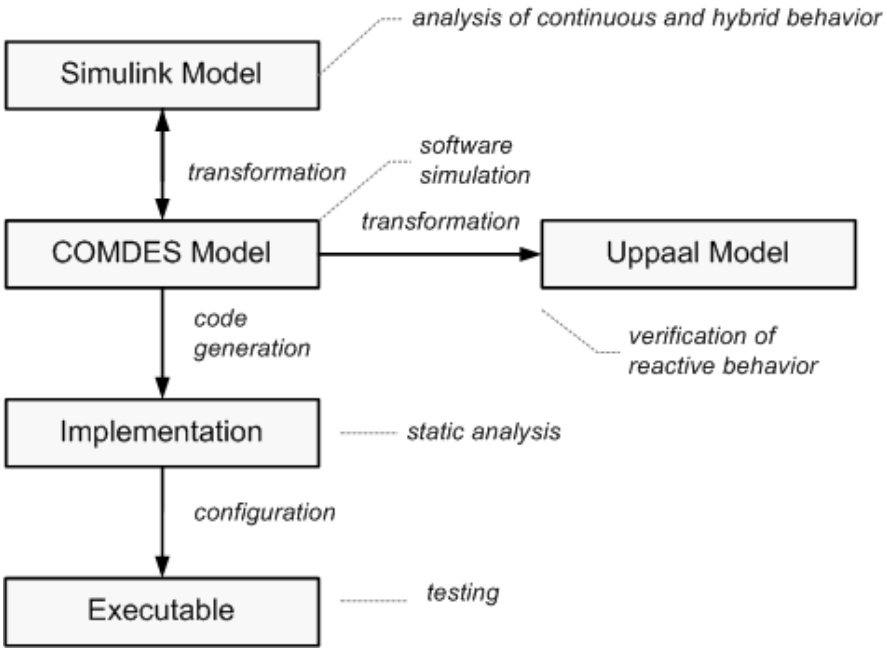


Figure 3.6: *COMDES-II* software development process

the behavioural semantics of the framework. On the other hand, the use of timed multitasking makes it possible to engineer highly predictable systems operating in a flexible, dynamic scheduling environment.

However, the configured applications must be proven correct with respect to the required functional and timing behaviour. This is facilitated by the principle of *separation of concerns*, which is widely used in the proposed framework, e.g. separate treatment of computation and communication, functional and timing behaviour, reactive and transformational behaviour, etc. Accordingly, system behaviour can be analysed using appropriate techniques and tools, following semantics-preserving transformations from system design models to the analysis models supported by standard tools such as *Simulink* and UPPAAL.

In particular, *Simulink* can be used to analyse the behaviour of predominantly continuous systems via numerical simulation, e.g. closed-loop and modal continuous control systems composed of signal-processing actors/function blocks. That is facilitated by the similarity between *COMDES-II* design models and *Simulink* analysis models representing the controller part of the system, both of which are discrete-time data flow models (function block networks). This can be done by converting a validated *Simulink* model into an equivalent *COMDES-II* model. Alternatively, it is possible to export a *COMDES-II* design model to the *Simulink*

environment, by wrapping *COMDES-II* components into S-functions and wiring them together [67][68], following the interconnection pattern of the original design model. However, it is necessary to model explicitly the latching of input and output signals, so as to precisely convey the timed multitasking semantics of the *COMDES-II* design model.

UPPAAL [69][70] is envisioned as a verification tool for systems exhibiting predominantly sequential reactive behaviour, which are usually modelled by state machines, and systems of interacting state machines. However, in this case *COMDES-II* design models are quite different from the analysis models (UPPAAL timed automata), which requires substantial model transformation in order to preserve the semantics of the original design model [61]. UPPAAL can also be used for the purpose of schedulability analysis. However, the separation of timing and functional behaviour, which is inherent to Distributed Timed Multitasking, makes it possible to assess transaction schedulability using numerical response-time analysis methods and tools, e.g. the method presented in [66].

The envisioned software development process will make it possible to engineer embedded applications that are *correct by construction*. This will hopefully eliminate design errors, which are difficult and costly to repair. On the other hand, implementation errors will be eliminated through an automated process of code generation and configuration that will be supported by an integrated development tool-chain (see Fig. 3.7), whose architecture and Eclipse-based implementation are investigated in this project. The elimination of both design and implementation errors will ultimately increase software safety, which is of paramount importance for the safety of the embedded application as a whole.

3.4 Summary

The chapter has presented a formal specification of *COMDES-II* – a domain-specific framework for distributed embedded control systems, which combines open architecture and predictable behaviour under hard real-time constraints. In this framework, the embedded system is composed from autonomous system agents (actors), which are configured from trusted prefabricated components, such as basic, composite, state machine and modal function blocks.

Actors interact by exchanging signals, i.e. labelled messages with state message semantics, rather than using I/O ports or operational interfaces. This feature facilitates system reconfiguration and provides for transparent communication between system actors, resulting in flexible and truly open distributed systems. Signal-based communication is also used for internal interactions involving con-

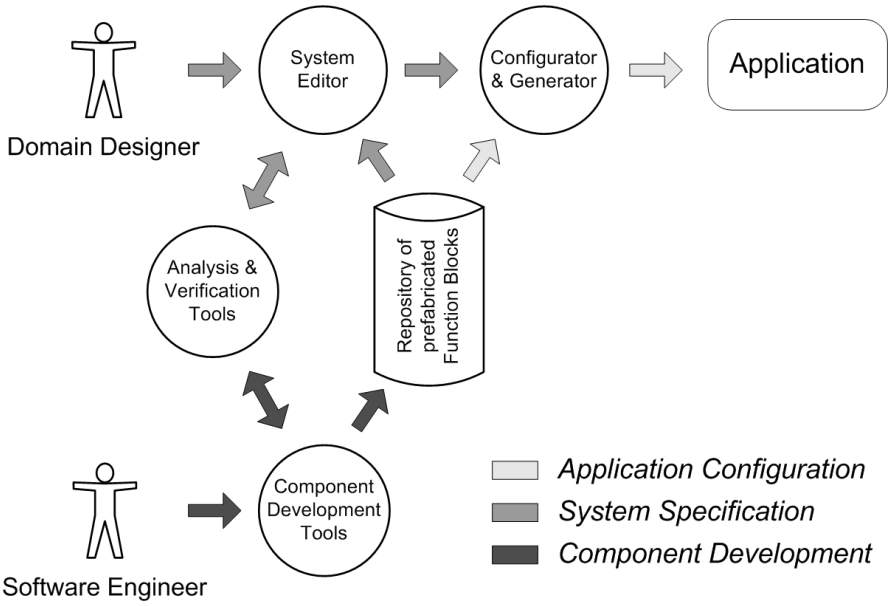


Figure 3.7: Overview of the development toolchian

stituent function blocks. That is why system configuration is specified by data flow models at all levels of specification. Consequently, actor behaviour is represented as a composition of component functions, and system behaviour – as a composition of actor functions.

A clocked synchronous model of execution is applied at actor and system levels, i.e. Distributed Timed Multitasking. With this model, input and output signals are latched at transaction start and deadline instants, respectively, resulting in constant, non-zero delay from inputs to outputs. Non-zero delay eliminates causality loops and fixpoint problems and is better suited to distributed operation. On the other hand, the latching of input and output signals results in the elimination of I/O jitter at both task and transaction levels. Consequently, this technique makes it possible to engineer highly predictable distributed systems while retaining the flexibility and ease of reconfiguration that are inherent to dynamically scheduled systems.

The above features make it possible to treat separately functional and timing behaviour at both actor and system (transaction) levels. Concurrency is separated from functionality too, being delegated to the actor and transaction levels. Likewise, it is possible to treat separately different kinds of functional behaviour, i.e. reactive (event-driven) and transformational (data flow) behaviour, which are delegated to separate components – supervisory state machine and modal function blocks.

Separation of concerns is expected to strongly facilitate system design and analysis, resulting in the development of embedded systems that are correct by construction. This will eliminate design errors, which are difficult and costly to repair. On the other hand, implementation errors will be eliminated through an automated process of configuration and code generation. In the end, the elimination of the basic sources of error will increase software safety, which is of paramount importance for the overall safety of embedded applications.

The formal design models presented in this chapter have been used to derive the meta-models of software components and the framework as a whole, which are presented in the following two chapters.

Chapter 4

Platform-Independent Model: the COMDES Meta-Model

COMDES is a framework intended for component-based development of software for embedded systems, and specifically – for control systems with hard real-time constraints. As a component-based framework, COMDES not only advocates the reuse of components at implementation level (code), but also supports the specification of systems at design level (model) from predefined component models. By following a model-driven software development methodology, a meta-model should be defined and used to specify relevant aspects of system structure and behaviour. Based on the meta-model, a number of models can be derived to fully describe a designed system, thus allowing for an application to be automatically synthesised.

This chapter presents an overview of the COMDES design method, followed by a detailed discussion of the COMDES framework, i.e. the COMDES meta-model – a set of models that are used to specify components and systems, developed under the framework, in a platform-independent manner. The meta-model describes formally domain models using a number of class diagrams in a bottom-up fashion. It is derived from, and consistent with the formal specification of the framework design models presented in Chapter 3. Where necessary, the description is accompanied by corresponding domain-specific models, in the form of COMDES graphical notations.

4.1 Overview of the COMDES software design method

A COMDES solution to a problem in the distributed real-time embedded control system domain is represented by four building blocks, which emphasize related aspects of the COMDES software design method (see Fig. 4.1): domain-specific language, run-time environment, executable function blocks and platforms. Additionally, repositories are used to store reusable components as well as applications.

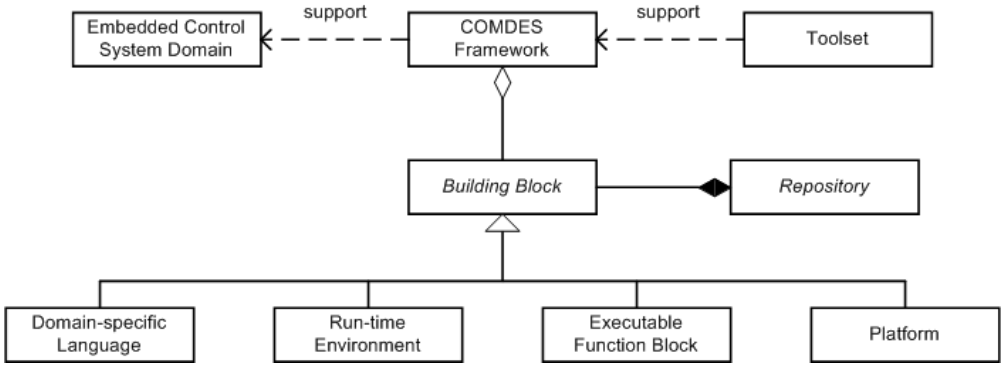


Figure 4.1: COMDES software design method

- The Domain-Specific Language is defined in terms of design models that are used to represent system structure as well as behaviour in a platform-independent way (see Chapter 3).
- The Run-time Environment provides mechanisms for concurrent execution of COMDES actors, inter-actor communication and event handling, in accordance with the adopted timed multitasking model of computation.
- Function blocks are used to implement the reactive and transformational behaviour of the system.
- Platforms provide a number of hardware architectures on which the system may be implemented.

A system modelled using the COMDES DSL is composed of function blocks and mapped onto objects provided by the run-time environment and platform. In particular, the COMDES model of concurrency requires that the run-time environment is implemented as a timed multitasking kernel, and function blocks

are prefabricated for several processor architectures. Specifically, the platform-independent COMDES system model is related to an actual implementation (see Fig. 2) by means of:

- Platform: allocating actors onto physical nodes having appropriate means of communication and interaction with other nodes and the physical environment.

- Run-time Environment: mapping actors onto real-time tasks, global signals onto messages, and choosing the right scheduling scheme, etc.

- Function Block: configuring actors from prefabricated software components.

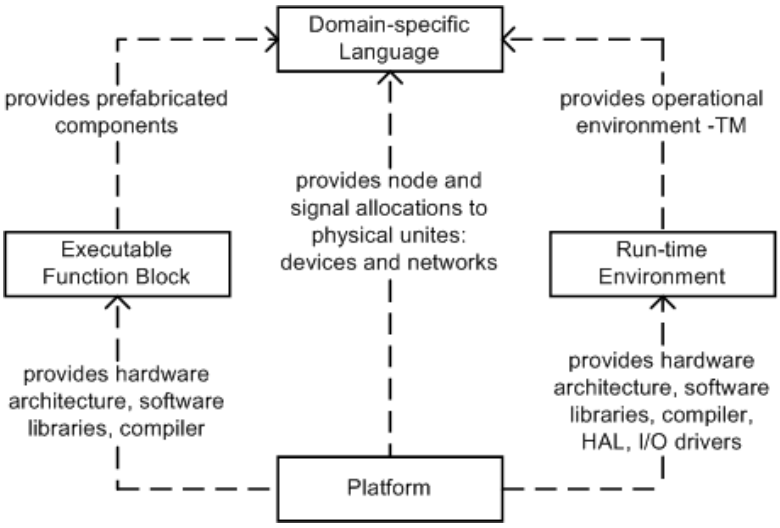


Figure 4.2: Building blocks of the COMDES solution

COMDES DSL: COMDES provides a domain-specific language specifying relevant aspects of system structure and behaviour within the domain of distributed real-time embedded control systems. A system design formulated in the DSL can be hierarchically constructed from a set of components without syntactical errors and ambiguity of semantics. A COMDES system in particular is composed from a number of actors that communicate by exchanging labelled messages (signals). Communication is transparent, i.e. independent of the allocation of actors onto network nodes. An actor consists of a signal-processing block configured from reusable components – function blocks, which is mapped onto a non-blocking task, as well as input and output signal drivers that are used

to exchange signals with other actors and the outside world. The COMDES system model can be seen as a high-level specification concerning the issues such as functionality, concurrency, interaction, and so on, from a collection of predefined components. From a MDA point of view, system models correspond to the Platform-Independent Models.

Function block: A function block is a component class that may have multiple instances within a given configuration. Function block types (see Fig. 4.3) can be reused across multiple projects (e.g. filter, PID, etc.). They are instances of kind and can be instantiated when building an application. Types are stored in a component repository and define the behaviour of a component. When developers are building an application, they load the component types from a repository and instantiate them. Instances of the same type share the same behaviour but differ by the data structure. Typically, there are many instances of a component of a given type. Types are created by skilled software engineers using specific design patterns. In contrast, an application is configured by domain experts from already available components stored in the repository (see Fig. 3.7).

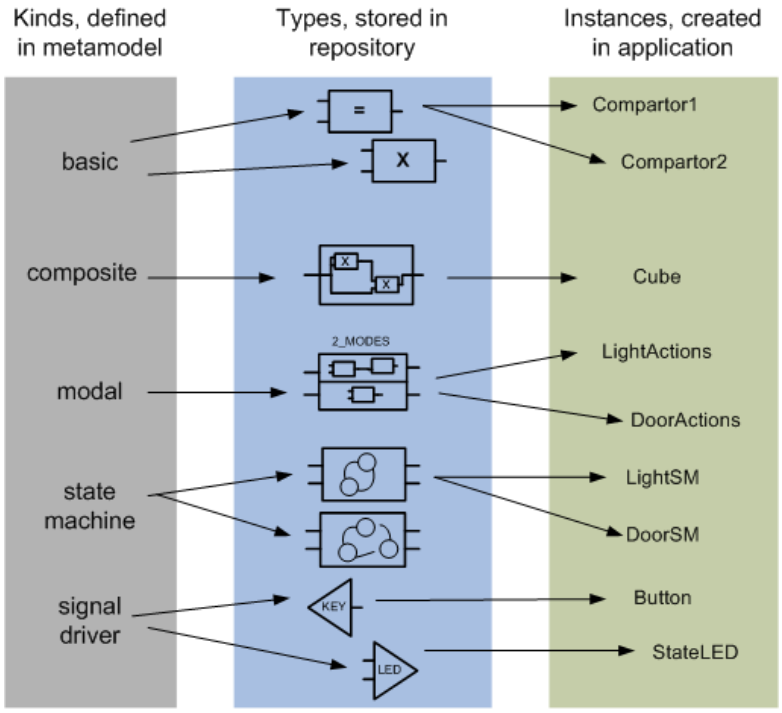


Figure 4.3: COMDES function blocks

Run-time environment: The functionality of a COMDES system will be implemented as a composition of reusable function blocks, whereas the timing as-

pects of the system will be managed by the underlying run-time environment – a real-time kernel HARTEXμ that implements the timed multitasking model of computation. HARTEXμ has been specifically developed to provide an operational environment for COMDES-based applications, and it can be characterised by a number of advanced features that are discussed in more detail in Chapter 5. Some of them have to be specifically highlighted, since they provide the functionality needed by COMDES applications:

- Preemptive priority scheduling of non-blocking basic tasks used to implement system actors. Split-phase execution of tasks and I/O drivers as required by the Timed Multitasking model of computation.
- Content-oriented message addressing: With this technique a message is addressed by its name, thus providing support for transparent signal-based communication.
- Stand-alone and distributed operation: Both modes of operation are supported making the kernel a versatile solution supporting a broad range of embedded applications.
- Extensive support for Timed Multitasking including both event-driven and time-triggered mechanisms used to implement this mode of operation.

The development of HARTEXμ kernel is not a task of this project, but a meta-model that can be used to define a HARTEXμ model, and then to generate code from kernel models, will be briefly presented in Chapter 5. With concrete information about the run-time environment, a COMDES system model can be transformed into a HARTEXμ model. HARTEXμ models correspond to the Platform-Specific Models in MDA.

Platform: Essentially, this part is responsible for providing hardware and fundamental software abstractions for function blocks, run-time environment, and eventually – COMDES system models written in the DSL. At the implementation level, function blocks are prefabricated components ready for reuse. To build a function block as an executable for different hardware platforms, it is necessary to have platform models providing information such as architecture, compiler, and fundamental software libraries (e.g. GNU glibc as C library defines the "system calls" and other basic facilities such as `open`, `malloc`, `printf`, `memcpy` ...), etc, for each platform. For the HARTEXμ kernel, an extra Hardware Adaptation Layer (HAL) must provide specific functions such as context saving and restoring, global disabling and enabling of interrupts, tick interrupt, idle mode operation, and network communication drivers. Physical input/output drivers can wrap the

HAL interface to a specific hardware resource. Finally, from the viewpoint of the COMDES DSL, the platforms are abstracted as networked nodes that host subsets of actors. With information concerning the hardware and software aspects of a platform, a final executable system can be generated and deployed on the target microprocessors, in either a stand-alone or a distributed environment. Obviously, the concrete system implementation corresponds to the PSM as well.

4.2 Domain-specific meta-modelling concepts and notations

A DSL comprises a set of concepts that are used to describe a specific class of systems. In COMDES, these concepts are derived with attention to the distributed real-time embedded control systems domain, and are substantiated as a number of models dealing with dedicated problems: computation, concurrency, system partitioning, etc. The concepts that describe a DSL are commonly defined using a meta-model, which is a model that specifies how models can be constructed in the design modelling language.

Fig. 4.4 illustrates the meta-modelling approach used to develop the COMDES framework, where a meta-meta-model specifies the COMDES domain-specific meta-model that is in turn used to specify reusable component models as well as application models in the particular domain. In particular, the meta-modelling approach supports component-based software development by defining different kinds of component as a part of the COMDES meta-model according to concepts from the domain. As a result, component types can be specified by the COMDES meta-model and stored into repositories. The types will be instantiated to compose application models. Component types, instances and applications are located at the M1 layer of the MOF four-layer architecture (see Fig. 1.4).

The Ecore provides a meta-meta-model as one of the MOF implementations. It is located at the M3 layer of the MOF four-layer architecture. In Ecore, a meta-model can be defined by a set of concepts like ***EClasses***, ***EAttributes***, ***EReferences***, ***EOperations***, etc. A simplified subset of the Ecore model is shown in Fig. 4.5, which shows only parts of the key Ecore concepts. For a full description of the Ecore model, please refer to [71]. Most of the modelling concepts that Ecore defines should be quite straightforward to those who are familiar to UML or object-oriented design. In this section, Ecore is examined in a small detail, with the aim of enabling readers to understand effectively meta-models defined for COMDES.

EClass is used to model classes themselves. Classes are identified by *name* and

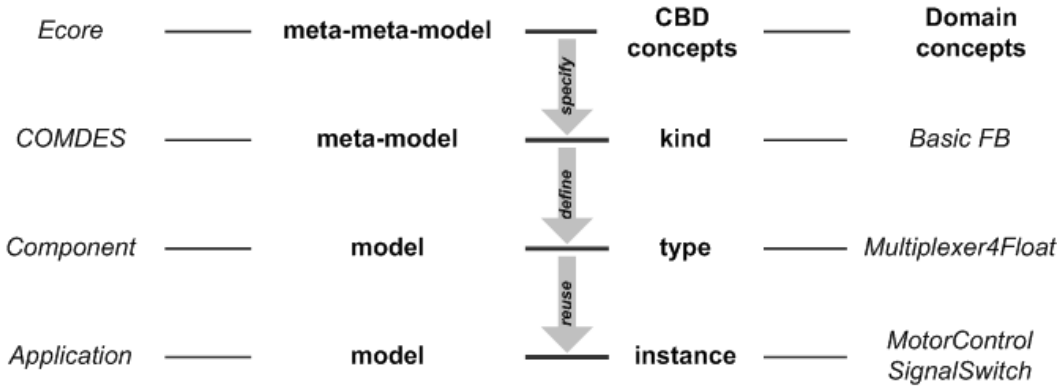


Figure 4.4: MDSD approach for building applications in COMDES

can contain a number of attributes and references. To support inheritance, a class can refer to a number of other classes as its super types. The two attributes defined by ***EClass*** itself can be used to specify the particular type of class being modelled. If *interface* is *true*, the ***EClass*** represents an interface that declares its operations and the accessors for its attributes and references, but provides no implementation for them. An interface cannot be instantiated. The implementation of an interface is modelled by including the interface in the *eSuperTypes* reference of a non-interface ***EClass***; that class will implement the operations and the accessors for the structural features declared by the interface. If *abstract* is *true*, the ***EClass*** represents an abstract class, from which other classes can inherit features, but which cannot itself be instantiated.

EAttribute models attributes, the components of an object's data structure. They are identified by *name*, and each of them has a data type.

EDataType models the types of attributes, representing primitive and object data types that are defined in Java (because the Ecore provided by the Eclipse EMF is implemented in Java). Data types are also identified by *name*.

EReference is used in modelling associations between classes; it models one end of such an association. Like attributes, references are identified by *name* and have a type. However, this type must be the ***EClass*** at the other end of the association. If the association is navigable in the opposite direction, there will be another corresponding reference. A reference specifies lower and upper bounds on its multiplicity. Finally, a reference can be used to represent a stronger type of association, called *containment*.

EOperation models the behavioural features of an ***EClass***. ***EOperations*** are contained by an ***EClass*** via the *eOpearations* reference. Notice, that *eOperations* is part of a bidirectional association, which allows an ***EOperation*** to easily

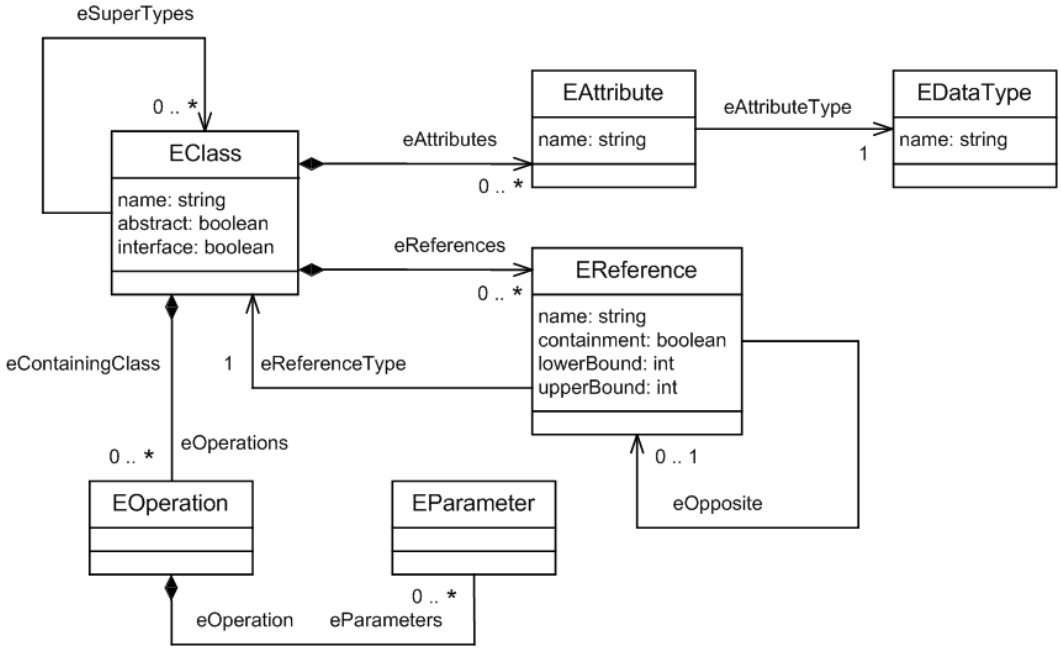


Figure 4.5: Core Ecore models in UML class diagram

obtain the ***EClass*** that contains it via the opposite reference – *eContainingClass*.

EParameter models the operation's input parameters. An ***EOperation*** contains zero or more ***EParameters***, accessible via *eParameters*. Again, this reference constitutes half of a bidirectional association; the ***EParameters*** can access the ***EOperation*** to which they belong via *eOperation*.

The Ecore features mentioned above are sufficient enough to define the COMDES meta-model. The meta-model can be represented as class diagrams with these concepts. Thus, UML notations for concepts like class, interface, attribute, and inheritance, etc. could be used to express the COMDES meta-model, as described in the following sections.

4.3 Meta-model of COMDES

4.3.1 Reuse pattern

In order to achieve component reuse, a *Kind-Type-Instance* pattern has been developed at the meta-model level (see Fig. 4.6a). Basically, the abstract ***ComponentKind*** class defines the kind of a component that is used to build concrete objects of the ***ComponentType*** class at the component development stage. Types are

stored in ***Repository*** for later reuse. During application development, objects of the ***ComponentInstance*** class are created, which have an internal structure identical to the structure of the ***ComponentType*** they refer to.

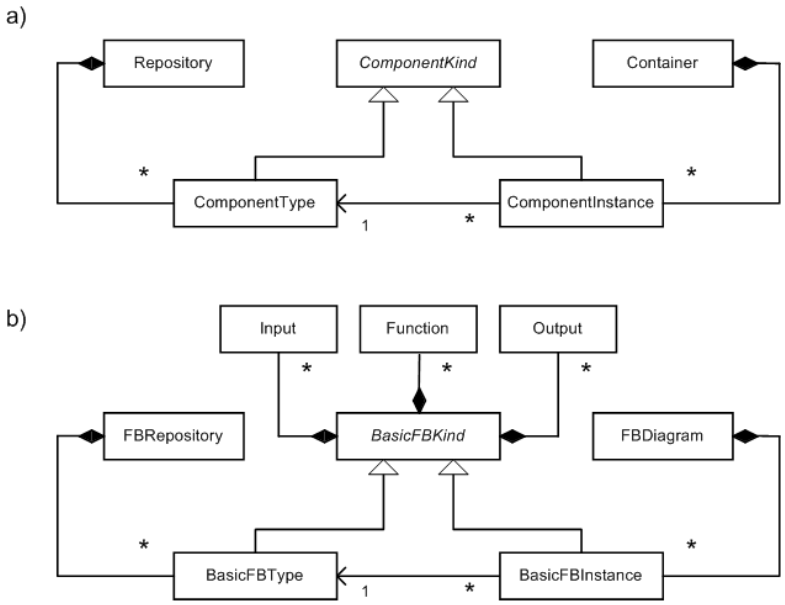


Figure 4.6: *Kind-Type-Instance* pattern

As an example, the Basic FB part of the COMDES meta-model is conceptually presented in Fig. 4.6b. The type (***BasicFBType***) and the instance (***BasicFBInstance***) classes inherit the kind (***BasicFBKind***), so that the instance (or the type) of that kind can get the definition of its structure. The instance can be instantiated in the ***FBDiagram*** whereas the type can be created in the ***FBRepository*** for reuse. In this manner, an instance can also be created at the application design time without a predefined component type model.

Therefore, there are two ways to create a component instance: 1) An instance can be created in an application using a predefined type stored in a repository. Then, the type's all internal elements are copied into the instance. 2) An instance can be created without a predefined type in an application. A designer is free to add inputs, outputs or internal elements into this instance. And if necessary, this instance can be exported to the repository and saved as a type, which increases the flexibility of the development process, at the cost of extra complexity added to the implementation of the development environment.

This approach is flexible in that it can be implemented in any object-oriented environment. It avoids the requirement for the tool implementation environment

to support run-time definition (construction) of classes [72], although that is supported in the Java implementation environment using the core reflection API of the language (or the dynamic EMF API of the Eclipse platform).

4.3.2 Function blocks

4.3.2.1 A generic meta-model of function blocks

According to the pattern of reuse, three abstract classes are firstly defined as foundations of all kinds of function blocks. These three abstract classes should be inherited respectively by other three concrete classes when defining each kind of COMDES function block.

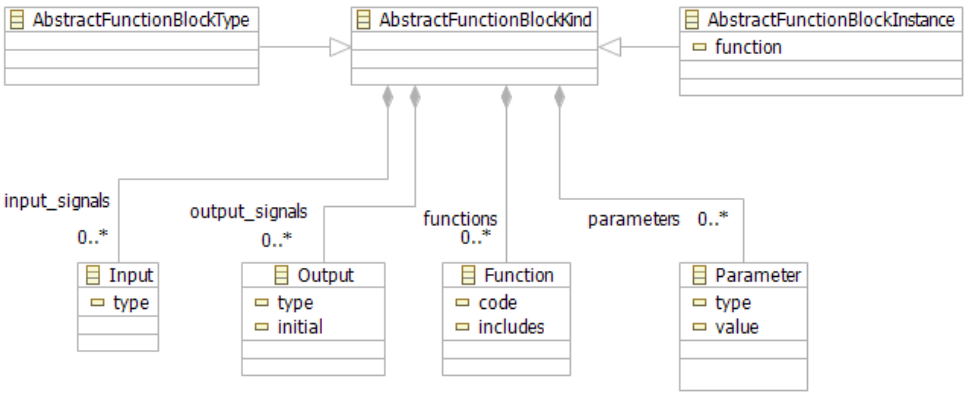


Figure 4.7: Generic function blocks

Fig. 4.7[1] shows a generic meta-model specifying the common architectural characteristics of all kinds of FBs in COMDES. Each kind of FB (***AbstractFunctionBlockKind***) contains functions which are defined by the class ***Function***.

The *code* of a ***Function*** should do the computational work for a function block. It could be specified either in a general-purpose language (e.g. the C language), which is directly used to implement the function block model, or in a special DSL that will have to be transformed into the language that implements the function block model. The first approach does not require a transformation step when

[1]In order to offer a clean presentation of the meta-model, some classes, which are not important for depicting the COMDES concept, have been omitted from the figures. For example, the majority of classes in the meta-model have an attribute "*name*" (i.e. ***Input***, ***Output***, ***Constant***, ***BasicInstance***, etc.), which is obtained by inheriting a class called "***NamedElement***" that defines the name attribute and is not shown on all figures in this thesis.

transforming model into code, but involves moving code-level decisions into the model without raising the abstraction level. It usually leads to contamination of the models with implementation concepts that are not derived by the domain expert, and therefore constitutes a potential source of errors. Alternatively, when using a DSL that belongs to the same domain, domain experts are spared the implementation concepts. However, designing such a DSL as well as related tools takes a considerable amount of time. Therefore, for time being, when such a DSL does not exist for COMDES yet, the *code* is specified in the C language.

The attribute *includes* of a function designates what library functions are needed by the function block, which is usually accessed by the *code*. The libraries are usually provided by platforms or run-time environment that function blocks are running on.

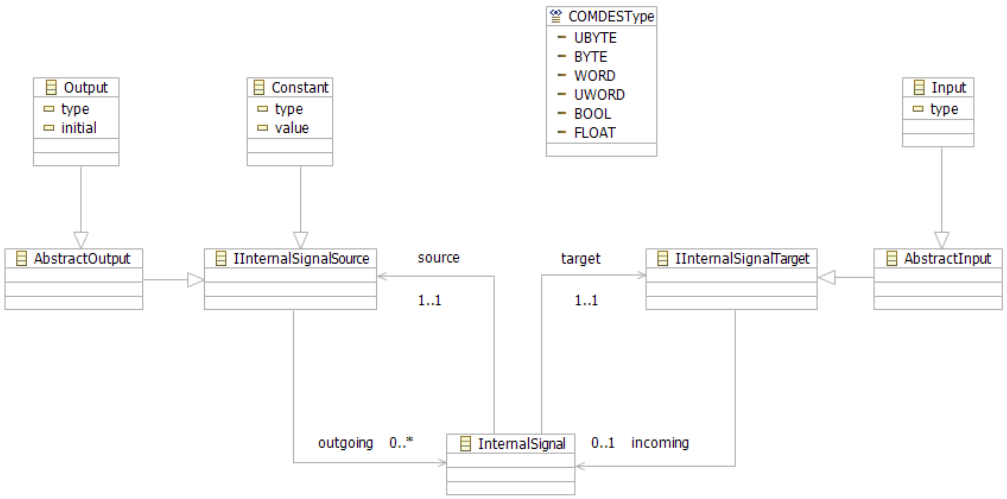


Figure 4.8: ***Input***, ***Output***, ***Constant*** and ***InternalSignal***

The ***AbstractFunctionBlockKind*** class contains parameters that are modelled by the ***Parameter*** class. Each parameter needs to be specified with data *type* and *value*. The data type of a parameter is defined by the attribute *type*. A parameter can be used as an input to the FB providing constant data for functions of a function block. Alternatively, it can be used as an internal variable that a FB function manipulates when running. In latter case, the *value* of a parameter becomes an *initial value* of the internal variable.

The attribute *function* of the ***AbstractFunctionBlockInstance*** specifies which function to execute during the invocation of a particular FB instance. Its value must be one of a list of available functions associated with the FB kind definition.

Each function block has inputs and outputs, which are modelled by the ***Input*** class and the ***Output*** class respectively. Normally, a function block should have both inputs and outputs. However, a driver – as a special kind of FB – has only one of them, i.e. an input driver FB has only outputs, whereas output driver FB has only inputs. The definitions of the input and output are depicted in Fig. 4.8.

In Fig. 4.8, the ***Input*** and the ***Output*** are connection points of ***InternalSignal*** which transfers data from source to target that are modelled by interfaces ***IInternalSignalSource*** and ***IInternalSignalTarget*** respectively. An internal signal source could be either an output or a constant (defined by the ***Constant*** class), whereas an internal signal target is an input. An output and a constant must have initial values, and their type is one of the ***COMDESType*** containing a set of primitive data types. However an initial value of an input can be obtained from the signal source to which that input is connected. One output can be connected to several inputs, whereas one input can only accept one incoming internal signal. A constant does not have an incoming internal signal as its value cannot be changed at run-time, obviously. According to COMDES FB design patterns in the C language (see Chapter 6), an input of a FB is implemented as a pointer to either an output of anther FB or a constant. Outputs and constants are implemented as variables.

4.3.2.2 Basic function blocks

Basic Function Block (BFB) is a fundamental component that implements transformational behaviour – it generates a set of output signals defined as functions of subsets of input signals, via the corresponding inputs and outputs. BFB implements specific process control functions, such as various types of variable preprocessing, conventional and advanced control algorithms, e.g. PID. Additionally, function can load/store a persistent piece of data modelled as a parameter that is used during the next invocation of the component.

The meta-model of basic FBs exactly follows the reuse pattern, as shown in Fig. 4.9. The ***AbstractBasicKind*** extends the ***AbstractFunctionBlockKind*** class, and thus aggregates classes of ***Input***, ***Output***, ***Function***, and ***Parameter***. The ***BasicType*** and the ***BasicInstance*** are concrete classes, which obtain their internal structures by inheriting from the ***AbstractBasicKind*** class. The ***BasicInstance*** class is associated to the ***BasicType*** class via the relationship *type*.

The Fig. 4.10 illustrates a BFB instance model named as *basic1* in its graphical concrete syntax, based on the meta-model. It has two inputs and one output. The function to execute is called *plus*, which is the only function defined in its function

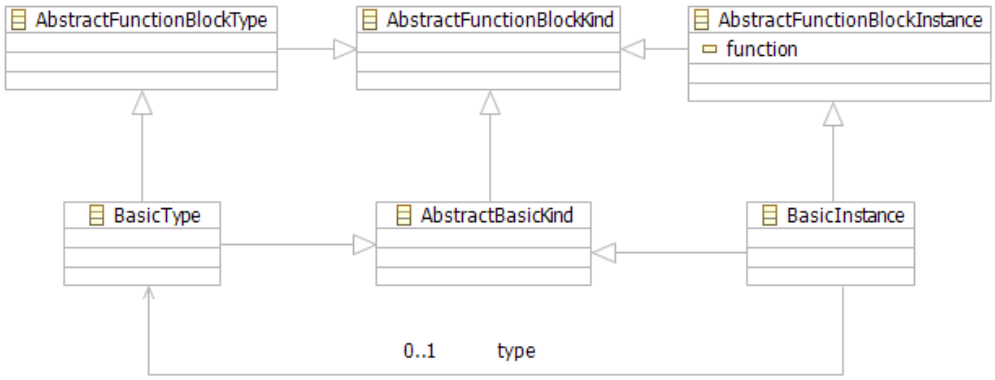


Figure 4.9: Meta-model of Basic FB

list. The function simply outputs the sum of two inputs, and the code of the function is implemented in the C language.

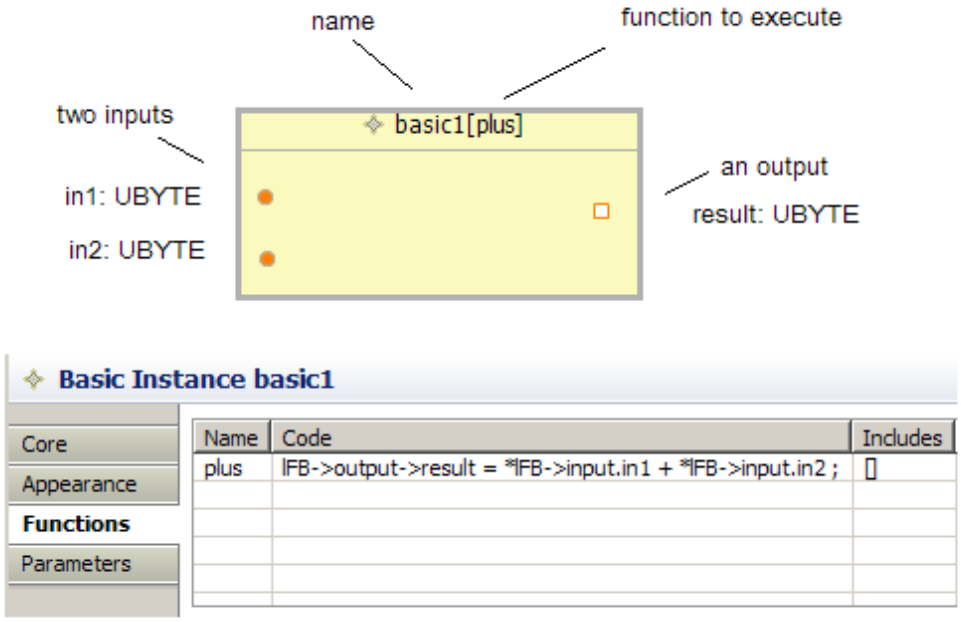


Figure 4.10: A basic FB model

4.3.2.3 Composite function blocks

Composite Function Block computes complex signal transformations from inputs to outputs, e.g. various signal processing functions, according to the synchronous data flow model [73][65]. In the meta-model of a CFB (Fig. 4.11), the ***AbstractCompositeKind*** extends the generic FB meta-model in order to obtain *inputs*, *outputs*, *parameters* and *functions*. (The other two abstract classes – for instance and type – from the generic FB pattern are not shown in this figure, in order to keep the figure clean and simple. These two classes will be neglected when describing the following FB kinds.) Additionally, the class also extends the ***FBDiagram*** class and consists of two extra relations – ***ExtendInput*** and ***ExtendOutput***.

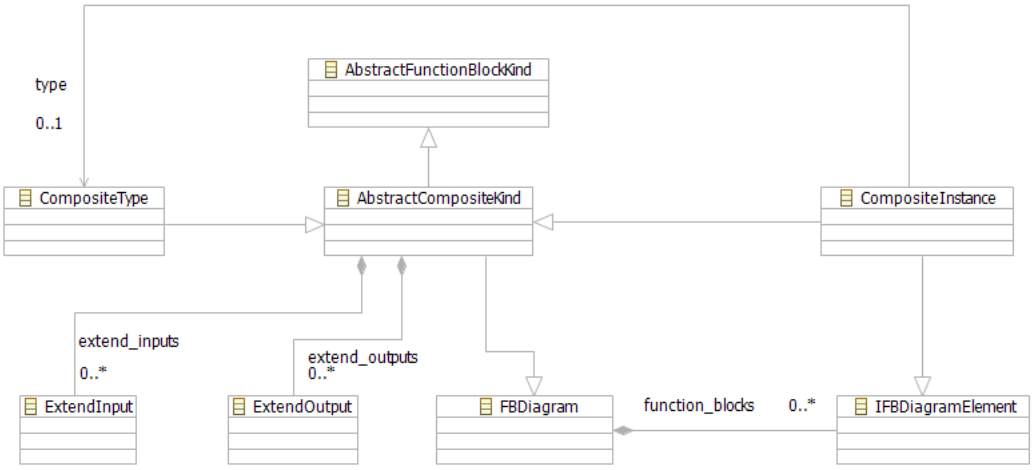


Figure 4.11: Meta-model of Composite FB

The Function Block Diagram (***FBDiagram***), as a directed graph, aggregates a number of internal signals and diagram elements (Fig. 4.12). The ***IFBDiagramElement*** is an abstract interface implemented by FB instances as well as constants. Data is exchanged between outputs and inputs of constituent FBs through internal signals. A FB diagram consisting of interconnected function blocks must be acyclic, which can be enforced by a constraint.

The use of ***FBDiagram***, ***IFBDiagramElement*** and their composition for meta-modelling the composite FB kind is actually inspired by one of the notable structural design patterns – the Composite Pattern [74], which allows for specification of hierarchical components. The hierarchy can be theoretically unlimited, since the composite FB instance implements the interface ***IFBDiagramElement***.

A relation – ***ExtendInput*** is defined to allow for the connection between an input of a container CFB and an input of a constituent FB instance (Fig. 4.13

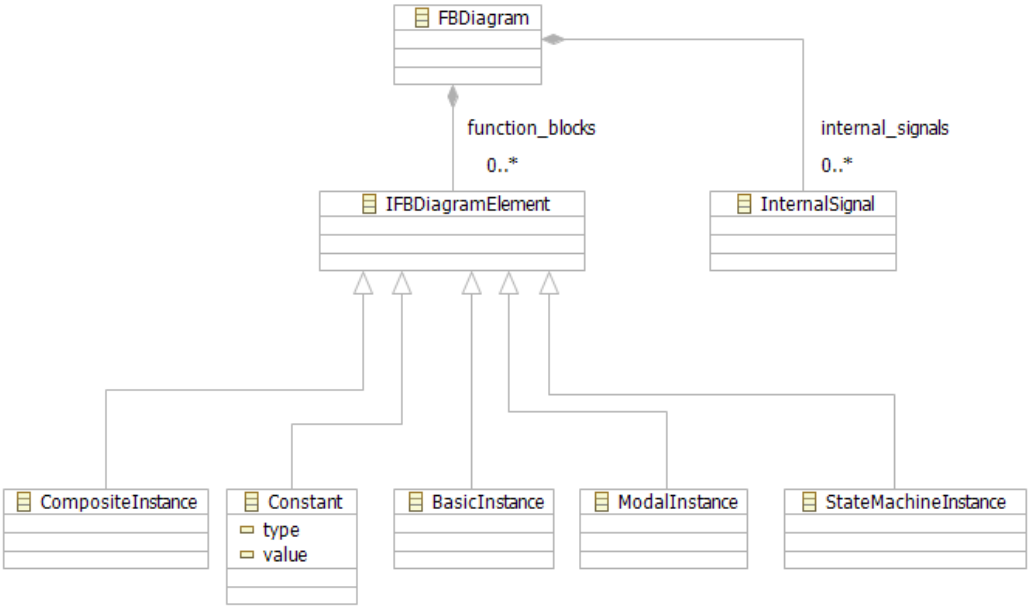


Figure 4.12: Meta-model of FB diagram

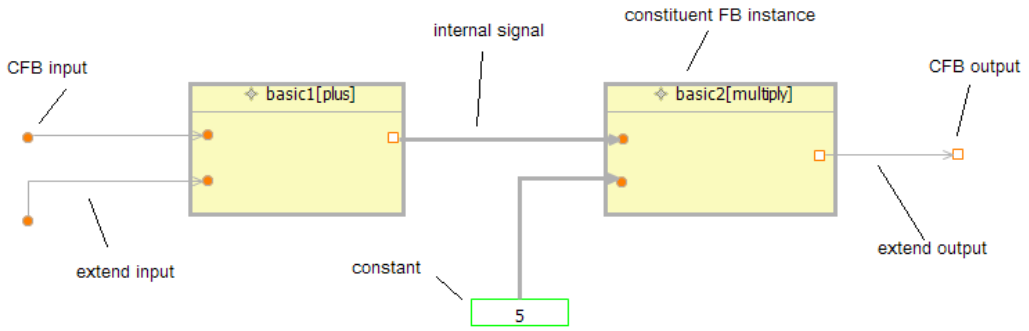


Figure 4.13: A FB diagram of a CFB

shows an example model), so that the data to the container CFB can be passed into its constituent FB input. The constituent FB instance should be an immediate child of its parent – the container CFB. Analogously, ***ExtendOutput*** is defined to connect two outputs – between the container CFB and its immediate constituent FB output (see Fig. 4.14).

The purpose of the two extra relations is actually to expose the inputs or outputs of constituent FB instances. A CFB input can be connected to another input via ***ExtendInput***, which means data on the two inputs are identical. However,

constraints must be added to avoid wrong connections, e.g. two inputs of the container CFB being connected by the ***ExtendInput***, or two inputs of constituent FB instances being connected by the ***ExtendInput***. In other words, an ***ExtendInput*** can be only used to connect an input of a constituent FB instance to an input of its immediate container CFB. Similar constraints should be applied to the ***ExtendOutput***, too.

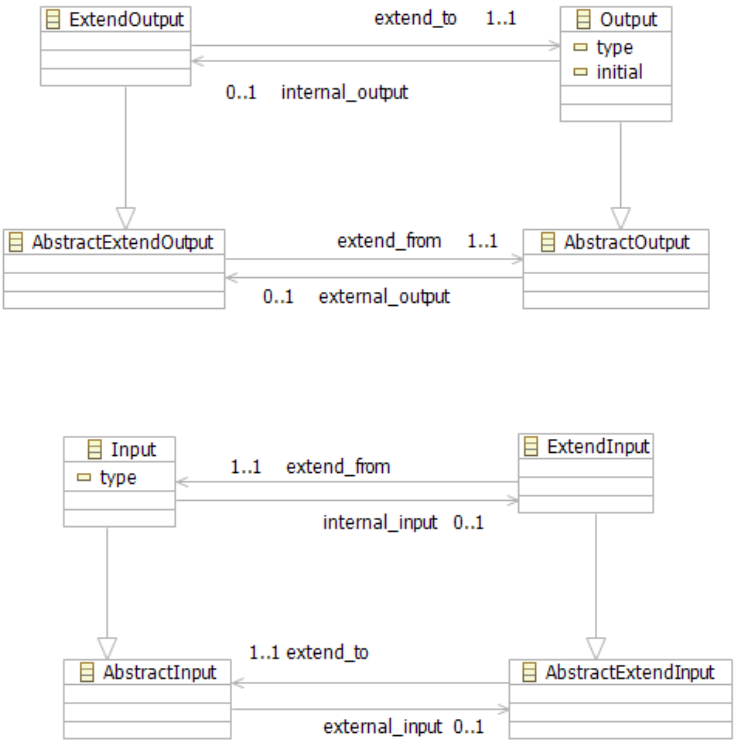


Figure 4.14: Meta-model of ***ExtendInput*** and ***ExtendOutput***

The CFB meta-model with multiplexity specifies only partially the construction rules for a composite kind (i.e. how many connections can be made on an output), while other, more strict syntactical rules with respect to interconnections with child FB instances are specified through implicit constraints. These additional constraints are summarized below:

- When playing the role of input of a container (parent) CFB, an input can only be connected to one input of its immediate constituent (child) FB instances via ***ExtendInput***.
- When playing the role of input of a constituent (child) FB instance, an

input can be connected either to one output of another sibling constituent FB instance via ***InternalSignal***, or to one input of its immediate container (parent) CFB via ***ExtendInput***.

- When playing the role of output of a container (parent) CFB, an output can only be connected to one output of an immediate constituent (child) FB instance via ***ExtendOutput***.
- The *type* attribute of connected input and output by ***InternalSignal***, or input and input by ***ExtendInput***, or output and output by ***ExtendOutput*** must be the same.
- A FB diagram must be acyclic.
- A CFB must have at least one function – the CFB kind driver.

CFB executes the function block diagram by means of a standard routine – the so called *CFB kind driver*. As a built-in function of CFB, it invokes encapsulated FB instances according to a static execution schedule, i.e. a linear sequence of its constituent FB instances. In such an execution schedule, each constituent FB instance should execute exactly once during a single activation of the CFB. Because the CFB is actually a directed acyclic diagram, the liner sequence can be derived from the flow of signals in the corresponding FB diagram, using a topological sort algorithm [75].

4.3.2.4 State machine function blocks

COMDES separates the treatment of reactive behaviour (control flow) and transformational behaviour (data flow). This is accomplished via two types of component – state machine function blocks (SMFBs) and modal function block, whereby the state machine function block (master) is used to indicate the current state to one or more modal function blocks (slaves) that perform the required signal transformations within the corresponding states/modes of operation (see Fig. 4.15).

Furthermore, an advantage of this separation-of-concerns model is that the SMFB and its MBF can be placed in different actors deployed in distributed network nodes, thus enabling online reconfiguration of system actions in the sense that the MFB nodes can be dynamically replaced by another set of modal FB nodes, when the master SMFB node is in an appropriate state.

A state machine FB consists of a number of binary inputs, an event-driven state machine model, and exactly two outputs: ***state*** and ***state_ updated***(see Fig. 4.16 for an example). When a SMFB is executed, the two outputs are determined

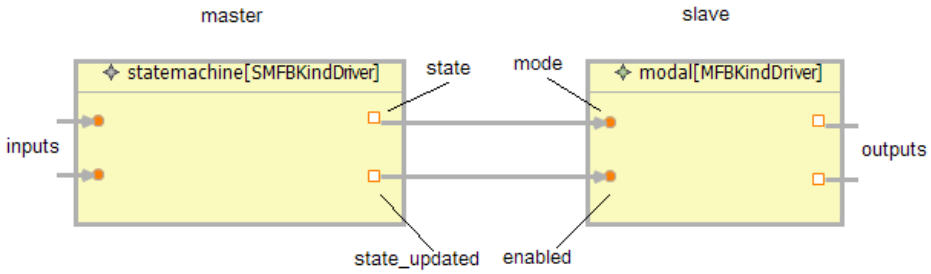


Figure 4.15: Coupling SMFB and MFB

according to the input signals. The ***state*** output represents the currently active state of the FB, and ***state_ updated*** output will be set *true* if a state transition has happened, otherwise it is *false*. The two output signals from a SMFB will be used by the corresponding modal FBs to execute control actions associated with specific states (see Fig. 4.15).

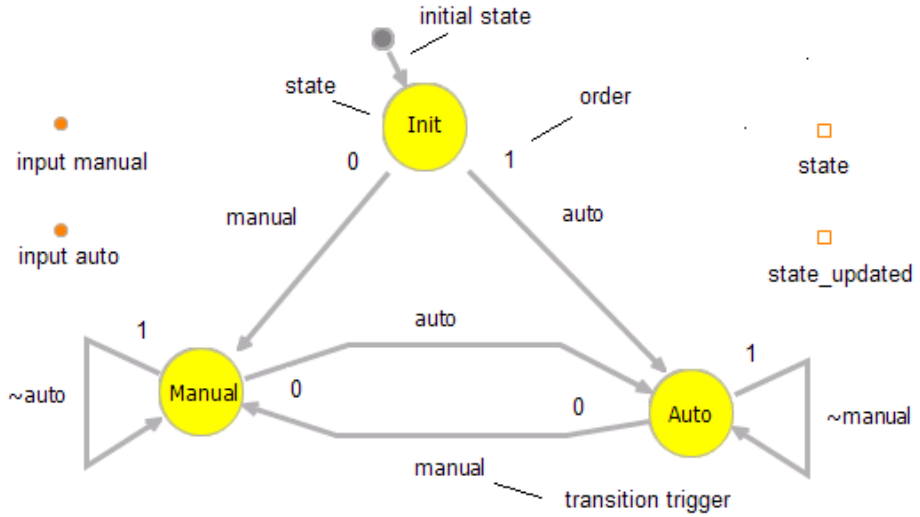


Figure 4.16: A state machine model

The meta-model of SMFB is depicted in Fig. 4.17. The ***AbstractStateMachineKind*** extends the ***AbstractFunctionBlockKind*** class to inherit aggregation relations to the ***Input/Output*** classes, the ***Function*** class, and the ***Parameter*** class, similar to how other kinds of FB are constructed. However, there is a constraint regarding outputs that must be complied with:

- There are exactly two outputs associated with a specific state machine FB:

state and ***state_ updated***, and the *type* of the ***state_ updated*** must be *Boolean*.

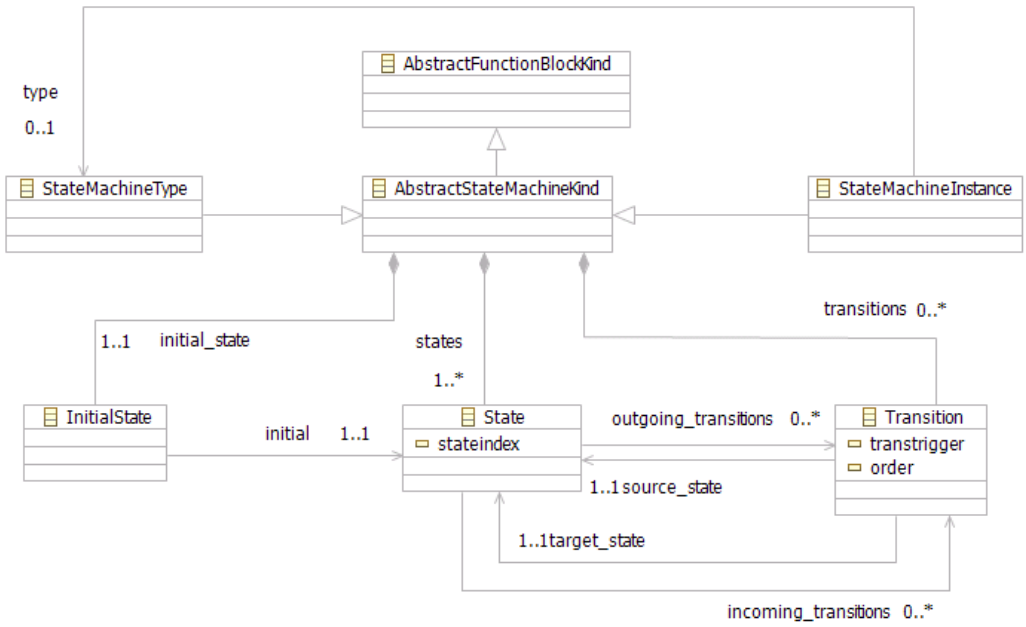


Figure 4.17: Meta-model of state machine FB

A SMFB contains an initial state, a number of states labelled by a *stateindex*, and transitions labelled by transition trigger (*transtrigger*) and *order*. A transition trigger string should be an expression composed of a set of Boolean values, Boolean variables and operators. The expression results in a Boolean value, i.e. *true* or *false*. All Boolean variables should be provided by inputs or parameters of the SMFB, hence these should be of Boolean type as well. Furthermore, a mechanism is required to test the transition trigger expression against the proper format consisting of associated Boolean variables, Boolean values and operators, etc. Counting occurrences (i.e. brackets) should be considered accordingly. The above requirements can be specified as constraints:

- All Boolean variables in each transition trigger expression of an SMFB must be declared in inputs or parameters of the SMFB.
- The format of the expression string must be valid.

When multiple outgoing transitions from a source state are enabled to fire at the same time, transition order is used to deterministically fire the most important

state transition and determine the current state. Therefore, the *order* on each outgoing transition of one source state should be different. Hence, the following constraint:

- The *order* of each outgoing transition of one source state must be unique to the source state.

Each state has a unique state index that will appear at one of SMFB outputs – the ***state*** output, when the state is activated. From outside of the SMFB, it should be possible to identify which state is the current active state, and that is why a constraint has to be applied to the state machine FB in order to guarantee that all states are distinguished by the *stateindex*.

- The *stateindex* number of each state must be unique to a state machine

A *State Transition Table* can be derived from a state machine model and subsequently be interpreted by a standard routine – a *SMFB kind driver*, activated by the corresponding host actor. When activated, the driver processes the table containing the successor states of the state visited in the previous execution, in order to determine the current state. If a state transition has taken place, the ***state*** and ***state_ updated*** outputs are updated accordingly. Hence:

- A SMFB must have at least one function – the SMFB kind driver.

4.3.2.5 Modal function blocks

A modal function block has a number of operational modes, where each mode executes a specific action implemented with a number of constituent function blocks or constants, e.g. discrete control, continuous control, etc. Constituent function blocks are instances of function block types implementing the corresponding signal transformation functions.

An input ***mode*** is used to select a particular group of function block to be executed in a given mode. This input is usually generated by an SMFB. A mode is enabled for execution by the corresponding value (*true*) of the ***enabled*** input, i.e. the control action should only be performed when a state transition occurs, since the SMFB–MFB pair operates as a clocked event-driven state machine (see Chapter 3 – Section 3.2.2).

The above features are reflected in the meta-model of the MFB kind, which is shown in Fig. 4.18. It comprises a set of operation modes, and each ***Mode*** class is a subclass of the ***FBDiagram*** class. The Composite Pattern used when defining CFB is applied to the ***Mode*** once again. As a result, hierarchy in each

mode is possible. Analogous to the meta-model of CFB, a MFB also contains ***ExtendInput*** and ***ExtendOutput*** in order to expose the inputs and outputs of encapsulated FBs. In addition to that, a MFB should contain two mandatory inputs: ***mode*** and ***enabled***. The ***mode*** input is used to select executing modes, and the Boolean ***enabled*** input determines whether the execution of a particular mode is allowed or not. Hence, the following constraint:

- Each MFB should have at least two inputs: ***mode*** and ***enabled***, and the *type* of the ***enabled*** input must be *Boolean*.

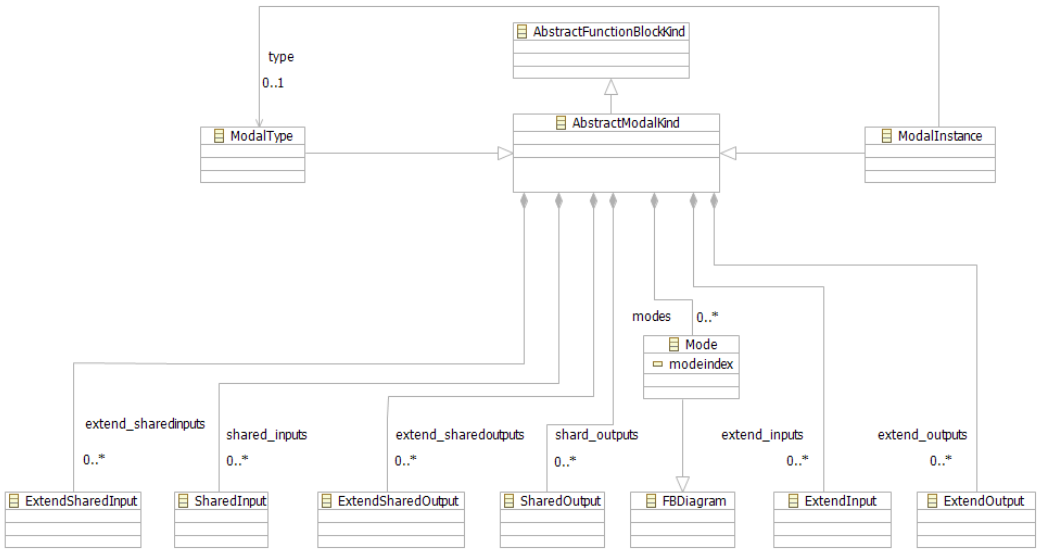


Figure 4.18: Meta-model of modal FB

Since FB instances residing in each mode are executed alternately, sometimes FB instances executed in different modes need to provide data to one and the same output of their container MFB. For example, in Fig. 4.19, three function blocks in three different modes are connected to one output of their container MFB in order to provide data through the same output named *outsignal*

Therefore, ***SharedOutput*** and ***ExtendSharedOutput*** are defined to allow for such kind of connections. A ***SharedOutput*** extends the ***AbstractOutput***, thus it can be connected to an output of its container MFB through an ***Extend-Output*** relation (see Fig. 4.20). The relation ***ExtendSharedOutput*** is used to connect an output of a constituent FB instance to a ***SharedOutput***. A ***Shared-Output*** allows for multiple ***ExtendSharedOutput*** connections. At the end of each execution of MFB, a ***SharedOutput*** will be updated to the value of the FB

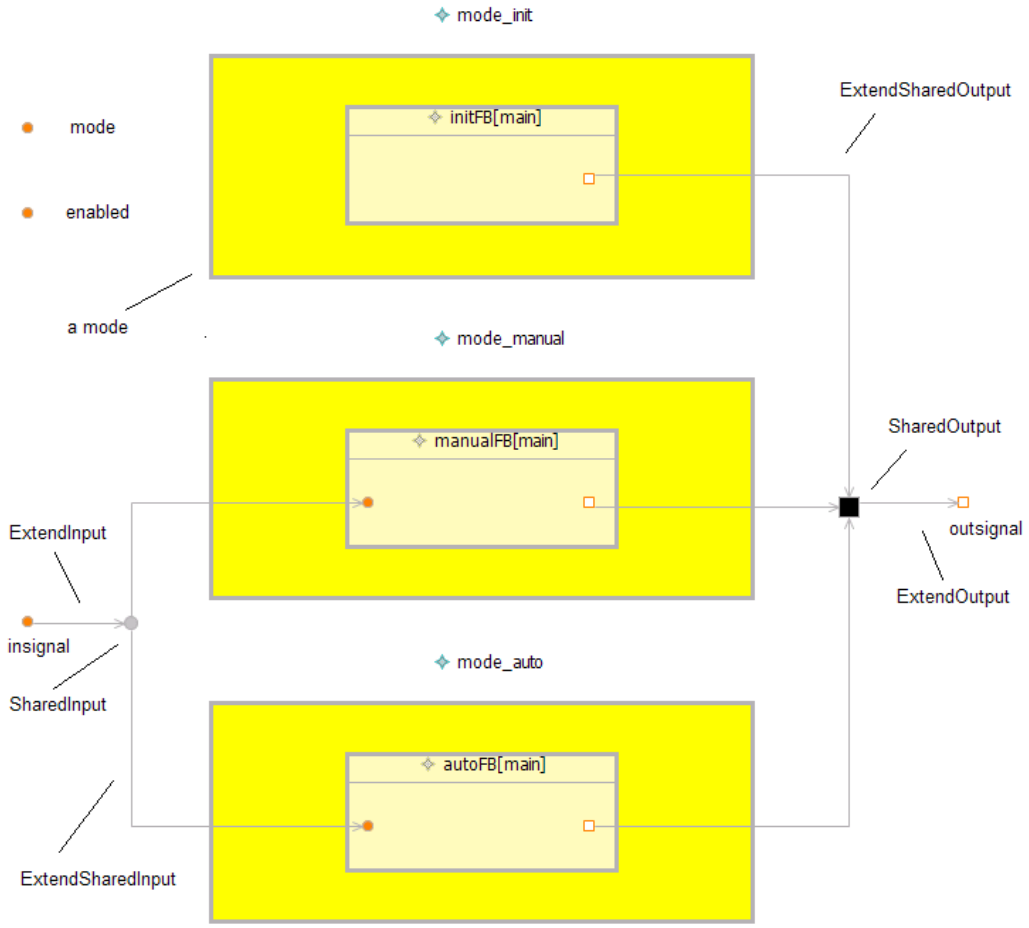


Figure 4.19: A modal FB model

output that is connected to the ***SharedOutput*** via ***ExtendSharedOutput*** and whose FB instance has been executed in the current execution. The value is in turn propagated to a container MFB's output via an ***ExtendOutput*** connection.

SharedInput and ***ExtendSharedInput*** are defined for a similar purpose, so that inputs of FB instances from different modes are able to obtain data from one container MFB's input, e.g. input *insignal* in Fig. 4.19.

Constraints of connections for ***ExtendInput*** and ***ExtendOutput*** defined for CFB still hold for MFB, except that an input can have one more connection related to the ***ExtendSharedInput***. The following list presents constraints concerning ***SharedOutput***, ***ExtendSharedOutput***, ***SharedInput*** and ***ExtendSharedInput***:

- When playing the role of input of a container (parent) MFB, an input cannot

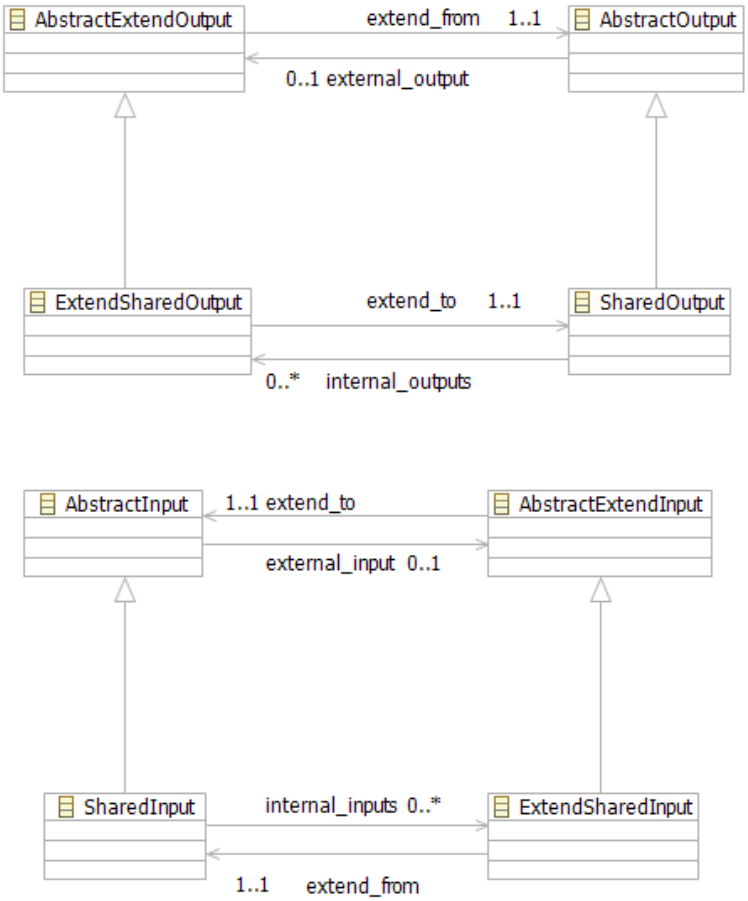


Figure 4.20: Meta-model of ***SharedInput*** and ***SharedOutput***

be connected to a ***SharedInput*** by ***ExtendSharedInput***.

- When playing the role of input of a constituent (child) FB instance, an input must be connected to one of follows: to one output of another sibling constituent FB instance by ***InternalSignal***; or to one input of its immediate container (parent) MFB by ***ExtendInput***; or to one ***SharedInput*** of the same container MFB by ***ExtendSharedInput***.

- When playing the role of output of a container (parent) MFB, an output cannot be connected to a ***SharedOutput*** by ***ExtendSharedOutput***.

- Every two outputs connected to a ***SharedOutput*** of a container MFB through ***ExtendSharedOutput*** must belong to immediate constituent (child) FB instances that are in different modes of the MFB.

MFB utilizes a standard routine – *MFB kind driver* to execute each mode. Similar to CFB, the driver invokes all FB instances of a selected mode according to a linear sequence that is derived from the interconnections of these FB instances.

- A MFB must have at least one function – the MFB kind driver.

4.3.2.6 Drivers

An actor interacts with the outside physical world or other actors through drivers. In fact, they communicate with each other or with the external world only when input or output drivers are executed. Drivers are executed atomically, i.e. they are not interrupted or preempted by other tasks.

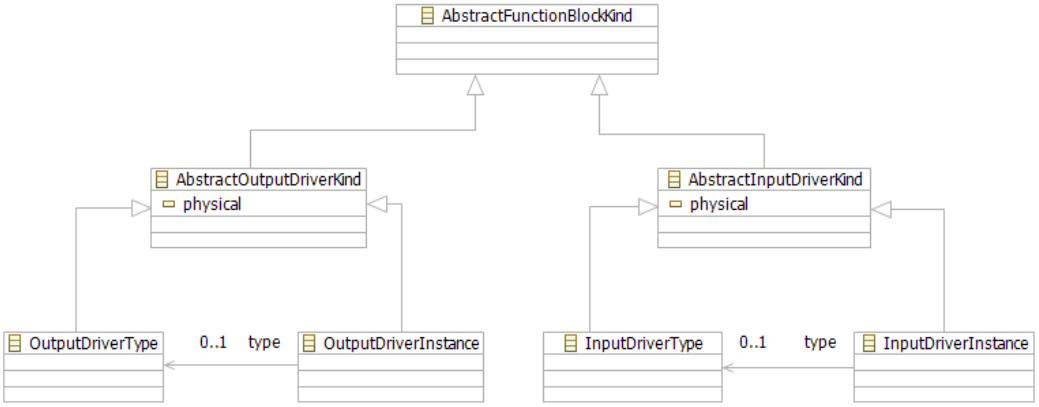


Figure 4.21: Meta-model of drivers

Drivers can be modelled as a special kind of function block, thus in the meta-model (Fig. 4.21) input driver and output driver are subclasses of the ***AbstractFunctionBlockKind***. Drivers are classified as *communication signal drivers* and *physical drivers*. Communication signal drivers are responsible for broadcasting and receiving signal messages within local as well as remote interactions. This is done in a transparent fashion using kernel communication primitives (in local communication), which may invoke the services of a network communication protocol (in remote communication). Physical drivers are used to sense and actuate signals from/to physical peripherals. In the meta-model, a Boolean attribute called *physical* is employed to distinguish the two categories.

I/O drivers play the role of actor interfaces supporting interaction between the outside environment and the internal context of the actor. A driver can be considered as a wrapper, which uses a function to decompose an incoming signal

into local variables that are accessible via driver outputs – with input driver, and conversely uses local variables provided by inputs to compose an outgoing global signal – with output driver. Therefore, in the meta-model, an output driver has only inputs while an input driver has only outputs, which is enforced by the following constraints:

- An input driver does not have inputs.
- An output driver does not have outputs.

The functions of drivers are usually related to platform peripherals or run-time environments, e.g. external input interrupt, ADC (analog-digital converter), PWM (pulse-width modulator), CAN (Controller Area Network) controller, or communication services provides by a real-time operating system.

4.3.3 System composition

4.3.3.1 Actor

From the composition point of view, an actor consists of a signal-processing block, which is mapped onto a non-blocking (basic) task provided by a run-time environment, as well as an input latch and an output latch containing input and output drivers respectively, that are used to exchange signals with other actors and the outside world. Actors are configured from FBs, whereby drivers as special-purpose FBs are placed in the latches, whereas other kinds of FB are instantiated and interconnected in the signal processing block, so as to compose a FB diagram accomplishing the specified functionality. An actor model is illustrated graphically in Fig. 4.22, and the meta-model of actor is depicted in Fig. 4.23.

From the timing point of view, actors are running in a run-time environment supporting a model of computation called Distributed Timed Multitasking, featuring split-phase execution of distributed embedded actors. In such an environment, drivers are executed separately from the signal processing task. Input drivers are executed when the task is triggered, and output drivers are executed when the task deadline arrives or when the task comes to an end (if it has no deadline), resulting in the effective elimination of task execution jitter.

A number of actors could be allocated onto one network node, which raises the issue of concurrent execution of actors. Therefore timing properties must be explicitly specified in the meta-model.

There are three possible ways to trigger an actor: *periodic timing event*, *sporadic external event* and *signal arrival event*. This is specified by the attribute

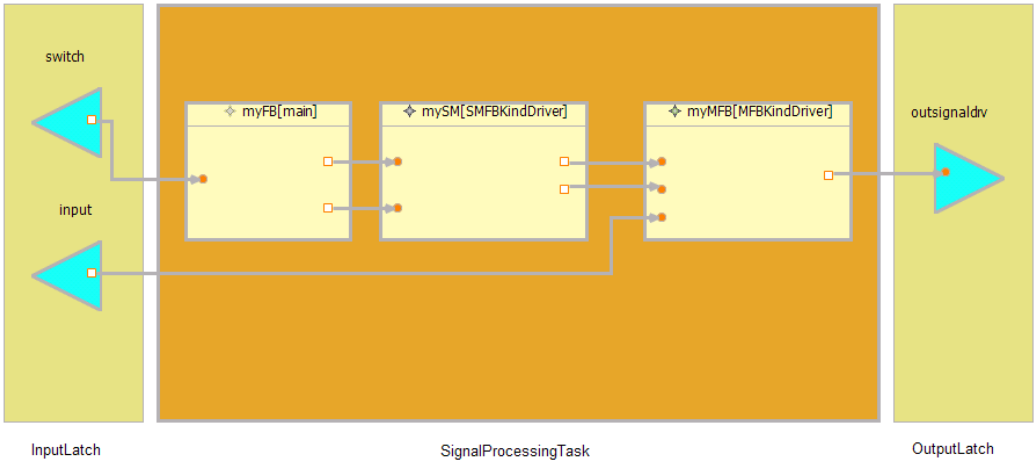


Figure 4.22: An actor model

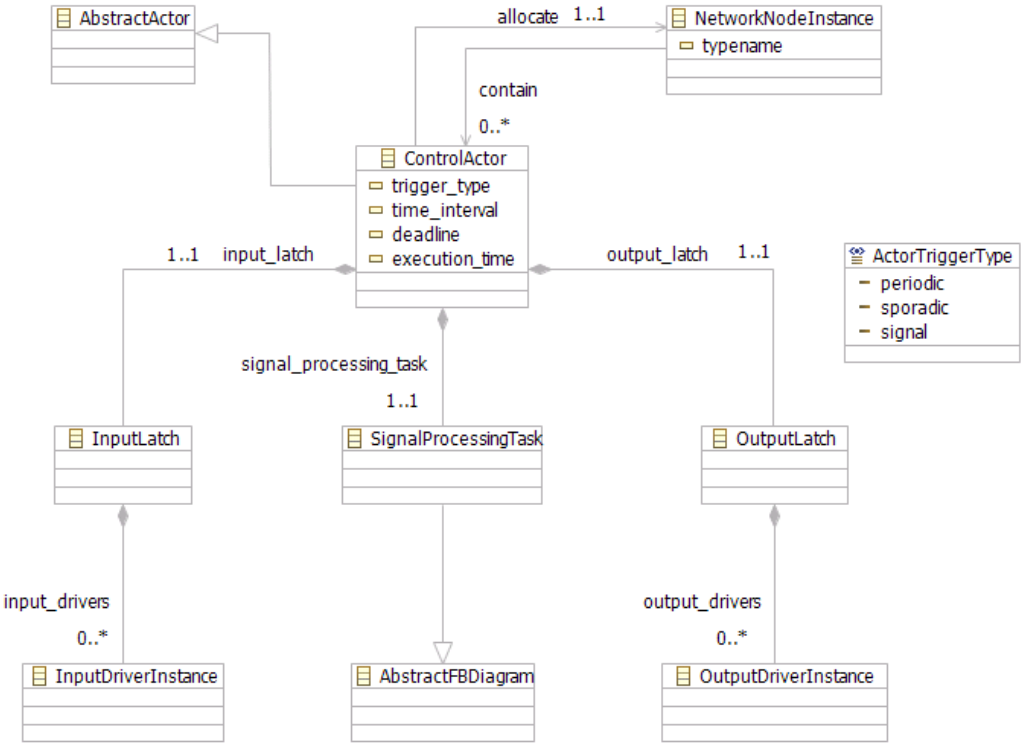


Figure 4.23: Meta-model of actor

trigger_type of the ***ControlActor*** class in the meta-model. An actor can have at most one trigger.

The attribute *time_ interval* is a constant integer strictly greater than zero. If an actor is triggered by a periodic event, the *time_ interval* attribute indicates the activation interval of a periodic actor. If an actor is triggered by a sporadic event, the *time_ interval* means the minimal interval of the occurrence of the sporadic event. In this case, its value is used for the purpose of analysis because an actor should finish its execution before the next possible sporadic event comes. The value of the *time_ interval* can be ignored if an actor is triggered by a signal arrival event.

The attribute *deadline* is a constant integer strictly greater than or equal to zero, designating the deadline of a given actor. When the deadline of an actor is explicitly specified and greater than zero, then the actor will generate its computation outputs at the specified deadline instant, otherwise the actor outputs its computation results immediately after finishing computation. For each periodic or sporadically triggered actor, the *deadline* value should be strictly no greater than the corresponding *time_ interval* value.

The attribute *execution_ time* is a constant integer strictly greater than zero designating the required computation time of a given actor, and typically referring to the worst-case execution time. If the deadline of an actor is specified (i.e. $deadline > 0$), then the *execution_ time* value of the actor must not exceed its deadline value. If the deadline is not specified but the actor is periodically activated, then the *execution_ time* value must not be greater than the specified *time_ interval* value. This attribute is for the purpose of analysis as well.

4.3.3.2 System

A particular COMDES system is a composition of actors, network nodes onto which actors are allocated, and signals through which actors interact with each other via drivers. The contained actor class in the meta-model (see Fig. 4.24) is abstract because a system could be composed of actors modelling controllers or modelling controlled environments. The definition of environment actor can be similar to the definition of the control actor. Fig. 4.25 shows an example system model in graphical notations.

A system is eventually built out of function block instances and network node instances whose types are stored in a repository. When configuring a COMDES system from these predefined types, it is necessary to access the repository to obtain information about these types. For example: function blocks are stored as prefabricated objects, and the location of the objects in the repository needs to be known, so that these objects can be linked to instances that are generated from a system model. The access to repositories is modelled via the relation *repositoryroot*

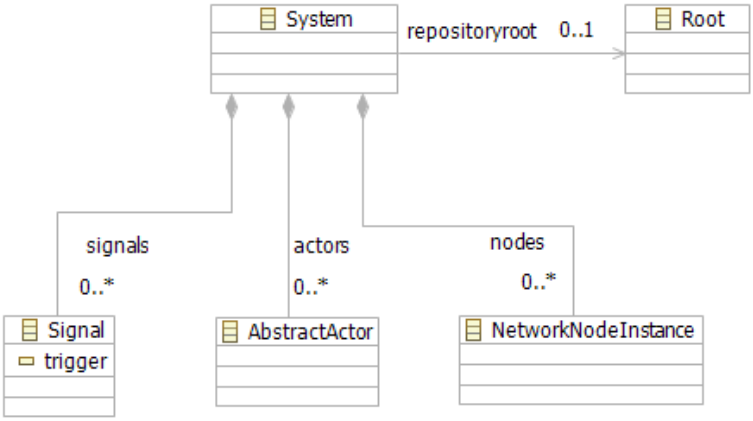


Figure 4.24: Meta-model of system

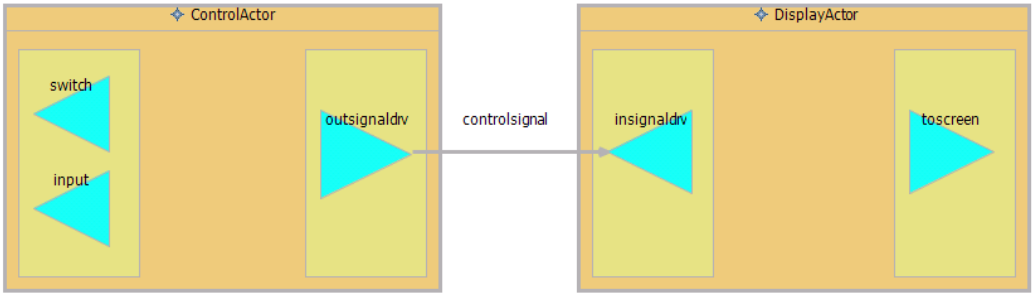


Figure 4.25: A system model

to the ***Root*** class.

Signal (see Fig. 4.26) consists of one or more variables and each variable contained in a signal should have a data type (the ***COMDESType*** as shown in Fig. 4.8), as well as a name specified by the attribute *variable*. A signal connects from output driver instance to input driver instance of communicating source and destination actors. A Boolean flag *trigger* is used to designate whether or not the signal triggers a receiver actor.

4.3.4 Platforms

A network node model is a platform model on which COMDES actors can execute. Each network node has a hardware aspect (i.e. type of microcontroller) and a software aspect (i.e. tool-chain used to build application). A COMDES system may employ a number of identical hardware platforms to host different actors.

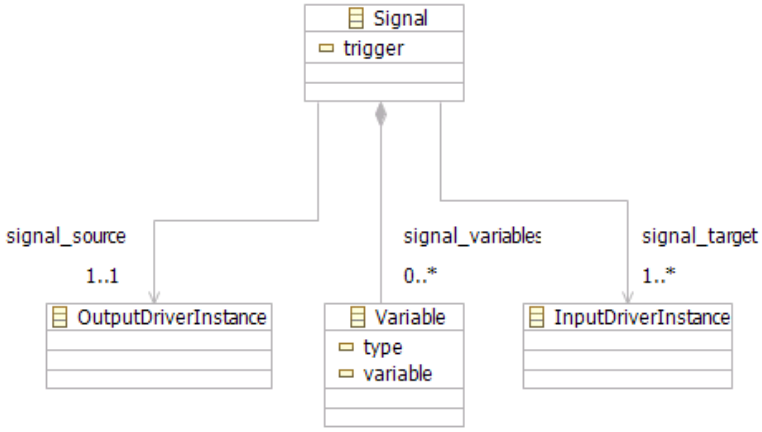


Figure 4.26: Meta-model of signal

Therefore, the *Kind-Type-Instance* pattern is used once again in the meta-model to allow for reuse of network nodes.

An assumption is that that each network node has only one microcontroller, that is specified by the attribute *cpu*, as shown in Fig. 4.27. The attribute *required_ toolchain* specifies what tools (i.e. compiler, linker) are needed to build a COMDES executable from the generated source code and prebuilt FBs for the given node. Values of the two attributes could be referred to by another attribute – *build_ script*.

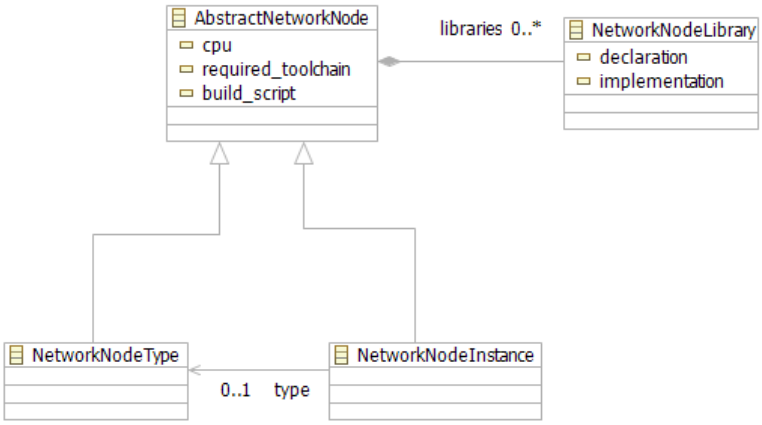


Figure 4.27: Meta-model of network node

For the purpose of automating the build process, the attribute *build_ script* specifies how to build the COMDES executable with the required tool-chain, and

what resources are required. The script can be written in a language that is specific to a build tool that COMDES employs (i.e. GNU make). In the script, the information about *cpu* or *reqiured_ toolchain* would be needed.

Moreover, in order to find out all resources for the build, this script needs to have access to the artefacts generated from a COMDES system model, as well as prebuilt components from the FB repository. However, what source files to generate is dependent on the COMDES implementation and code generators, which are unknown before a generation step. Thus, it is very hard to specify such information during the modelling stage.

A simple solution is to define a set of external variables representing such information. The variables are used by the *build_ script*. These variables can be assigned in another script, which is generated during a transformation step from a COMDES system model. Then the *build_ script* only needs to include or import the generated script and use the predefined variables (Chapter 6 – Section 6.3).

Another straightforward solution without using any extra definitions of variables would require a two-step generation. In the first step, a build script can be specified with information from a COMDES model, because nothing is known yet about generated files. The script is then used to generate a COMDES code generator's template (or input). In the second step, the code generator reads the COMDES model and generates a complete script based on the template. However, this approach requires a complicated code generator supporting dynamic template reading during generation, because in the second step, the code generation templates are not fixed.

The meta-model of the network node can be refined with more classes and associations to model platforms including hardware aspect, software aspect or even timing aspect in more detail. For example, each node could have a number of devices and each device needs a device driver. When a COMDES actor driver accesses a device, it is necessary to check if the device and its driver exist on the allocated network node. If it does not match, meaning that the network node does not satisfy the actor's requirement, other types of network node should be considered.

Delivering general hardware abstractions is a difficult task caused by the large variety of architectures used in embedded system design. Furthermore, since COMDES is intended to provide a solution for embedded software rather than hardware, such detail is not provided in the meta-model. However, tasks like checking a device driver against a COMDES driver can still be fulfilled if the task is left to compilers. In the meta-model, a network node can contain a number of libraries. Each library needs to be specified with a *declaration* and an *implementa-*

tion, which describes what function calls (i.e. device drivers) the node can provide. Remember that the ***Function*** class in the generic FB pattern (Fig. 4.7) contains the attribute *includes*, specifying a list of function calls required in the code of a function. So, when a component developer creating a driver (or other FBs) and writing implementation code for it, he (or she) has the knowledge of what device drivers or function calls he (or she) is invoking from within the code and has to list these function calls in the *includes* attribute. Next, the developer will compile the source code of the component for a specific platform, which provides the required function calls. If the required function calls do not match the provided ones, the component will not be compiled, and as a result, a prebuilt component for the platform will not be available for reuse. Consequently, the component model as well as its objects should not be seen in the repository.

4.3.5 Repository

A COMDES repository root comprises three distinct repositories: an application repository that collects all COMDES applications, function block repository that stores all reusable function block type models, interfaces and executables, and a platform repository that contains models related to hardware and software (i.e. definitions, function libraries, etc.).

The meta-model of the repository is quite straightforward (Fig. 4.28), and the underlying idea is that a repository structure can be generated out of a repository model. The structure may be implemented by a file system or a database system. Consequently, the generated structure just needs to be filled with contents, i.e. interface files, model files, executable files, etc. For example, the application repository contains a number of projects that are modelled respectively by the ***ApplicationRepository*** and the ***COMDESProject*** class. Each project model can be generated as a folder in a file system that physically holds a number of application-related files, e.g. models, source code, executables, configuration scripts, etc.

A FB repository is classified according to particular categories, each of them containing a number of folders modelled by the class ***FBTypeFolder***. Each folder represents one type of function block and thus should contain all related information; for example, the component type model as well as the interface used to access the component must be provided in its type folder. According to the COMDES notion of component, function blocks are prebuilt components specific to various platforms, which is modelled by the class ***Executable*** and the relation *platform* to the ***NetworkNodeType*** class.

The platform repository stores models of network node types as well as hardware-

specific definitions that COMDES needs, i.e. definitions of COMDES data types *BTYE*, *UBYTE*, etc. These definitions are saved in one or more files that are specified by the *includes* attribute the ***HardwareFolder*** class. The information can be later retrieved when a platform is selected for a configuration.

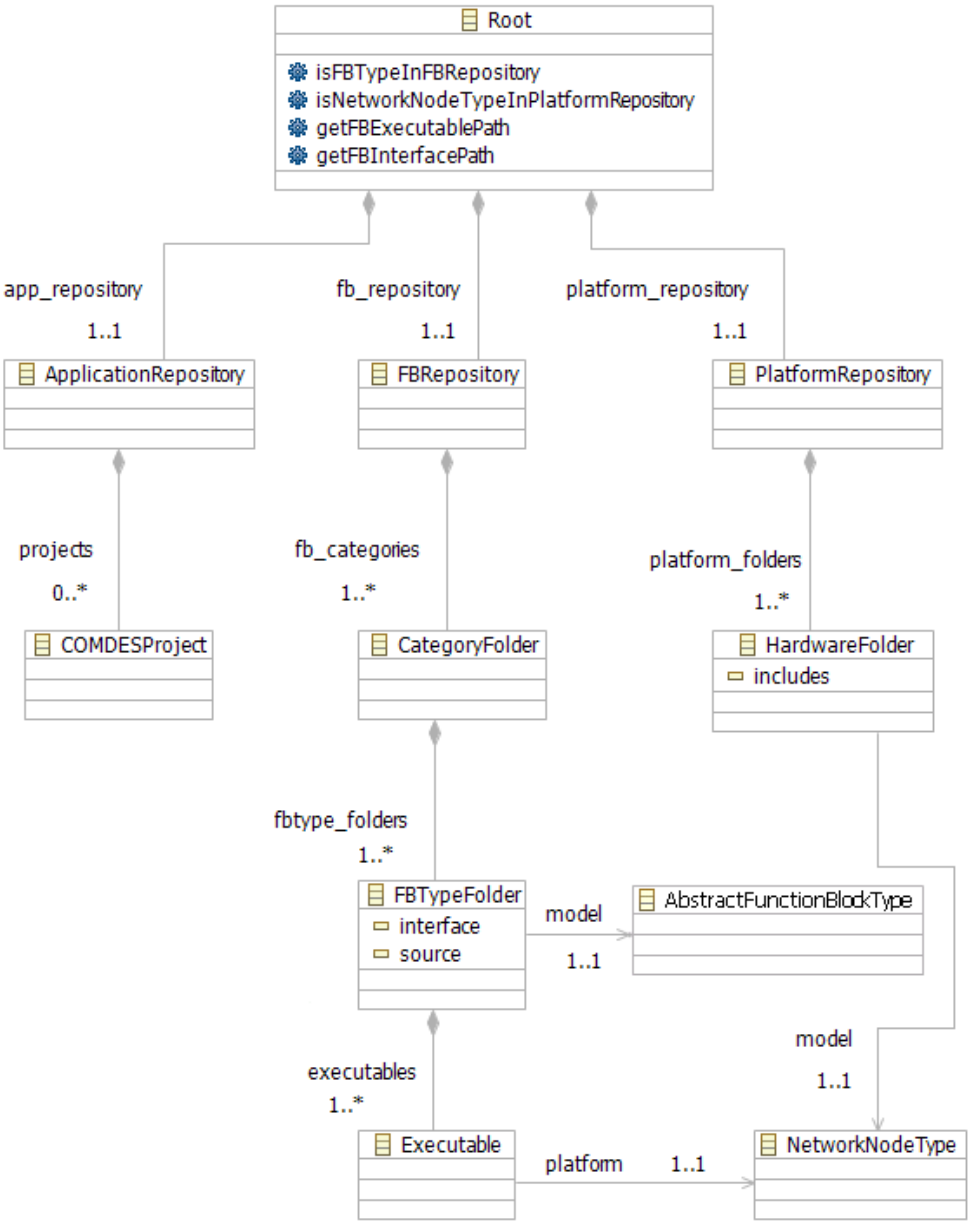


Figure 4.28: Meta-model of repository

Fig. 4.29 demonstrates a repository model containing an application reposi-

tory, a FB repository as well as a platform repository. There are two kinds of platform in the platform repository: ***PC*** and ***SAM7_EX256***. The FB repository has three categories: ***Math***, ***InputDriver*** and ***OutputDriver***. In the figure, a FB type folder from the ***InputDriver*** category is selected, and the corresponding information of the folder (i.e. interface, FB type model reference, etc.) is displayed in the table below. This input driver has only one executable based on the platform ***SAM7_EX256***.

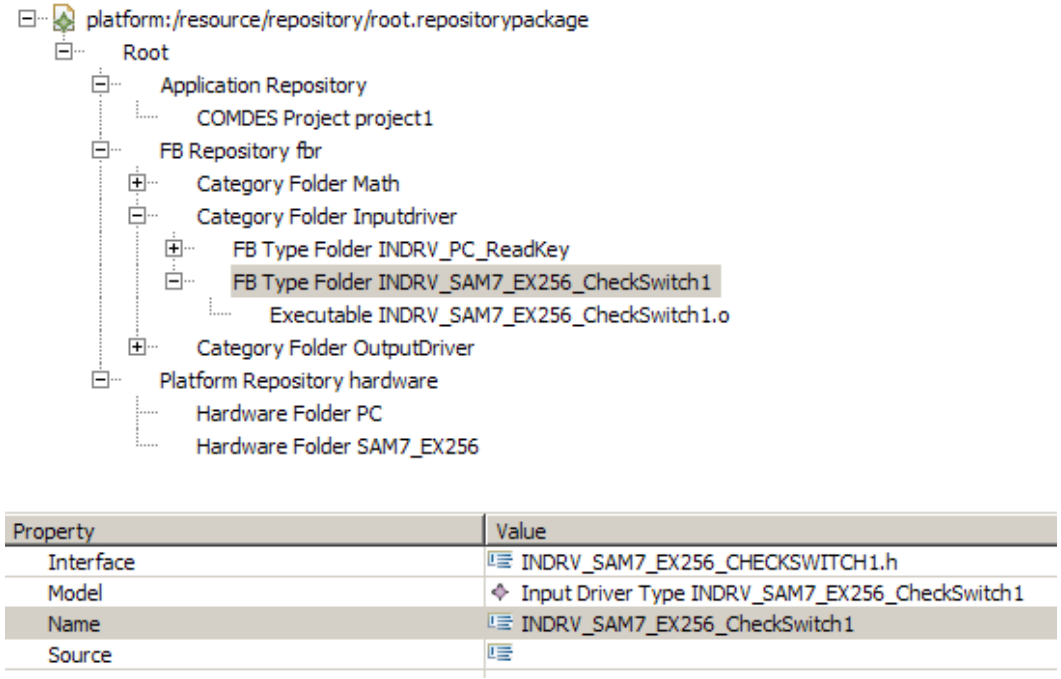


Figure 4.29: A repository model

In addition to the repository structure defined in the meta-model, methods to access the repository must be available, in order to allow for automatic configuration. For instance, if the repository is implemented as a file system, it is necessary to know where a FB executable of a given platform is located, so that it can be linked to other compiled FB instances in an assembly. Thus, a number of operations of the ***Root*** class must be implemented. The list given below contains the declarations of four methods that are utilized by the configurator tool (see Chapter 7) to generate the configuration script. There could be more methods provided in order to facilitate the process of repository search and information retrieval.

- `boolean isFBTypeInFBRepository(String fbtypeid)`

- `boolean isNetworkNodeTypeInPlatformRepository(String nodetypeid)`
- `URI getFBExecutablePath(String fbtypeid,String nodetypeid)`
- `URI getFBInterfacePath(String fbtypeid)`

4.4 Summary

This chapter has presented the modelling techniques that COMDES defines for building embedded control applications from reusable components, at a higher level of abstraction. Specifically, a meta-model specified by the Ecore meta-modelling language formally defines the syntax of the COMDES framework, which is in turn used to support the specification of concrete COMDES component types, as well as the system models constructed from predefined component types. As a foundation for developing real-time embedded control software using the COMDES approach, the meta-model contains sufficiently complete information, so that tools can be employed to automatically generate code from models that can be directly running on platforms. Based on the meta-model, system dynamics can be formally specified in terms of timing and functional behaviours, in collaboration with a function block repository. With concrete information about execution platforms, system implementations can be automatically synthesized via one or more transformation steps. Final system executables can be configured by dedicated compilers and linkers, and subsequently deployed onto target microprocessors.

The meta-model can be seen as an implementation and a detailed specification of the COMDES framework introduced in Chapter 3. However, this chapter focuses only on COMDES building blocks regarding DSL, function blocks, platform and repository. Another part of the COMDES framework – a run-time environment model that can be transformed from a domain model, specified by the COMDES meta-model, will be described in the next chapter.

Chapter 5

Platform-Specific Model: A Run-time Environment

The COMDES design models take both system structural and behavioural aspects into account. In terms of structure, a system is conceived as a network of actors that communicate with one another by transparently exchanging signals via encapsulated drivers. An actor can be hierarchically composed from different kinds of function block instances to realize the required system functional behaviour. For modelling system behaviour, a separation-of-concerns approach is systematically applied, such that a number of non-functional properties (i.e. real-time execution and response) are specified with respect to actors, while system functionality is modelled as specific function block diagrams contained within actor tasks. Separation of concerns allows for clear specification of system dynamics in different aspects. The COMDES framework employs a distributed timed multitasking model of computation, which can be implemented by a corresponding run-time environment that manages the preemptive execution of prioritized actors and atomic execution of signal drivers at predefined time instants. The HARTEXμ kernel implements such an environment, and thus can be adopted as a platform of COMDES-based applications [76][77][78][79][29][62][80].

The original version of the kernel, supporting only stand-alone operation, was developed by Gourinath Banda in his Master thesis: "Scalable Real-Time Kernel for Small Embedded Systems", under the guidance of Prof. Christo Angelov, University of Southern Denmark (SDU), 2003. The kernel implementation has been subsequently optimised and improved in many aspects, and it has been extended for distributed operation adopting the CAN communication module, which was originally developed by Jens Lorenzen in his Master thesis: "Communication Protocol for Distributed Embedded Systems", under the guidance of Prof. Christo Angelov, SDU, 2003. Furthermore, the functionality of the kernel has been ex-

tended by new subsystems supporting Timed Multitasking, and the Task Manager has been redesigned to support the execution of tasks and drivers configured from function blocks. This version was developed by Krzysztof Sierszecki in his Ph.D. thesis: "Component-Based Design of Software for Embedded Systems", supervised by Prof. Christo Angelov, SDU, 2007. The kernel has been validated in a number of stand-alone and distributed experiments, as well as in course projects and master projects executed at the Mads Clausen Institute, SDU. Furthermore, research in component-based design using the HARTEX architecture has been carried out by Jesper Berthing in his Ph.D. thesis "Component-Based Design of Safe Real-Time Kernels for Embedded Systems", supervised by Prof. Christo Angelov, SDU, 2008.

The latest version of HARTEXμ has been specifically developed to provide an operational environment for COMDES-II applications, and it can be characterized by the following features [62]:

- *Boolean vectors* used instead of linked-list queues: Bitwise processing of Boolean vectors has resulted in considerable reduction of kernel overhead and constant execution time of kernel functions, independent of the number of tasks involved.

- *Basic tasks* sharing one common stack: Tasks are implemented as basic (non-blocking tasks) that share a common stack, which contributes to smaller memory overhead and simpler implementation.

- *Integrated task and resource management*: An elegant protocol called the System Ceiling Protocol [1] is employed in order to achieve predictable behaviour over shared resources by eliminating undesirable effects such as deadlock, unbounded priority inversion, etc.

- *Integrated time and event management*: Timing interrupts and external events are treated in a uniform manner, using event counters and event control blocks. The event control block specifies an operation to be carried out on the occurrence of an event, e.g. release one or more tasks. Thus the tasks can be released when a specified time interval elapses or an event threshold is reached.

- *Boolean vector semaphores*: This is a new type of synchronization object, which can be used to instantaneously notify a number of tasks about event occurrence or message arrival.

[1] This protocol is also known as the Stack-Sharing Ceiling Priority Protocol.

- *Content-oriented message addressing*: With this technique a message is addressed by its name, freeing application developers from all the associated details of senders/receivers, message size, message source and destination etc., thus providing support for transparent signal-based communication.

- *Stand-alone and distributed operation*: Both modes of operation are supported making the kernel a versatile solution supporting a broad range of embedded applications.

- *Timed Multitasking*: This mechanism combines the advantages of static and dynamic scheduling and makes it possible to eliminate task and transaction execution jitter in a dynamic scheduling environment. It is supported by the Integrated Event Manager.

- *Static Time Manager*: This is a dedicated kernel component used to efficiently handle hard real-time periodic tasks executing under Timed Multitasking, in the context of concurrently executing time-triggered transactions.

- *Easy porting*: All architecture-related issues are located in one hardware-specific module, i.e. the Hardware Abstraction Layer, which makes porting easy.

In this chapter, a simplified representation of the HARTEXμ meta-model is firstly developed using the Ecore notations by avoiding details that are not relevant for COMDES-based applications. Secondly, a transformation from the COMDES model to the HARTEXμ model based on the corresponding meta-models is presented. Finally the transformation is illustrated with a trivial scenario.

5.1 Meta-model of a run-time environment

5.1.1 Application

The HARTEXμ kernel consists of various modules implementing specific subsystems: Task Manager; Task I/O Manager; Resource Manager; Synchronisation and Communication Bus; Integrated Event Manager; Timed Multitasking; and Static Time Manager. Each module provides certain calls (primitives) that can be invoked by other module and/or tasks. In addition, the kernel modules and kernel objects depend on the services of a Hardware Adaptation Layer.

The kernel manages several types of software entity: events, tasks and task inputs/outputs, and it also provides support for task interaction via messages.

The Event Manager processes events and generates execution requests for tasks and I/Os. This is achieved by means of the primitives `release(tasks)`, and `finish(tasks)`, which are invoked by the kernel Event Manager (or Static Time Manager) when processing task-release and deadline events, respectively. The task outputs and inputs are invoked from within the kernel Task Manager and executed non-preemptively, before task scheduling takes place, if I/O execution requests have been registered by the Event Manager. Upon Task Manager activation, the registered outputs are executed before inputs, in order to make sure that the precedence relation between sender and receiver tasks is maintained. Released tasks are executed in a preemptive priority-based environment, the highest-priority task first.

Due to the specific architecture of the kernel, a real-time application is constructed by just configuring application-specific kernel objects (i.e. tasks, messages, events, etc.) within the corresponding application-independent modules (i.e. Task Manager, Synchronisation and Communication Bus, Integrated Event Manager, etc.). Therefore, only the application-specific kernel objects need to be specified in an application model.

The HARTEXμ kernel has been designed for stand-alone and distributed real-time systems. In the meta-model (Fig. 5.1), each distributed real-time application encapsulates a number of nodes as well as messages used to communicate between nodes. Each node consists of a number of kernel objects, i.e. tasks, semaphores, resources, events, etc., which are set up during a configuration. Subsequent sections discuss relevant models of kernel objects (objects that are not related to this thesis and COMDES are omitted).

5.1.2 Tasks

Application software in real-time systems is decomposed into several discrete, significant and appropriate smaller jobs. These are implemented as subroutines and are referred to as tasks, whose execution is controlled by the kernel task manager. Synchronisation and communication between tasks are assisted by other kernel objects like semaphores, messages, etc. All the tasks have a fixed statically defined priority starting from *1*, the lowest priority in the system. The HARTEXμ kernel supports only basic tasks, which can never be blocked. Basic tasks may be temporarily or permanently disabled. A running basic task will switch to an inactive state upon exiting the system, and it could be released again if enabled.

Transitions to and from the active task state are effected by means of task management primitives. An incomplete list of the task management primitives is given below:

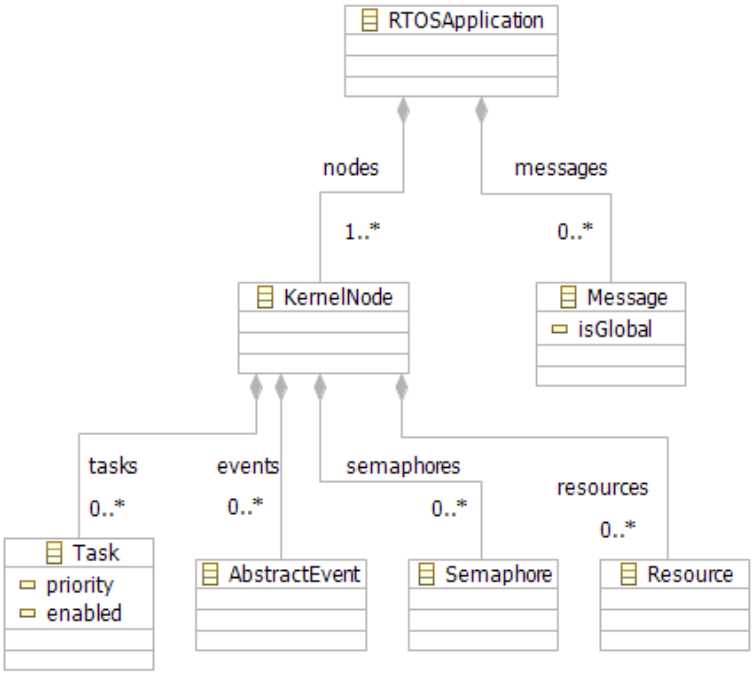


Figure 5.1: Meta-model of a HARTEXμ application

- `release(tasks)` – releases one or more enabled tasks and registers input drivers for execution, as specified by the tasks argument vector
- `finish(tasks)` – registers output drivers for execution of tasks that just completed (became inactive)
- `disable(tasks)` – disables one or more tasks, as specified by tasks
- `enable(tasks)` – enables one or more tasks, as specified by tasks

The four primitives are modelled using interfaces in the meta-model shown in Fig. 5.2, such as ***IReleaseTasks***, ***IFinishTasks***, ***IDisableTasks*** and ***IEnableTasks***. Any kernel object which implements any of the interfaces is able to operate on tasks accordingly.

In the meta-model (Fig. 5.3), the task itself is modelled using the ***Task*** class. A task that constitutes an application can have code in which certain primitives could be called to fulfil requirements of the application. Task input/output are invoked at the beginning and at the end of task execution respectively. More precisely, the kernel supports split-phase execution of tasks and I/O in accordance with the concept of timed multitasking, whereby inputs and outputs are executed

at particular time instants in separation from the task. Both of them are assumed to be short pieces of code executed atomically (non-preemptively). Moreover, they are executed according to a priority order, such that higher-priority tasks have higher-priority input/output.

5.1.3 Messages

The kernel provides means for tasks to communicate with one another, so as to achieve the needed system functionality, in stand-alone and/or distributed sys-

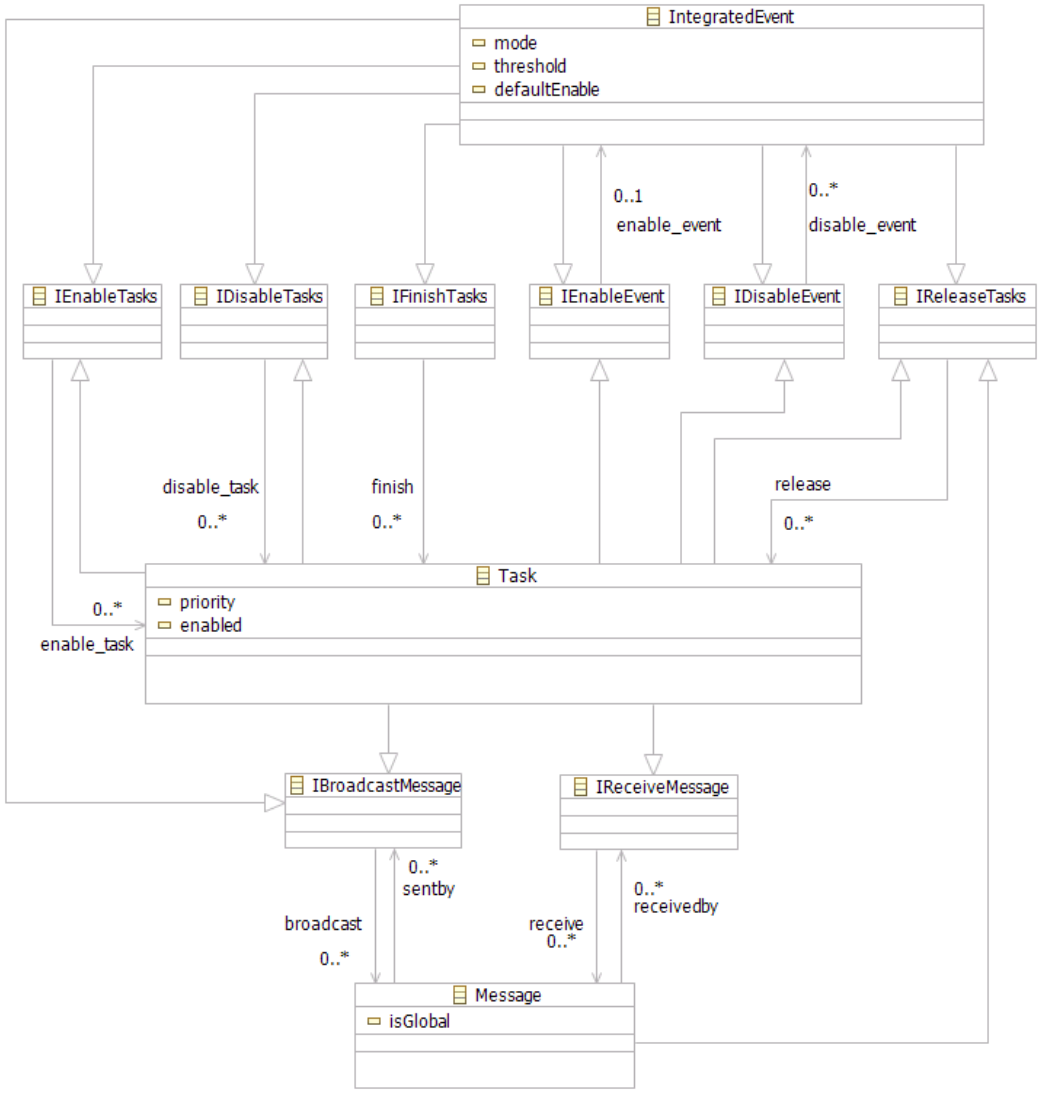


Figure 5.2: Meta-model of kernel primitives

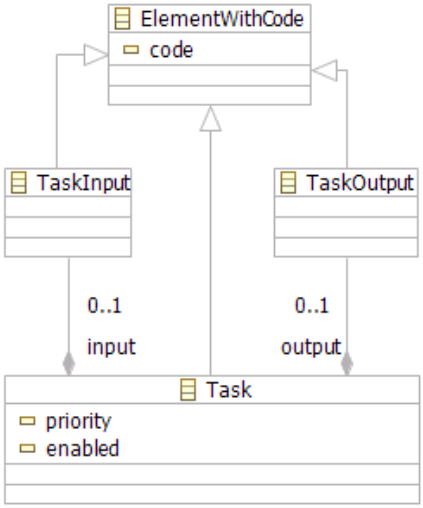


Figure 5.3: Meta-model of tasks

tems. Communication is of non-blocking type, i.e. state message communication with message-overwrite semantics. As shown in the meta-model (Fig. 5.4), a message is composed of one or more variables. The task sending a message is referred to as a *Sender* task, while the task receiving the message – as a *Receiver* task. Sending and receiving messages is effected by operations `broadcast(message)` and `receive(message)`, which are modelled as interfaces ***IBroadcastMessage*** and ***IReceiveMessage***, respectively. If the receiver task and the sender task are located in different nodes, the message is considered as a global message. In the case, the *isGlobal* attribute should have the Boolean value *true*.

Message arrival is notified by means of an event. So, the sender task on reaching

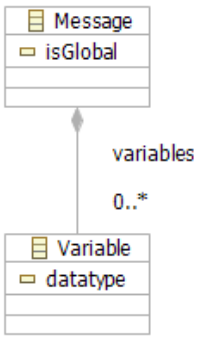


Figure 5.4: Meta-model of messages

its communication point, broadcasts the message; it notifies the receivers that the message is ready by signalling them (and eventually releasing all or some of them). The receiver tasks, on reaching their communication points, check whether the message is ready; if ready they just read the message to their local destination buffers and continue their execution sequence.

5.1.4 Events

The purpose of the real-time system is to recognize various timing, external and internal events, and to generate relevant reactions by executing the corresponding tasks in a timely and predictable manner. It is the duty of an Event Manager to identify the occurring events and accordingly, execute specific operations, e.g. `release (tasks)`, `finish(tasks)`, etc., when the event counters of one or more events expire.

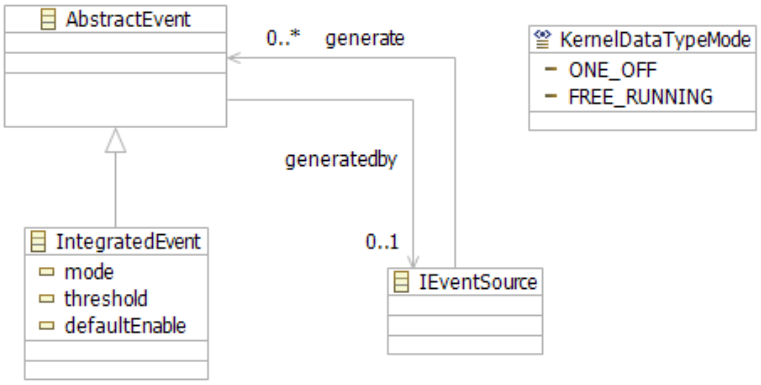


Figure 5.5: Meta-model of events

HARTEXμ uses the concept of *Integrated Event*, whereby all kinds of events are treated in a uniform fashion. An integrated event is generated by an event source, e.g. a tick interrupt or an external hardware interrupt. Events coming from a particular source are counted and a specific event-processing action is executed when the event counter expires. These actions can be: enable/disable events, enable/disable tasks, release tasks, finish tasks, and send message. The actions are modelled as the corresponding interfaces shown in Fig. 5.2. However, an action needs to be taken when an associated event has occurred for a predefined number of times. This is an attribute of an integrated event called *threshold*, whose possible value is an integer. By default, an event can be either *enabled* or *disabled* which is indicated by the attribute *defaultEnable*, whose possible values

are *true* or *false*. This parameter can be changed by disable or enable operations. The attribute *mode* of the ***IntegratedEvent*** class specifies whether an event is a *free running* or *on-off* type. If an event is of on-off type, the current event will be disabled once occurred. If it is of free-running type, the event counter of the event will be reloaded with the threshold value once a new event occurs, and start to count again.

5.2 Transformation specification

To make sure that a COMDES system is correct regarding timing requirements, a COMDES system model can be transformed into a HARTEXμ application model consisting of a number of HARTEXμ kernel nodes corresponding to network node instances in the COMDES model, such that each COMDES node is mapped to one kernel node. The latter supervises the execution of actors allocated to the network node. However, if there were no requirements related to timing, an application could be executed without a real-time kernel.

In this section, a transformation specification is given to show how to map a model or an attribute of a model in COMDES to a model or an attribute of a model in HARTEXμ . The first part focuses on the structural transformation of a COMDES system model, such as from actor to task, from signal to message, etc. The second part explains how to deal with the timing aspect of the COMDES system model. This specification can be implemented by a transformation tool in the COMDES development environment. The input of the tool is a validated COMDES system model, and the output is a HARTEXμ model.

Transformation Specification (Part 1):

1. **Source** COMDES system

   Target RTOS application

   Description For a COMDES system model,

   (a) Create a HARTEXμ application model

   (b) Transform all network nodes into kernel nodes using rule 2

   (c) Transform signals to messages using rule 6

   (d) Connect all objects for timing using rule 11

2. **Source** Network node instance

   Target Kernel node

Description For a source network node, and the created HARTEXμ application model do following:

(a) Create a kernel node model

(b) Transform the network node contained actors into tasks using rule 3

(c) Add the kernel node to the HARTEXμ application model

3. **Source** Actor

   Target Task

   Description For an actor on a network instance node, and the kernel node model transformed from the network node using rule 2, do:

   (a) Create a task

   (b) Transform the input latch and the output latch of the actor using rules 4 and 5

   (c) Assign the task a priority according to a given policy

   (d) Transform timing attributes using rules 8, 9, or 10

   (e) Add the task to the kernel node model

4. **Source** Input latch

   Target Task input

   Description For an input latch of an actor, and the task transformed from the actor using rule 3, do:

   (a) Create a task input

   (b) Add it into the task

5. **Source** Output latch

   Target Task output

   Description For an output latch of an actor, and the task transformed from the actor using rule 3, do:

   (a) Create a task output

   (b) Add it into the task

6. **Source** Signal

   Target Message

   Description For a signal, and the HARTEXμ application model transformed from the COMDES system model using rule 1, do:

(a) Create a message

(b) Transform the signal's constituent variables to message variables using rule 7

(c) Set the message source to the task transformed from the source actor

(d) Set the message target to the task transformed from the target actor

(e) If the signal releases an actor, set the message to release the task

(f) If the signal has source and target on different network node instances, set the message as global

(g) Add it to the HARTEXμ application model created using rule 1

7. **Source** Signal variable

   Target Message variable

   Description For a signal variable of a signal, and the message transformed from the signal, do:

   (a) Create a message variable

   (b) Set variable's type to the type of the signal variable

   (c) Add the message variable to the message

Under the COMDES framework, an actor must be triggered for execution. If an actor is inactive when it is triggered, the actor becomes ready for execution. Consequently, the actor could be selected to have its state changed to an executing state, in which the actor is being executed on a processor. Inactive refers to a state where the actor is not among those actors that can be selected for execution.

There are three ways to trigger an actor: periodic event, sporadic event, and signal arrival (see the meta-model of actor in Chapter 4). Each actor can be triggered by only one of them. Once an actor is triggered, it is ready to be executed, and its execution is up to a scheduler. Meanwhile, its deadline is monitored by the Event Manager, so as to release the actor's output drivers and execute them at the deadline time instant. If the deadline is zero, the output drivers are executed immediately after the actor task is finished.

In case that an actor is periodically activated, the actor is typically triggered by a timer. The duration of the period is specified by the actor attribute: *time_interval*. In this case, the deadline is measured starting from the initial time instant of the period. An actor can be triggered by some external event other than a periodic timer (e.g. a button pressed). In this case, its deadline is measured from

the time instant at which the actor has been triggered. If an actor is triggered by a signal which is sent by another actor, the two actors constitute an *actor chain*, and then the deadline should be measured from the trigger (release) instant of the first actor of the chain.

In an actor chain, except for one actor, each actor should be triggered by a signal that is sent by another actor in the same chain. The actor that is not triggered by a signal is called *head*, and the head actor can be triggered either by a periodic timing event or a sporadic event. Consequently, the whole chain is triggered by that event. The actor that is triggered by a signal and is not sending a signal triggering any other actors is called *tail*. The deadline of the chain is defined at its tail actor. The actor chain is introduced because the deadline of an actor chain is measured from release instant of the head actor in such a chain.

The simplest chain consists of one actor. Here, the head and the tail are the same. This chain can be triggered by either periodic timing event or sporadic event, and its deadline is measured from the instant when it is triggered. More complex chains are built by adding subsequent actors. A two-actor chain is shown in Fig. 5.6a, where A_2 is triggered by A_1. A_1 has the chain's trigger, while A_2 has the chain's deadline. Likewise, an n-actor chain is as shown in Fig. 5.6b, where A_1 has the chain's trigger, and A_n has the chain's deadline.

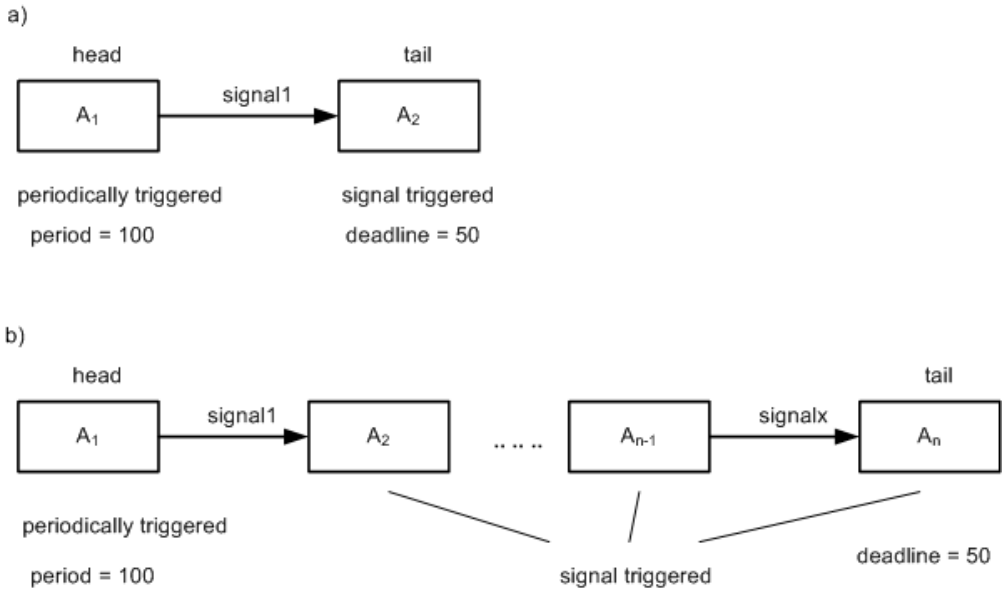


Figure 5.6: Examples of actor chain

The transformation of the timing aspect of a COMDES-based application has mainly to deal with events that trigger actors and set timers to measure deadlines

using HARTEXμ kernel objects. The timer measuring the deadline is referred to as a *deadline timer*, whereas the timer measuring the period is referred to as a *periodic timer*. In the context of the HARTEXμ kernel, an Integrated Event (IE) can be configured as either an external event or a timer, in terms of different settings, which are processed by the Event Manager module.

In an embedded system, typically, a timer measures time in multiples of tick interrupt that occurs with a fixed period (i.e. a time period of 10 milliseconds). This tick interrupt corresponds to the fine granularity of the time in the system. In the present kernel the basic tick interrupt has a time period of 10 milliseconds.

A Timer is *enabled* if it has been started and not stopped since last started, and there is a future time instant when it is expected to fire an event, else it is *disabled*. Enabling starts the timer. When time is due, some actions can be taken. The actions in HARTEXμ include: release tasks, enable timers or events, and finish tasks.

The behaviour of a *one-off timer* is that of a timer that does not automatically restart after an initial firing. It is specified in terms of the required firing interval. The behaviour of a *free running timer* is that of a timer that automatically restarts operation after an initial firing. It is specified in terms of activation period, which is equal to the firing interval.

Transformation Specification (Part 2):

8. **Source** Actor attribute: deadline

   Target Integrated event (configured as a deadline timer)

   Description For an actor, the task transformed from the actor, and the kernel node model that contains the task, if the deadline of the actor is specified (> 0), do:

   (a) Create a one-off timer

   (b) Set the timer to fire a *finish* event, and it finishes the task

   (c) Set the due time to the value of the deadline

   (d) Assign the timer to a timing source

   (e) Disable the timer by default (how this timer is enabled depends on how the actor is released)

   (f) Add the timer to the kernel node model

9. **Source** Actor attribute: trigger type is periodic

   Target Integrated event (configured as a periodic timer)

Description For an actor, the task transformed from the actor, and the kernel node model that contains the task, if the trigger type of the actor is periodic, do:

(a) Create a free running timer

(b) Set the timer to fire a *release* event for the corresponding task

(c) Set the due time to the value of the *time_ interval* of the Actor

(d) Assign the timer to a timing source

(e) Enable the timer by default

(f) Add the timer to the kernel node model

10. **Source** Actor attribute: trigger type is sporadic

    Target Integrated event (configured as an external event)

    Description For an actor, the task transformed from the actor, and the kernel node model that contains the task, if the trigger type of the actor is sporadic, do:

    (a) Create an event

    (b) Set the event to release the corresponding task

    (c) Enable the event by default

    (d) Assign the event to an event source

    (e) Add the event to the kernel node model

11. **Source** Actor attribute: deadline

    Target Connect to deadline timer

    Description For an actor and the task transformed from the actor, if the deadline is specified (> 0), do:

    (a) Find the head of the actor chain that the actor belongs to

    (b) Make the periodic timer of the head (created in rule 9) or external event of the head (created in rule 10) enable the deadline timer of the actor created in rule 8

The transformation specification mainly focuses on the timing behaviour of a COMDES system model. Other aspects of the model have been abstracted away. However, the functional behaviour of each actor modelled by function blocks can be simply represented as *code* of a HARTEXμ task, which has been abstracted away from the specification. Furthermore, it does not show how to deal with hardware-related aspects either. For example, an external event is used to release a task, but the source firing the event has to be configured from one of the interrupts of the selected hardware platform.

5.3 Example

A trivial case is given in this section to illustrate the transformation of a COMDES system model into a HARTEXμ application model. As shown in Fig. 5.7, a COMDES system model consists of two actors as well as one signal. The output driver of ***ControlActor*** is broadcasting the signal that is received by the input driver of ***DisplayActor***. The signal triggers the ***DisplayActor***.

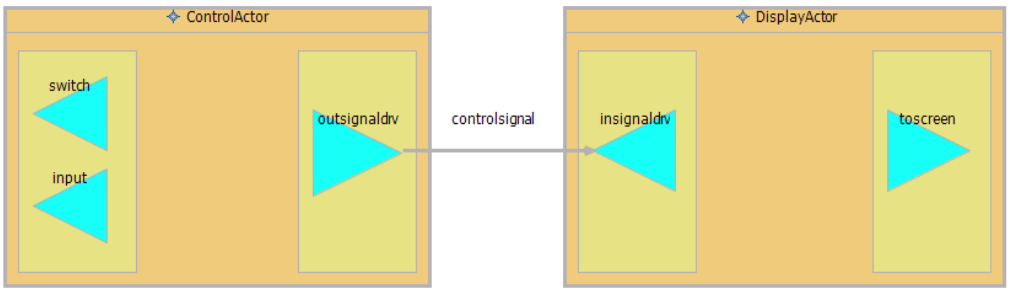


Figure 5.7: An example of COMDES system model

What is not shown visually is that the actor ***ControlActor*** is allocated to a platform called ***platform1***, whereas the actor ***DisplayActor*** is allocated to another platform called ***platform2***. The timing attributes of ***ControlActor*** are as follows:

- *trigger_type* is *periodic*;
- *time_interval* is *100*;
- and *deadline* is not specified, as it is of no concern.

The timing attributes of ***DisplayActor*** is specified as:

- *trigger_type* is *signal*, which is the received signal;
- *time_interval* is not used because the trigger type is *signal*;
- and *deadline* is *80*, which should be measured from the release time instant of ***ControlActor***.

A HARTEXμ model can be obtained after a transformation from the COMDES system model, which is shown in Fig. 5.8. The model contains two kernel nodes and one global message that are transformed from the COMDES platforms and

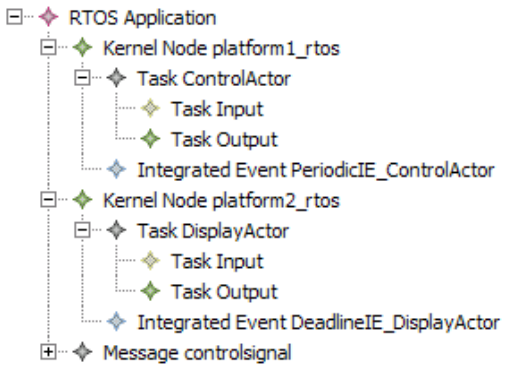


Figure 5.8: Transformed HARTEXμ application model

the signal respectively. Each node hosts a task whose input and output correspond to the drivers of the corresponding actor.

Specifically, the kernel node ***platfrom1_rtos*** (named in accordance with ***platform1*** in COMDES) has a kernel object – integrated event configured as a periodic timer. The attributes of ***PeriodicIE_ControlActor*** are as follows:

- *threshold* is *100*;
- *mode* is *free running*;
- it is *enabled* by default;
- it releases the task ***ControlActor***;
- and it enables the deadline timer called ***DeadlineIE_DisplayActor***.

The kernel node ***platfrom2_rtos*** contains the deadline timer for task ***DisplayActor***. The attributes of ***DeadlineIE_DisplayActor*** are as follows:

- *threshold* is *80*;
- *mode* is *one-off*;
- it is *disabled* by default;
- and it finishes the task ***DisplayActor*** by activating its output.

5.4 Summary

This chapter has presented a meta-model of a real-time kernel – HARTEXμ , implementing a distributed timed multitasking operational environment that has been conceived in the context of the COMDES framework. COMDES actors are scheduled dynamically by the HARTEXμ Task Manager at run-time but their I/O drivers are invoked at precisely defined time instants, thus making the application safe and predictable. A HARTEXμ model can be seen as a platform-specific model corresponding to a platform-independent COMDES model. Therefore, a transformation from the PIM to the PSM has been presented based on source and target meta-models, which focuses mainly on the timing aspect of a COMDE-based application.

The transformation is an implementation of the semantics of the COMDES DSL regarding its timing behaviour. Thus, it can be understood how the actors of a COMDES system are executed in a real-time multitasking environment. The transformation specification given in this chapter could be used as a guideline when other real-time kernels are considered to be employed as the run-time environment of COMDES applications. The ultimate idea is to enable tools to automatically generate a configuration of a distributed timed multitasking operational environment from any given COMDES model, in terms of the transformation specification.

Chapter 6

Platform-Specific Models: COMDES Implementations

The previous chapters have introduced a number of COMDES models, which are platform-independent, in the sense that models are irrelevant to programming languages. Indeed, these models are crucial for the development of embedded systems; however, according to the MDSD approach, the platform-independent models will ultimately be transformed into source code and finally – executable code either automatically or manually. While models help people specify an embedded system, code is the one running inside the system, thus it is also significant to know how the models are implemented.

This chapter focuses in particular on the design patterns of function blocks – the main kind of component in COMDES. The function block implements embedded systems functional behaviour using a programming language, which represents a lower level of abstraction than the COMDES DSL. The C language has been selected as a programming language for COMDES since it is a 'de facto' standard in embedded system programming; it is a high-level language that abstracts hardware, and at the same time provides low-level access to hardware, if necessary. As a result, these design patterns can be regarded as C platform-specific models. Thanks to the elegant design of these patterns, the reuse and instantiation of FBs as well as composition of applications can be efficiently realized in the COMDES framework.

The executable format for function blocks and COMDES-based applications will also be presented in this chapter, in order to state clearly how COMDES models are implemented. Source code based on the patterns needs to be transformed into executable code that is loaded into a system. In embedded systems, the format of the executable code is usually native binary code, because the direct execution of code in native machine form provides the best performance for

a given hardware platform. The transformation basically requires a C compiler since the FB design patterns are written in the C language. There is a variety of C compilers available, but GCC together with GNU Binutils, providing assembler and linker, have been chosen to implement the COMDES framework.

Additionally, traditional software developers are sometimes reluctant to adopt a generative approach for their development work on embedded systems because they simply do not trust the generated code. Therefore, understanding the implementation described in this chapter really makes sense to such readers. This chapter will try to make it clear through detailed discussion accompanying illustrations of code, which will hopefully build up the reader's confidence.

6.1 Function block design pattern

6.1.1 Introduction to function block implementation

Function blocks (FBs) are the main COMDES building components. At the source code level, FB can be seen as a class specifying a number of reentrant functions, as well as inputs and outputs used to exchange signals among function blocks. Function blocks are interconnected via *softwiring* using pointers to the corresponding data locations. Softwiring is conceived as an output-to-input(s) connection: output data is stored in an output buffer of the source FB, and it is subsequently accessed by one or more destination function blocks through the corresponding input pointers. This allows for efficient one-to-many connections by eliminating the need to copy source data to multiple destination inputs.

6.1.1.1 FB interface and implementation

In accordance with the definition of a software component proposed by Katharine Whitehead (see Chapter 1), there is a distinction between two perspectives of a COMDES function block: FB interface and FB implementation. The FB interface summarizes the properties of a FB that are externally visible to the other parts of the system. All communications between COMDES function blocks are carried out through interfaces that are defined as a set of input pointers and a set of output buffers. (The concept of interface in COMDES is different from the one usually used in the Object-Oriented world where an interface consists of a set of functions, which are invoked by other components.) The FB implementation is the executable realization of a FB and it must conform to the properties stated in their interfaces.

The separation between interface and implementation guarantees FB independence. The real implementation of a function block is hidden behind its interface. The separation also provides a way of updating a function block without influencing other components that interact with the function block – as long the interface is not changed, resulting in greater flexibility with respect to possible updates. According to COMDES, the implementation provided to users should be in the form of compiled objects rather than source code. In this manner, the components can be directly assembled in an application without compilation. Furthermore, shipped components are not allowed to be changed by component users, thus eliminating possible errors due to manual changes of components source code.

6.1.1.2 FB functions – reentrant functions

Each function of a FB should be defined as reentrant code, so it can be preempted in a multitasking environment where a higher-priority task can preempt a lower-priority task. A reentrant function can be interrupted at any time and resumed at a later time without loss of data. Reentrant functions either use local variables or protect their data when global variables are used [81].

To implement a COMDES-compliant component, there are three rules that apply to the implementation of a FB function, so that the function is reentrant:

- A function cannot use variables in a non-atomic way unless they are stored on the stack (local variables) or are otherwise the instance variables stored in the function block execution record (a section of code is atomic if it cannot be interrupted).
- A function cannot call any other functions that are not themselves reentrant.
- A function cannot use the hardware in a non-atomic way.

As a result of the above guidelines, only local variables and members of an execution record can be used within a function.

Things become complicated when function blocks use a third-party library, because it is never known, which parts of the library are reentrant and which are not. However, nowadays vendors have taken the initiative to provide reentrant versions of libraries, and therefore, in the following discussion it is assumed that third-party libraries are reentrant.

6.1.1.3 Function block type and instance

Function blocks are reusable due to their particular features combining functionality and data: type and instance. The FB type defines FB behaviour (the func-

tions). FB instances just store the values for the variables of the corresponding FB type. Typically, there are many instances of a function block of a single type. FB instances of the same FB type share the same behaviour and only differ in the data they operate on. The behaviour is defined by the corresponding FB type.

The FB instance has a kind of "memory" – the so called function block execution record, which can store local data values, parameters and inputs/outputs over several invocations. Such a memory is important for FBs as their behaviour is dependent on the current status of the internal values (persistent variables). All such data are stored in a static memory area which is assigned to each FB instance.

6.1.2 Basic function blocks

Each basic FB type must have one interface and one implementation. An interface is in principle a data structure encapsulating significant execution attributes of a given type of FB, including inputs, outputs, parameters as well as persistent internal variables. Listing 6.1 presents a data structure that implements the basic FB interface, where the basic FB execution attributes are declared. The `TypeName` in the name of the declared interface template should be replaced by the actual FB type name when a specific type is defined.

Listing 6.1: Basic function block interface pattern

```
/* output structure */
typedef struct {
   /* variables, example: BYTE result; */
} TFBTypeNameOutput;

/* FB interface structure */
typedef struct {

   /* input structure */
   struct {
      /* pointers, example: BYTE *data; */
   } input;

   /* parameter structure */
   struct {
      /* pointers, example: BYTE *parameter; */
      /* BYTE *internal_variable;            */
   } parameter;

   TFBTypeNameOutput *output;
```

```
} TFBTypeName;
```

The interface defines the function block data structure `TFBTypeName` which contains an input structure, a parameter structure and an output pointer. FB outputs are placed into an output buffer containing one or more variables defined by the structure `TFBTypeNameOutput`. When a specific FB instance is created, its interface as well as output buffer will be instantiated with appropriate values.

For the purpose of the softwiring connection, all data in the input and parameter structures are defined as pointers, in order to provide connection to another function block output or another data buffer (i.e. a constant). Input pointers are used to be connected to FB output buffers or to constants, through which a FB can get input data for computation. The output buffers encapsulating a set of variables stores the computation results. Parameter pointers can be connected to static variables if they are constant during the run-time. Internal persistent variables of a FB type should be also defined within the interface and be accessed by pointers that are included in the parameter structure.

Listing 6.2 illustrates a FB implementation skeleton consisting of a number of FB functions as well as the declaration of a FB type data structure – `TFBTypeNameFunctions`, through which a specific function can be accessed. The implementation pattern is in fact generic for all kinds of COMDES FBs. Different kinds of FBs only differ in their interfaces and concrete functions.

Listing 6.2: Function block implementation pattern

```
typedef void(*TFBFunction)(void*);

/* FB type data structure */
typedef union {

   /* function structure */
   struct {
      TFBFunction init;
      TFBFunction main;
      TFBFunction exit;
   };

   TFBFunction function[3];

} TFBTypeNameFunctions;

/* definition of a FB function: init */
void FBTypeNameInit(void* FB)
{
   /* local FB pointer used to access FB instance data */
```

```
    TFBTypeName* lFB = (TFBTypeName*)FB;

    /* Start of function code: init */

    ... ...

    /* End of function code: init */
}

/* definition of a FB function: main */
void FBTypeNameMain(void* FB)
{
    /* local FB pointer used to access FB instance data */
    TFBTypeName* lFB = (TFBTypeName*)FB;

    /* Start of function code: main */

    ... ...

    /* End of function code: main */
}

/* definition of a FB function: exit */
void FBTypeNameExit(void* FB)
{
    /* local FB pointer used to access FB instance data */
    TFBTypeName* lFB = (TFBTypeName*)FB;

    /* Start of function code: exit */

    ... ...

    /* End of function code: exit */
}

/* declaration of FB Type */
TFBTypeNameFunctions FBTypeNameFunctions =
{ {
    (TFBFunction) FBTypeNameInit,
    (TFBFunction) FBTypeNameMain,
    (TFBFunction) FBTypeNameExit,
} };
```

The FB type data structure has two alternative options to access the associated functions: one is through explicitly named function pointers (i.e. `init`, `main` or `exit`); another is via an array of function pointers whose index represents the

corresponding function (i.e. `function[0]` denotes the associated `FBTypeNameInit` function.) Similar to the interface pattern, the `TypeName` part of each name should be replaced by the actual FB name.

Each FB type should have at least one function, where code for the computational algorithm should be placed. (The example implementation pattern defines three functions). A FB function (e.g. `FBTypeNameMain`) accepts a pointer referring to the address of a particular instance, in order to acquire the needed execution information. In this way, the application-specific data stored in FB instances is separated from the application-independent functions defined as FB types. As a result, predefined components can be reused in different applications.

A function needs to be registered as a part of FB type data structure. A global variable (`FBTypeNameFunctions`) is declared through which application code can access functions of a FB type.

If a developer is implementing a reusable basic FB type manually without using any model and code generator, he/she firstly needs to fill out the interface pattern with specific inputs, outputs, and parameters. Keep in mind that the persistent internal variables are also implemented as parameters. Secondly, he/she has to write concrete functions manually in C code and must comply with the rules of reentrant functions. Finally, these functions should be registered into the FB type data structure.

Listing 6.3 exemplifies the use of the pattern where an FB instance is created according to a given type. An output buffer is created, which can be accessed by other FB instances. A FB is instantiated by creating a variable using the type defined in the interface, and an appropriate value for each pointer must be assigned.

Listing 6.3: Basic function block instance pattern

```
/* create a FB instance output buffer */
TFBTypeNameOutput instanceOutput = {/* output initial values */};

/* create a FB instance */
TFBTypeName instance = {
   /* input structure */
   {/* addresses of connected FB outputs */},

   /* parameter structure */
   {/* addresses of parameter variables */},

   /* output */
   &instanceOutput,
};
```

Listing 6.4 demonstrates how to execute the FB instance by using the specific function of a given FB type. In the example, the main function is invoked to process the specified FB instance (also called `instance` in this example) by using the FB type variable – `FBTypeNameFunctions`.

Listing 6.4: Basic function block execution pattern

```
/* execute the main function of a given FB type on the instance */
FBTypeNameFunctions.main(&instance);
```

6.1.3 Drivers

Patterns for input drivers and output drivers are also implemented similar to the basic FB kind pattern. However, the interface pattern of the input driver will not have the input structure (input pointers), because it reads data from the external environment (i.e. peripheral device or operational environment, by invoking the corresponding service routine or communication primitive) rather than the output buffers of other function blocks. For similar reason, the interface pattern for the output driver does not have an output structure as well as a corresponding output pointer.

The driver implementation pattern is the same as the one used with basic function blocks. However, the input/output drivers are assumed to be short pieces of code executed atomically. The assumption might be considered valid since drivers are implemented as wrappers copying data from one location to another one without significant computation. These are typically dedicated routines servicing specific peripherals (with physical I/O drivers) or kernel communication primitives (with communication I/O drivers).

6.1.4 Composite function blocks

A composite function block (CFB) encapsulates a number of FB instances that are suitably interconnected to constitute a data flow model, i.e. a function block diagram (FBD). The CFB executes encapsulated FB instances according to a static execution schedule, i.e. a linear sequence, which is derived from the encapsulated function block diagram when transforming a CFB model into code, using the topological sort algorithm [75]. In principle, the execution schedule of a given function block diagram is a table composed of records, each of which specifies the execution of the designated function of a given FB type for the corresponding FB instance (see Listing 6.5).

Listing 6.5: FBD table record structure

```
typedef unsigned char UBYTE;
typedef UBYTE TFBType;
typedef UBYTE TFBFunctionType;
typedef void* TFBInstance;

typedef struct {
   TFBType           type;
   TFBFunctionType   function;
   TFBInstance       instance;
} TFBDiagram;
```

The FBD table record is defined as a structure named `TFBDiagram`, consisting of three fields that specify the internal components to be executed, as follows:

- `type`: specifies the type of an encapsulated function block
- `function`: specifies which function of the encapsulated function block should be executed
- `instance`: specifies which instance of the encapsulated function block should be executed

Listing 6.6: Composite function block interface pattern

```
/* Composite FB interface structure */
typedef struct {

   TFBDiagram* FBdiagram;

} TFBComposite;
```

Accordingly, the CFB interface pattern consists of a single item – `FBdiagram`, which is a pointer to the first record of the FBD table (see Listing 6.6). The inputs, parameters and outputs of a CFB type are not necessarily specified explicitly in the CFB interface structure, since each internal signal going into a CFB input is actually passed to one of its encapsulated FB inputs, whereas each internal signal going out from a CFB output is in fact passed from one of its encapsulated FB outputs. In the meta-model, such input-to-input and output-to-output connections are modelled using the ***ExtendInput*** and the ***ExtendOutput*** relations (see Chapter 4 – Section 4.2).

The FB function implementation pattern of the CFB kind is the same as the one for the basic FB, except that one of the functions must be the standard and reusable driver – `CFBKindDriver` used to execute the function block diagram.

The algorithm of the driver allows a CFB to interpret and execute its associated execution schedule via the corresponding instance interface (Listing 6.7). This algorithm accepts a CFB instance as an input parameter, scans the corresponding execution schedule table and executes the encapsulated FB instances by invoking the designated functions of the constituent FB types, until reaching the end of the table denoted by a `NULL` encoding of the corresponding instance field.

Listing 6.7: Composite function block driver

```
/* definition of a FB function */
void CFBKindDriver(void* FB)
{
   TFBComposite* lFB   = (TFBComposite*) FB;
   TFBDiagram*   lFBD  = lFB->FBdiagram;
   while(lFBD->instance!=NULL){
      /* execute function on an instance of a FB type,  */
      /* as specified by the current FB diagram record; */
      (FBTypes[lFBD->type])->
            function[lFBD->function](lFBD->instance);
      lFBD++;
   }
}
```

The `FBTypes` is a global static table containing pointers to data structures defining the type functions for all FB types used in a system. It is used to find the start address of a particular function of a given type, which is then invoked with a pointer to the instance record, as specified by the current line of the FB diagram table (see Fig. 6.1).

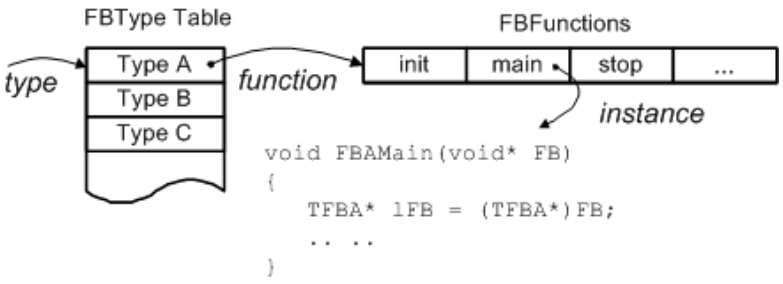


Figure 6.1: Function block type table

6.1.5 State machine function blocks

A state machine FB can be implemented using a new version of the State Logic Controller (SLC) design pattern originally introduced in [82]. The SLC is built

around a data structure that contains the computer image of the state transition graph. It can be efficiently implemented as a table containing modified (multiple-output) binary decision diagrams that represent the next-state mappings of various states within the state transition graph.

The next-state mapping of state s is defined as a subset of states $Fs = s'$ that are immediate successors of s in the state transition graph. Accordingly, the state transition graph can be specified by defining the next-state mappings of all states $s \in S$, whereby transition arcs are specified by tuples $(s, s'|s' \in Fs)$ labelled by the corresponding combinations of transition trigger and transition-order symbols.

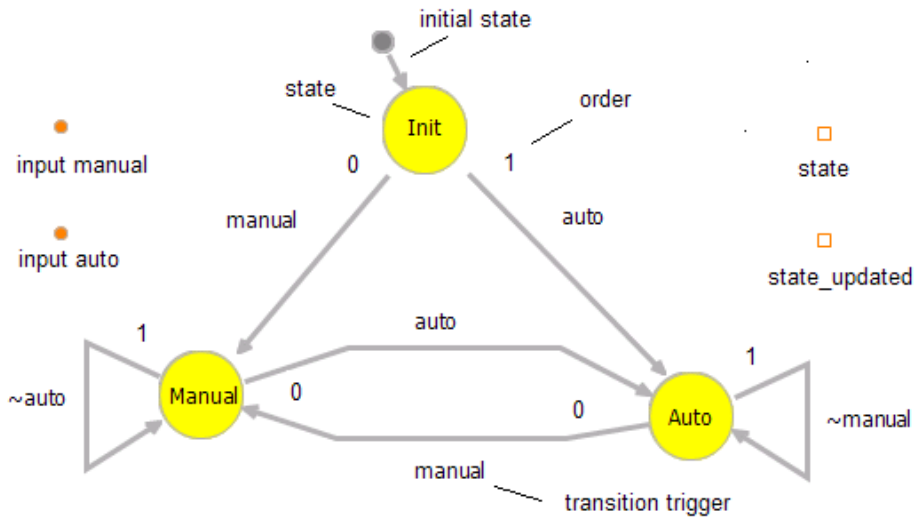


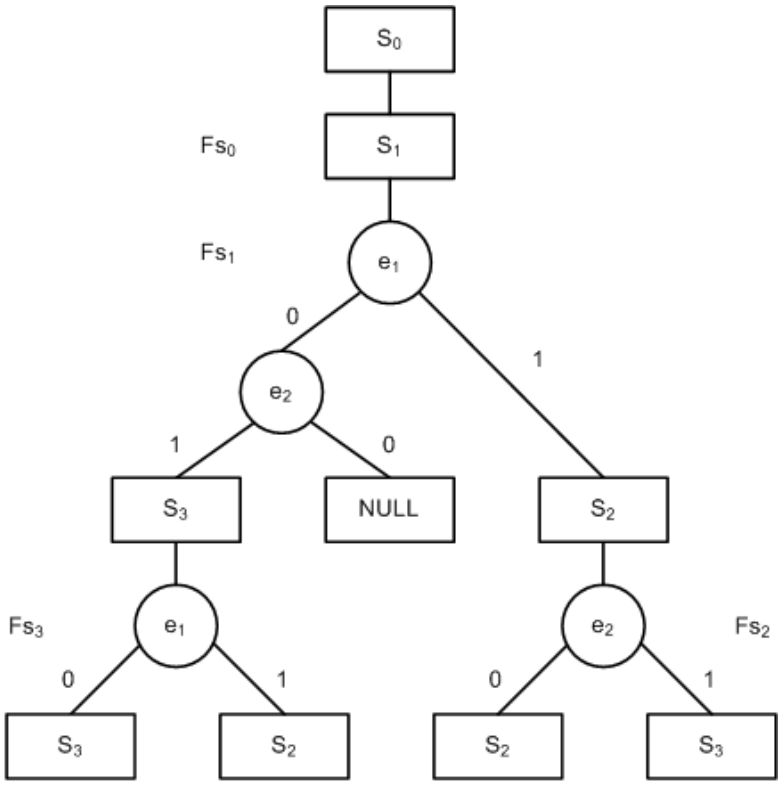
Figure 6.2: DC Motor Control System: mode change control state machine

This technique will be illustrated with a simple tutorial example, i.e. a Mode Change Control state machine, which constitutes the upper level of a hierarchical state machine used to control a DC motor (see Fig. 6.2). Its state transition graph can be represented as follows:

$$\begin{aligned} Fs_0 &= s_1 \\ Fs_1 &= s_2(e_1, 0), s_3(e_2, 1) \\ Fs_2 &= s_3(e_2, 0), s_2(\neg e_2, 1) \\ Fs_3 &= s_2(e_1, 0), s_3(\neg e_1, 1) \end{aligned}$$

where s_0 denotes the initial pseudo-state, s_1, s_2 and s_3 denote states ***Init***, ***Manual*** and ***Auto*** respectively, e_1 and e_2 denote transition triggers ***manual*** and ***auto***, $\neg e_1$ and $\neg e_2$ denote the absence of the corresponding triggers, and bracketed expres-

sions – the corresponding <transition trigger – transition order> combinations.



Row	Node	succTrue	succFalse	Next-state mapping
0	S_1	1	--	Fs_0
1	e_1	3	2	Fs_1
2	e_2	4	5	
3	S_2	6	--	
4	S_3	7	--	
5	NULL	1	--	
6	e_2	4	3	Fs_2
7	e_1	3	4	Fs_3

Figure 6.3: Binary decision diagrams for next state mappings

Next-state mappings can be graphically represented by means of modified (multiple-output) binary decision diagrams, as shown in Fig. 6.3 for the example state machine. In these diagrams, circular nodes denote signals (transition triggers) that have to be tested in order to determine the current state to be activated from among the subset of successor states of the previously active state. These

are tested in a predefined sequence that reflects the order of the corresponding transitions.

For example, it is possible to make a transition from s_1 to either s_2 or s_3, whereby the former transition has higher importance, i.e. lower transition order index than the other one. That is encoded in the Binary Decision Diagram (BDD) whereby the trigger e_1 is checked first and the transition to s_2 – taken if e_1 is true; the transition to the s_3 will be taken only if e_1 is false and e_2 – true. In case neither of the trigger signals is present, the parsing of the BDD ends up in a NULL node, meaning that no transition is taken and the previous state has to be maintained in the current period of execution.

The binary decision diagrams of the next-state mappings can be encoded in a *State Transition Table*, as shown in Fig. 6.3. The table consists essentially of the columns *Node*, *successorTrue* and *successorFalse*, whereby the first column (*Node*) contains symbols denoting BDD nodes, and the other two columns – pointers to rows containing the corresponding BDD elements. The rows are grouped into segments containing the next-state mappings of states s_0, s_1, s_2, s_3.

The State Transition Table can be interpreted much in the same way as its graphical counterpart. In particular, it can be processed by a standard routine – a *state machine driver*, which is activated periodically by the corresponding host actor. Within each cycle, the driver processes the BDD containing the successor states of the state visited in the previous cycle, in order to determine the current state. If a state transition has taken place, the *state* and *state-updated* variables are modified accordingly, and an associated MFB is subsequently invoked to execute the corresponding state action. However, the action is executed only when the state is visited for the first time, and will not be executed in subsequent cycles if the state is maintained, unless a self transition is explicitly specified. Consequently, the state machine has event-driven execute-once semantics even though it is periodically activated.

For the purpose of implementation, the state transition table can be encoded using the row format shown in Listing 6.8, where `testedSignal` is a pointer to an object containing the values of tested binary signals (i.e. a signal driver or a preprocessing function block), `mask` denotes the bit position of the signal tested. A `NULL` value of that pointer denotes a state node, whereby the second field contains the state index. A `NULL` value of the `state` field denotes a situation where the state is not changed due to the absence of transition triggers (also called `STATENULL`). The other two fields contain the `successorTrue` and `successorFalse` variables. In the case of a state node, the first field contains a variable `nextState` denoting the first line of the corresponding next-state mapping.

Listing 6.8: State transition table structure pattern

```
typedef UBYTE TTestedSignal;
typedef UBYTE TSTTRow;
typedef UBYTE TState;

typedef struct {
   TTestedSignal* testedSignal;
   union {
      TTestedSignal  mask;
      TState         state;
   };
   union {
      TSTTRow   successorTrue;
      TSTTRow   nextState;
   };
   TSTTRow   successorFalse;
} TSTTRecord;
```

The state transition table of a state machine FB is integrated with its interface pattern through a pointer, as shown in Listing 6.9. In this state machine FB interface definition, the `recordTtable` points to a state transition table. The field `history` is a persistent internal variable used to store the row index of the first line of the next-state mapping that will be processed in the next cycle of execution. As the state machine has only two outputs: `state` and `state_updated`, they are declared directly in the interface without using an extra output structure. The inputs do not have to be specified in the interface structure, due to the fact that they are defined in the corresponding `testedSignal` fields of the state transition table.

Listing 6.9: State machine function block interface pattern

```
typedef struct {
   TSTTRecord* recordTtable;
   TSTTRow     history;
   BOOL        state_updated;
   TState      state;
} TFBStateMachine;
```

The implementation part of the reusable and reconfigurable state machine FB is the same as the implementation pattern of the basic FB. However, the state machine driver – `SMFBKindDriver` is required as a function in order to process the state transition table and execute the state machine (see Listing 6.10).

Listing 6.10: State machine function block driver

```
typedef STATENULL (0xFF);
```

```
void SMFBKindDriver(void* FB)
{
   TFBStateMachine* lFB  = (TFBStateMachine*) FB;
   TSTTRecord*      row  = lFB->recordTtable;
   TTestedSignal*   ts_pointer;
   TState           current_state;

   /* restore row pointer */
   row = &lFB->recordTtable[lFB->history];

   /* determine current state/mode and update output */
   do {
      ts_pointer = row->testedSignal;

      /* in case of a transition trigger */
      if(ts_pointer != NULL){
         if(*ts_pointer && row->mask)
            row = &lFB->recordTtable[row->successorTrue];
         else row = &lFB->recordTtable[row->successorFalse];
      }

      /* in case of a state */
      else {
         /* get current state */
         current_state = row->state;
         /* if a NULL state, state does not change */
         if(current_state == STATENULL){
            lFB->state_updated = 0;
            return;
         }
         /* if a real state, state changes */
         else {
            lFB->state = current_state;
            lFB->state_updated = 1;
         }
         lFB->history = row->nextState;
         return;
      }
   } while(TRUE);
}
```

Based on the patterns, the state machine instance implementing the example of Fig. 6.2 can be implemented as illustrated by Listing 6.11, where `sm_table_mySM[]` is an array of the `TSTTRecord` instances that is precisely consistent with the state transition table shown in Fig. 6.3. The `e_1` and `e_2` denote two binary inputs

of the state machine instance, and they are connected to two outputs of a FB called `myFBOutput` respectively. In one of the records, a `STATENULL` state denotes the BDD node that is reached when no transition trigger is present in the current cycle of execution. At the end of this code, an instance of state machine FB is created using the defined table.

Listing 6.11: A state machine BDD table and instance

```
#define INIT_BDD_STATE 0
#define MANUAL_BDD_STATE 1
#define AUTO_BDD_STATE 2
#define e1 (&myFBOutput.Manual)
#define e2 (&myFBOutput.Auto)

TSTTRecord sm_table_mySM[] = {
   { NULL, {INIT_BDD_STATE},   {4},  {0} } ,
   { NULL, {MANUAL_BDD_STATE}, {6},  {0} } ,
   { NULL, {AUTO_BDD_STATE,    {7},  {0} } ,
   { NULL, {STATENULL},        {-1}, {0} } ,
   { e1,   {0xff},             {1},  {5} } ,
   { e2,   {0xff},             {2},  {3} } ,
   { e2,   {0xff},             {2},  {1} } ,
   { e1,   {0xff},             {1},  {2} } ,
};

TFBStateMachine   mySM = {sm_table_mySM,0,0,0};
```

It is possible to automatically synthesize the state transition table implementation from the example state machine model. The key of the synthesis procedure is the translation from each transition trigger expression into a corresponding BDD section. Rolf Drechsler and his colleagues (Institute of Computer Science, University of Bremen, Germany) have developed a Java implementation of a BDD package [83], which may facilitate the implementation of a state machine code generator using the Java language.

6.1.6 Modal function blocks

A modal FB is in principle an assembly component containing multiple operational modes, where each operation mode executes a specific control action specified with constituent function block diagram. The modal FB has two mandatory input signals – `enabled` and `mode`. These are used by a function selecting a FB diagram to execute as indicated by the input signal mode, if its input signal enabled is *true*. The corresponding inputs are typically connected to state machine FB outputs: `state` and `state_updated` respectively.

From an implementation point of view, each operation mode of a modal FB encapsulates a function block diagram whose execution schedule can be statically determined before run-time. The execution table of a modal FB consists of records defined by the structure – `TFBModalModeFBDMappingTable` (see Listing 6.12). The latter contains a field – `modeIndex` that specifies the mode associated with a particular function block diagram. This index is compared with the mode input, in order to determine which FB diagram should be executed. The `FBD` is a pointer to the first record of a FBD table that specifies which function block instances will be executed in this mode.

Listing 6.12: Mode-to-FBD mapping table pattern

```
typedef struct {
   UBYTE         modeIndex;
   TFBDiagram*   FBD;
} TFBModalModeFBDMappingTable;
```

Once again, the execution table of a modal FB shall be integrated within the interface pattern of a MFB through a typed pointer – `exeTable` (see Listing 6.13). The interface has two input pointers `enabled` and `mode`. The structure `TFBModalOutput` defines the outputs of a MFB. The `updateTable` is a pointer to a table used to update the outputs of a MFB after the execution of a mode.

Listing 6.13: Modal function block interface pattern

```
typedef struct {
   TFBModalModeFBDMappingTable* exeTable;
   BOOL*            enabled;
   UBYTE*           mode;
   TFBModalOutput*  output;
   TFBModalUpdateOutput* updateTable;
} TFBModal;

typedef struct {
   UBYTE modeIndex;
   void* internalFBOutput;
}TFBModalOutputMap;

typedef struct {
   void*   output;
   UBYTE   outputSize;
   TFBModalOutputMap* map;
}TFBModalUpdateOutput;
```

The update table is defined by the structure `TFBModalUpdateOutput`. It maps the outputs of internal FB instances encapsulated in constituent modes onto the

outputs of the MFB, in accordance with the corresponding ***ExtendSharedOutput*** or ***ExtendOutput*** relations. In the `TFBModalUpdateOutput` structure, the `output` is a pointer to an output of a MFB; the `outputSize` is the length of the output; the `map` is a pointer to a mapping table storing the pointers to the output buffers of all internal FBs whose outputs are connected to the MFB's output, and the modes these FBs are encapsulated in. That table is defined by the `TFBModalOutputMap` structure.

The other MFB inputs are not specified explicitly in the MFB interface structure, because each internal signal going into a MFB input is actually passed to one of its encapsulated FB inputs. In the meta-model, such input-to-input connections are modelled using the ***ExtendInput*** or the ***ExtendSharedInput*** relations.

Regarding the function of a MFB, the `MFBKindDriver` is used to execute the execution schedule table that specifies constituent function blocks to be executed in a particular mode of operation (see Listing 6.14). In this algorithm, `FBTypes` is the global table storing all FB types used in a system, as explained before. The `memcpy` is a standard C function that copies the supplied number of bytes between two memory locations.

Here, the outputs of the MFB are synchronized only with those values computed by the specific internal FB instances that have been executed in the corresponding mode, by looking up the corresponding update table. If a MFB is disabled in the current cycle of execution ($enabled = false$), internal function block instances will not be executed and the outputs will be the values computed in the previous execution cycle. Ultimately, the MFB outputs will retain the values obtained in the last cycle when the MFB was enabled for execution.

Listing 6.14: Modal function block driver

```
void MFBKindDriver (void* FB)
{
    TFBModal*    lFB = (TFBModal*) FB;
    TFBModalUpdateOutput* lFBU = lFB->updateTable;
    TFBModalModeFBDMappingTable* lFBE = lFB->exeTable;
    TFBDiagram *fbd;
    TFBModalOutputMap* map;

    if(*lFB->enabled == TRUE){
       if(lFBE!=NULL){
          while(lFBE->FBD!=NULL) {
             if(lFBE->modeIndex==*lFB->mode){
                if(lFBE->FBD==NULL) break;
                fbd = lFBE->FBD;
                while(fbd->instance!=NULL){
                   FBTypes[fbd->type]->
```

```
                    function[fbd->function](fbd->instance);
                fbd++;
            }
            break;
        }
        lFBE++;
    }
}

/* update each output */
while(lFBU->output!=NULL){
    map = lFBU->map;
    while(map->modeIndex!=NULL){
        if(map->modeIndex == *lFB->mode){
            if(map->internalFBOutput!=NULL){
                memcpy(lFBU->output,
                       map->internalFBOutput,
                       lFBU->outputSize);
            }
            map++;
            break;
        }
        map++;
    }
    lFBU++;
}
}
}
```

For the purpose of demonstration of the above-mentioned patterns, the code of the example MFB in Fig. 6.4 is given in Listing 6.15 and Listing 6.16.

The MFB contains only one output (`outsignal`) that is connected to the outputs of three FB instances encapsulated in three different modes through a shared output. Thus, there is only one entry in the update table (`myMFB_updateTable`), which maps the output onto the three connected outputs of the constituent FB instances, by means of the mapping table (`myMFB_outsignal_output_map`). Accordingly, if a MFB contains more outputs, more mapping tables and more entries in the update table are required. An entry with all `NULL` values denotes the end of the table.

Listing 6.15: **Modal function block driver**

```
/* output */
typedef struct {
   UBYTE  outsignal;
} TFBModalOutput;
```

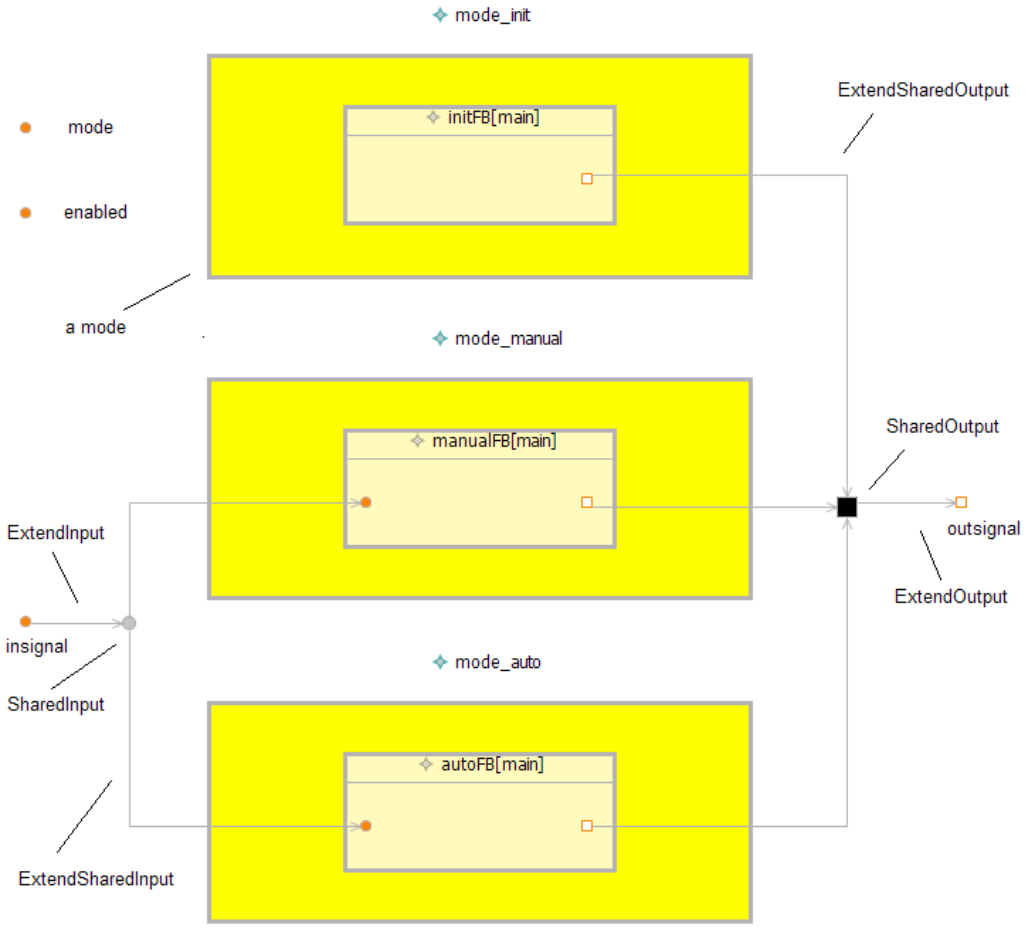


Figure 6.4: A Modal FB

```
/* output */
TFBModalOutput myMFBOutput = { 0 };

TFBModalOutputMap myMFB_outsignal_output_map[] = {
   {MODE_INIT,  (void*)&initFBOutput.output} ,
   {MODE_MANUAL,(void*)&manualFBOutput.output},
   {MODE_AUTO,  (void*)&autoFBOutput.output},
   {NULL, NULL},
};

TFBModalUpdateOutput myMFB_updateTable[] = {
   {(void*)&myMFBOutput.outsignal,
     sizeof(myMFBOutput.outsignal),
```

```
    myMFB_outsignal_output_map },
  {NULL, NULL, NULL},
};
```

The execution table `myMFB_table` specifies the mapping between the three modes and the three function block diagrams encapsulated in the three modes respectively, identified by the corresponding mode indexes (see Listing 6.16), e.g. `MODE_INIT`, `MODE_MANUAL` and `MODE_AUTO`. Through this table, each function block diagram can be located at the corresponding start address. In this example, function block diagrams are instantiated separately, and each of them consists of only one basic FB instance.

The execution table `myMFB_table` is specified together with the update table `myMFB_updateTable` when instantiating a MFB – an instance `myMFB` is created as shown in Listing 6.16. Additionally, the MFB instance encapsulates three basic FB instances – `initFB`, `manualFB` and `autoFB` in the corresponding modes of operation that have to be specified as well.

Listing 6.16: **A MFB execution table and instance**

```
TFBDiagram myMFB_exe_schedule_mode_init[] = {
  { FBINITFB, FBINITFB_MAIN, (void*)&initFB },
  { 0, 0, NULL},
};

TFBDiagram myMFB_exe_schedule_mode_manual[] = {
  { FBMANUALFB, FBMANUALFB_MAIN, (void*)&manualFB },
  { 0, 0, NULL},
};

TFBDiagram myMFB_exe_schedule_mode_auto[] = {
  { FBAUTOFB, FBAUTOFB_MAIN, (void*)&autoFB },
  { 0, 0, NULL},
};

TFBModalModeFBDMappingTable myMFB_table[] = {
  {MODE_INIT,   myMFB_exe_schedule_mode_init},
  {MODE_MANUAL, myMFB_exe_schedule_mode_manual},
  {MODE_AUTO,   myMFB_exe_schedule_mode_auto},
  {NULL, NULL},
};

TFBModal myMFB = {
  myMFB_table,&mySM.state_updated, &mySM.output,
  &myMFBOutput, myMFB_updateTable
};
```

6.2 Actor pattern

Actors in COMDES are not designed as reusable components. Therefore each actor should be created specific to a particular application. Hence, it is not necessary to distinguish actor instance from type.

As a unit of concurrency, an actor has both functional and timing aspects. From the functionality point of view, an actor executes input drivers in its input latch, followed by the execution of the signal processing block and the output drivers in its output latch (see Listing 6.17). The signal processing block is modelled by a FB diagram, which is mapped to a real-time task. Therefore, the FB instances encapsulated in that task should be executed in a liner sequence, which can be once again derived directly from the actor model using the topological sort algorithm [75].

Listing 6.17: **A MFB execution table and instance**

```
void ActorInputLatch (void) {
        / * invoke input driver FB instances */
}

void ActorSignalProcessingTask(void) {
    /* invoke constituent FB instances according to */
    /* their liner sequence of execution.           */
}

void ActorOutputLatch (void) {
        /* invoke output driver FB instances */
}
```

From the timing point of view, it is the underlying run-time environment, which manages the execution of these three functions by invoking them at particular time instants. Thus, there is no specific pattern at the actor source code level, since various environments might be used to achieve the required timing behaviour. However, in order to satisfy the timing requirements of a COMDES actor, drivers should be executed separately from the actor task. Specifically, input drivers are executed when the task is released, whereas output drivers are executed when the task deadline arrives if its deadline specified. If no deadline has been specified, the output drivers are executed immediately after the task is finished. Any run-time environment (i.e. real-time operating system) that can meet these requirements can be used to execute an actor as well as its latches. The HARTEXμ kernel is one example of such a run-time environment. It is possible to generate C code from a validated HARTEXμ model. However, kernel generation is not in the focus of this thesis. More information about the HARTEXμ kernel and its implementation in

the C programming language can be found in [29].

6.3 Down to executable code

Embedded systems are usually very limited in operational resources, e.g. memory, processor capacity, and the target platform does not have a native set of development tools, or does not have the necessary resources to perform the compilation, which is often the case. Therefore, it is natural that compile-time operations are carried out at a host platform where development takes place, whereas run-time tasks are executed at a target platform, in other words, in the embedded system. This kind of development approach is called *cross-development* meaning that the host system has to be able to produce executable code for another platform.

In a cross-development environment, an executable obtained from a COMDES model is truly platform-specific, where the term "platform" refers to hardware with a processor used to execute binaries. In a broader sense, it can include a software tool-chain used to generate the native binaries. However, as mentioned earlier, GCC with other GNU tools have been chosen as the tool-chain for building a COMDES application for a particular architecture, in order to target a wide variety of platforms. GCC is available for most embedded platforms and it target processor families include: ARM, Atmel AVR, Blackfin, H8/300, Motorola 68000, MIPS, etc. Chip manufacturers today consider a GCC port almost essential to the success of a product.

6.3.1 Binaries of function block and application

After processing by a GCC compiler, a function block implemented according to the corresponding design pattern is transformed into a relocatable object file. The object is the eventual format of a COMDES component stored for reuse. If selected in an application, it will be directly linked with other FB objects into an executable by a linker.

A relocatable file is an object file that holds code and data suitable for linking with other object files to create an executable. The relocatable file contains extensive symbol and relocation information needed by the linker along with object code. The object code is often divided up into many small logical sections that will be treated differently by the linker. Each relocatable file is translated as if it will reside at location zero with a symbol table showing which values in the file will need to change if it is moved to some location other than zero. The linker adjusts these values to be appropriate for where the code and data will actually

reside.

An executable file contains executable code that can eventually be run on a target hardware platform. The executable code of a COMDES application consists of prebuilt reusable components in the form of data structures and executable functions. The former represent component instances and the latter – component types; each particular instance has a corresponding data structure and is associated by its type with a number of functions.

An executable file is an object that holds a program suitable for execution. An executable file is directly generated by linking the object file, which has all relocation done and all symbols resolved (shared library symbols to be resolved at run-time with dynamic linking is not taken into account in the context of embedded software development). It is capable of being loaded into memory and run as a program, i.e. contains object code, but does not need any symbols, and needs no relocation information. The object code is a single large segment or a small set of segments that reflect the hardware execution environment.

An example of object file is the Executable and Linking Format (ELF) that was originally developed and published by UNIX System Laboratories as part of the Application Binary Interface. The ELF standard is intended to streamline software development by providing developers with a set of binary interface definitions that extend across multiple operating environments [84]. The objects in ELF format can be handled by the GNU C compiler, linker and binary utilities. Normally, the binary ELF executable generated by the linker, can be directly executed in the UNIX or Linux system. ELF has also seen some adoption in non-Unix embedded system. However, in an embedded system which usually does not accept the binary executable, an ELF file has to be translated into another format, e.g. Intel HEX or Motorola SREC and can be downloaded into the target. Therefore, it requires an extra tool to perform the transformation such as the GNU objcopy utility.

The information about the tools used for transformation from source code to image i.e. compiler, linker, etc. can be specified by the *reqiured_toolchain* attribute of a network node instance or type model (see Chapter 4).

6.3.2 Build scripts

In the field of computer software, the term "build" refers to the process of converting source code files into standalone software artefacts that can be run on a processor. The process of building a program is usually managed by a build tool that coordinates and controls other programs. Examples of such a program are *make*, *ant*, and *maven*, etc. The build utility needs to compile and link the various

files, in the correct order. The goal of such an automated build tool is to create a one-step process for turning source code into a working system. This is done to save time and to reduce errors.

Make is a utility for automatically building executable programs and libraries from source code. GNU make is frequently used in conjunction with the GNU build system that is part of the GNU tool-chain and is widely used in many free software and open source packages. The tool reads its instructions from text files called *makefiles* specifying how to derive the target program from each of its inputs.

The makefile containing instructions for build is in fact target platform-specific and application-specific. It requires tools and source files as inputs of the tools specified for a build process, since both of them have influence on the output. The tools, i.e. compiler, linker, etc., can be decided once a hardware platform is chosen, whereas the input source files have to be derived from application models. After each input is listed, a series of instructions may follow, which define how to transform the input into the output.

According to the COMDES meta-model, the makefile is modelled by the attribute *build_ script* of a network node class. However, a fully complete build script cannot be given based only on a network node model. The attribute *build_ script* can contain the building steps (makefile rules) performed by tools (i.e. C compiler, linker, etc.) that use a number of predefined variables to obtain inputs and parameters. This part is given when a valid network node model is created. Listing 6.18 demonstrates a piece of makefile rules (as value of the *build_ script* attribute), where one of the predefined variables, `COMDES_TYPE_OBJECTS`, is used in one of the rules. The application-specific variable is assigned in another makefile (`COMDES.mk`) that is included at the beginning of the rules.

Listing 6.18: **Makefile – platform-specific rules**

```
# import variables
include COMDES.mk

# makefile rules
all: $(OBJS) $(MEMORIES)

$(MEMORIES): $(OBJS)
        $(CC) $(LDFLAGS) -T$(LDSCRIPT)-$@.ld \
-o $(PROJECT)-$@.elf $^ $(COMDES_TYPE_OBJECTS) $(LIBS)

%.o : %.c
        $(CC) $(CPFLAGS) -c -o $@ $<
```

The predefined variables associated with the input files are generated from an input COMDES system model by the so-called *configurator* tool. Listing 6.19 lists four variables in the file `COMDES.mk` that are important for a building process. The values of these variables should be generated from the COMDES model by the configurator.

Listing 6.19: **Makefile – application-specific variables**

```
# the name of a final executable on a platform
COMDES_NODE_ID =

# all source files of FB instances, actors, etc.
COMDES_INS_SOURCES =

# all relocatable objects of FB types from FB repository
COMDES_TYPE_OBJECTS =

# paths of all FB interfaces, platform headers, etc.
COMDES_INCLUDES_DIR =
```

As one of the significant tools during the build steps, the linker is used to merge several relocatable object files into a single loadable module [85]. Its behaviour is controlled by the so-called *linker script* that describes the layout of memory on the target processor and includes instructions on how the linker is to place object code modules in that memory. It specifies how the sections in input relocatable files should be mapped into an output executable file. The GNU linker, ld, accepts Linker Command Language files written in a superset of the AT&T Link Editor Command Language syntax, to provide explicit and total control over the linking process. Its linker scripts are text files, which can be written as a series of commands, i.e. keyword, assignment to a symbol, etc. The contents of a linker script are both platform-specific and application-specific, and thus can be handled in a similar way to the makefile.

6.4 Summary

COMDES advocates the development of embedded control applications using reusable components at both modelling and implementation levels. Chapter 4 has presented the meta-modelling technique and concrete models concerning the former issue. At the implementation level, design patterns are defined so as to retain the reusability of components specified at the modelling level, as addressed in this chapter.

The FB design patterns standardise the code structure for all kinds of component and provide for efficient and reusable implementation of the function blocks. These patterns are actually programming algorithms, and enable models specified within particular kinds of component to be implemented in a reusable way. A FB is composed of two parts: an interface implemented as a data structure comprising necessary execution attributes (such as input pointers, parameters, etc.), as well as a group of reusable functions implementing the dedicated functionality of the corresponding type of FB. The interface of an FB can be instantiated to create a specific FB instance with appropriate execution data.

From the binary point of view, no matter how COMDES FBs are modelled or coded, they are fundamentally no more than relocatable objects, which are compiled from source files for a specific hardware platform, and can be linked to an executable. The relocatable objects are stored in a repository for reuse. When an application is created, the linker should assemble all the objects into a native executable. Finally, an image from the executable should be created and downloaded to the target hardware platform for execution. The generation process from source code to executable can be guided by a number of build scripts.

Tools should be developed such that components and applications can be automatically synthesized from their specifications to a maximum extent, thus relieving the programming effort and desirably minimizing the errors incurred by manual coding. Next, the generation of eventual component code in COMDES can be guided by the component design patterns. Finally, all generated source codes are automatically transformed into executable code by means of the GNU GCC and GNU Binutils.

Chapter 7

From Platform-Independent Model to Platform-Specific Model: the COMDES Development Environment

In order to make this chapter easier to understand, let us start with a small example that people have been familiar with: building a house from scratch.

The first thing to do is look for a licensed professional architect who can plan and design the building. He must understand the rules (i.e. building code) to which the design must conform, so that the requirements of the house (i.e. safety) can be satisfied. He also knows the construction methods available to the builder in constructing the building. According to his knowledge and customer's requirements, he should be able to come up with a house plan which depicts the principal information provided of the house. For example, a floor plan is an overhead view of the completed house indicating rooms, all the doors and windows and any built-in elements, such as plumbing fixtures and cabinets, water heaters and furnaces, etc. Floor plans will include notes to specify finishes, construction methods, or symbols for electrical items. House plans use some lines and symbols to convey the relationship between objects, i.e. a wall should be drawn using thick solid lines. Furthermore, a construction method and materials such as brick, stone, steel, concrete, or others have to be decided too. Next, according to the method and materials, the steps of building the house are also important. Normally, a huge majority of houses are built using completely standardized building practices adopted in America and Europe. The building usually goes through some steps like: grading and site preparation, foundation construction, framing, installation of windows and doors, roofing, and so on. There are many ways to perform

these steps. You could do everything on your own if you had enough knowledge. Alternatively, a popular way is to assign these tasks to subcontractors. For example, the framing is generally done by one subcontractor specializing in framing, while the roofing is done by a completely different subcontractor specializing in roofing. Each subcontractor is an independent business. All of the subcontractors are coordinated by a contractor who oversees the job and is responsible for completing the house on time and on budget.

All of the people involved in house building activities need some kind of tools for assistance, i.e. the architect needs pens, rulers or even a computer to draw the house plan, the subcontractor for framing needs clamps, and the subcontractor for roofing needs slate cutter, etc. Don't forget those who produce building materials like bricks, stones, steels, concrete, etc., also need some tools. Using the right tools makes their job relatively easy. Then a house can be erected up following the sequence of construction steps and with the help of those experts and tools.

Now, let us come back to our story involving embedded software, where the goal is to create an embedded program executed on one or more microprocessors mounted on hardware platforms, for the purpose of measurement and control. Usually, in the world of embedded software, the executable program running inside the microprocessors is in the format of binary code.

Similar to architects, domain experts are able to come up with a solution to a given problem, such as architecture of the system (i.e. closed loop), control algorithms used, etc. In the context of this project, they need to choose the component-based method, i.e. COMDES, to realize the design idea. Therefore, they need to give a specification using the DSL and the concepts (i.e. actor, signal) introduced in Chapters 3, 4 and 5. Function blocks are the materials used as building components. They also need to order function blocks supplied by embedded software engineers who are able to create function blocks for a given hardware platform, based on the patterns introduced in Chapter 6.

Subsequently, an executable program can be constructed from the specifications through a number of steps constituting the development process. The steps introduced in this chapter are categorized as: component development process – for the development of function blocks, and application development process – for the development of executable application programs, in the context of COMDES. Meanwhile, tools can be used throughout the steps in order to automate development, and their functionalities will be presented in this chapter. In the current context of COMDES, the application and function blocks are implemented in the C language; the run-time environment is provided by the HARTEXμ kernel, which is also written in the C language; and the repository is implemented in a file sys-

tem. Therefore, tools that deal with those C programs and file systems will be considered in order to limit the possibilities.

The second part of the chapter will present technologies that can be used to build these tools on the Eclipse platform, so as to fulfil their functionalities for COMDES development. It will not give all details on how to implement each tool in Eclipse. Instead, the discussion mainly focuses on possible technologies and tools that could be employed to accomplish the task of COMDES tool implementation, and most significantly, on the experience gained from the task carried out during the project.

The implementation of the COMDES development environment actually went through two iterations, where a first prototype [86] was built to try out different possible solutions and discover all potential problems, and an improved version was subsequently developed, based on the previous experience as well as the solution to the problems found before. Therefore, we believe that the issues discussed in the following sections contain meaningful points to consider when building similar tools. They are also close to practice due to the fact that most of them are derived from hands-on experiences.

7.1 Overview of the development process and tools

The development of an embedded control system is a difficult task, which involves various kinds of concepts, models, artefacts, etc. The COMDES methodology presents a systematic attempt to decompose this task by separating domain-specific concerns from implementation concerns, and time-related concerns from functionality concerns.

COMDES is a methodology for embedded real-time control software development, whose core is a domain-specific modelling language for control applications. It adopts a model-driven development approach allowing for a solution to a given problem, independent of implementation details. A COMDES model explicitly specifies the exact functional and real-time interaction of software components with the physical world, which is in turn transformed into implementation that ensures the specified behaviour on a given platform. The implementation generation is accomplished using a component-based development method, whereby the final application is composed from prebuilt executable components stored in binary format. Thanks to the completeness of the COMDES meta-model, the COMDES application model contains all the information needed, such that it can be effectively transformed into an implementation on a specific platform.

The functional aspect of a COMDES system model is transformed into func-

tionality code, independent of a run-time environment. The code includes component instance code used to "glue" prebuilt component type code according to a chosen hardware platform.

A COMDES model can express concurrency and timing. However, it does not contain information that requires detailed knowledge of a run-time environment. For instance, an actor model specifies when an input is read, which components are used for computing, and when an output is provided, without specifying a priority for the computation task, because that is related to the task management function of a real-time operating system (RTOS) based on a certain scheduling algorithm. In this way, different operational environments can be used to implement the COMDES timing behaviour. If strict timing were not crucially required, it is even unnecessary to employ an RTOS as a run-time environment: a basic round-robin scheduler invoking all the actors would be enough. However, in this project, a COMDES system model is transformed into an RTOS model – a HARTEXμ model, which is in turn used to generate timing executable code supervising the execution of the functionality code.

After linking, a final executable program is created including COMDES components for functionality and RTOS for timing, task scheduling and interaction on a specific hardware.

From the analysis point of view, information can be extracted from the complete COMDES system model, and transformed into appropriate models for analysis purposes. Thus, tools can be employed on the level of the abstract analysis model to analyse, verify or simulate the corresponding model. However, the analysis of COMDES models as well as transformation to analysis models is not a requirement of this project and is not discussed in this thesis.

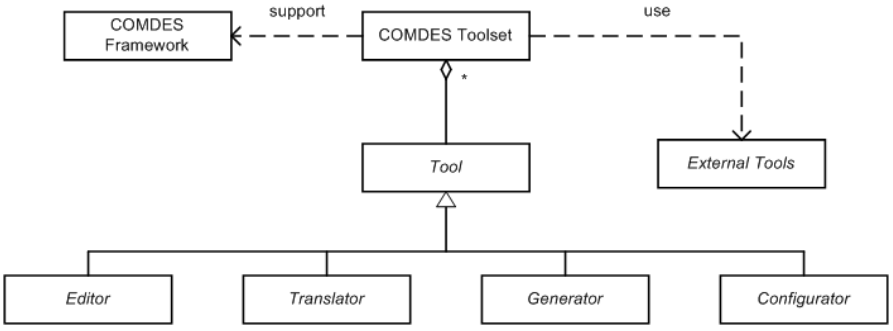


Figure 7.1: Overview of the COMDES Toolset

The envisioned software development process covers the main stages of system development (see Fig. 3.6), i.e. system modelling, system analysis, code gener-

ation and configuration from prefabricated components. It is supported by the COMDES toolset (see Fig. 7.1), which integrates a number of tools (i.e. Editor, Translator, Generator, Configurator, etc.), and eventually uses external tools (i.e. GNU compiler, GNU linker, Simulink, UPPAAL, etc.) in order to accomplish the desired operations, e.g. modelling, model analysis, model transformation, code generation, assembling of executables. The toolset will help to automate the embedded software development process and to improve the quality of the resulting code.

The above discussion just depicts the abstract concepts concerning the steps involved in an application development process, how to transform the corresponding models and what tools can be used in a general way. Conceptually, tools include a modelling tool, model-to-model transformation tool, model-to-text tool, etc. However, as usually, real world life is much more complex than only concepts. On the way from a COMDES system model to a runnable and deployable executable program, in particular – as outlined in Fig. 3.6, a multitude of details should be taken into consideration, the lack any of which may result in a failure when building applications.

In the following sections, the tool support for component design and implementation (including executable code generation) is addressed first. Next, application development support offered by other tools will be discussed, specifically addressing issues concerning system modelling, model transformation, code generation, and configuration.

7.2 Function block development

According to COMDES, function blocks are stored in component repositories in binary format, as required for a specific platform. Therefore, neither manual coding nor code generation of function block types takes place during the application development stage: a system is configured from these prebuilt components and is then able to execute on the corresponding platform.

Before a domain expert starts creating a COMDES system model out of reusable function blocks, the function blocks have to exist in repositories in the format of models and binaries. Therefore, embedded software engineers should create components following a component development process. Equipped with the knowledge of hardware and software, they are given the task to create reusable component models, refine general components to hardware-specific ones, and contribute to the component repositories, following the process outlined in Fig. 7.2.

Firstly, a FB type model needs to be created according to requirements from

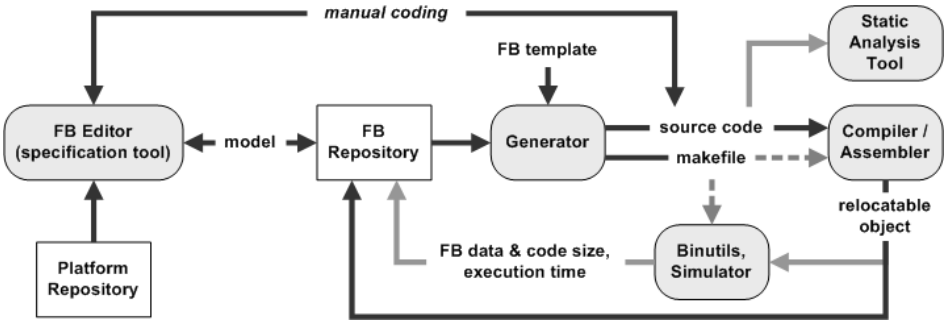


Figure 7.2: Component development tools

domain experts who need the component. Secondly, a code skeleton of the validated FB type model can be generated based on the function block design pattern. In case of basic function block, the skeleton needs to be filled out with real implementation code. Alternatively, in terms of the meta-model of COMDES, the implementation code can also be specified in the FB type model and in turn generated automatically. If necessary, some static analysis tools can be applied to the source code to further validate the component code against possible manual mistakes or programming defects, i.e. a basic function block input pointer could be misused as an output. Thirdly, the source code is compiled into a relocatable object for a given hardware platform that is modelled as a network node type model from the platform repository. As a matter of course, the network node type has to be created prior to this function block model. Binary tools could be used to evaluate the generated binary in order to obtain more platform-dependent information, i.e. GNU Binutils for code size, etc. Lastly, all created artefacts needs to be registered into the function block repository, so that they are available for reuse during the application development process, including FB type model, FB type interface, compiled FB type object and optional FB type source code.

Basically, tools like editors and generators should be created specific to COMDES function blocks. These tools operate based on the COMDES meta-model defined in previous chapters. Furthermore, external tools like GCC Compiler, GNU Binutils, etc. are used to assist the development of prefabricated function blocks.

FB editors are essential to the development process by supporting developers in the creation and modification of FB type models. There should be an editor available for each kind of FB, because each of them has a different construct according to the COMDES meta-model. Additionally, the editors should provide a validation functionality that checks models against constraints defined together

with the meta-model, so that validated FB type models can be provided for downstream tools.

A reusable function block type requires a number of functions defined, and it is inevitable for developers to code the functions of basic and driver function blocks; thus a programming editor (i.e. C editor) integrated with the corresponding FB type editor will offer great help and convenience. However, for state machine, composite and modal kind of FBs, the programming editor is not mandatory, owing to the fact that their functions are implemented by standard routines. These routines should not be modified, as any change could result in changing the semantics of those COMDES components.

FB generators integrate certain templates derived from the FB design patterns and transform FB type models into FB type code based on the meta-model of each function block kind. As a good practice, model validation should not be a major task of the code generators; otherwise the FB templates are polluted with a lot of code for checking input models that has nothing to do with generation. Instead, the code generators should focus on producing FB type source code for both implementation and interface.

The compiler converts FB implementation source code into objects – the real reusable components. The selection of compilers depends on the hardware platforms on which the FB targeted, hence the information such as CPU architecture, function libraries, and compiler parameters, etc. are specified in the corresponding network node type model. Subsequently, the transformation done by compilers can be specified by a script (makefile) generated by FB generators once a platform has been selected. Compilers are usually shipped with the hardware vendors, hence they are not covered by this study.

Likewise, in the embedded system world, target platforms can also be seen as reusable components, and can be represented roughly by the COMDES network type models. Code generation of both components and applications requires information from these models, therefore the creation and modification of validated network node type models should also be supported by the corresponding editor, which is functionally similar to the FB editors.

7.3 Application development

The application development tools support COMDES-based application development, which is outlined in the development process steps described below:

1. From a given control problem, a domain expert creates a COMDES system model by specifying system structure and behaviours. The functional

behaviour can be specified with predefined FB models, whereas the timing behaviour is specified with the attributes of each actor. Actors are allocated onto network nodes that are also derived from predefined models stored in the platform repository. Next, the system model has to be checked against syntax and static semantics. Meanwhile, the model could be verified or simulated against required system properties with adequate tools support.

2. From a validated system model, glue code of FB instances can be generated by a code generator. The generated code is accompanied with information stating the used predefined FB types stored in the FB repository. If timing behaviour is not specified in the model, an executable program can be obtained by compiling the generated code and linking the code with the FB type objects. Otherwise, a run-time environment, i.e. an RTOS is needed.

3. From the system model, a run-time environment (RTE) model and code can be derived, which determines the timing behaviour of the system. The code generated from the model should execute the actors in response to the behaviour of the physical environment.

4. The system model is also used to generate scripts including the whole configuration information. With such information the compiled application glue code, the selected FB types objects, run-time environment code and platform-specific objects or libraries can be linked together to generate an executable program.

5. In case of any change on the system model, i.e. addition, modification or removal of FBs, a reconfiguration process is applied. The new system model should be compared with the previous one in order to find out the difference. From the model difference, a patch that contains only the changed part of the system can be obtained.

The application development is accompanied by the corresponding toolset (see Fig. 7.3) that comprises a number of individual tools as well as repositories holding the exchanged information. All application specific files generated in each of the tools are stored in the application repository. In Fig. 7.3, the dotted arrows represent invocation of the Make utility according to the building script, here called makefile, whereas the grey arrows denotes the reconfiguration process.

System Editor: The system editor is used by domain experts to create a COMDES system model. In principle, it should integrate functions that check the created model against constraints. Furthermore, in order to create a complete

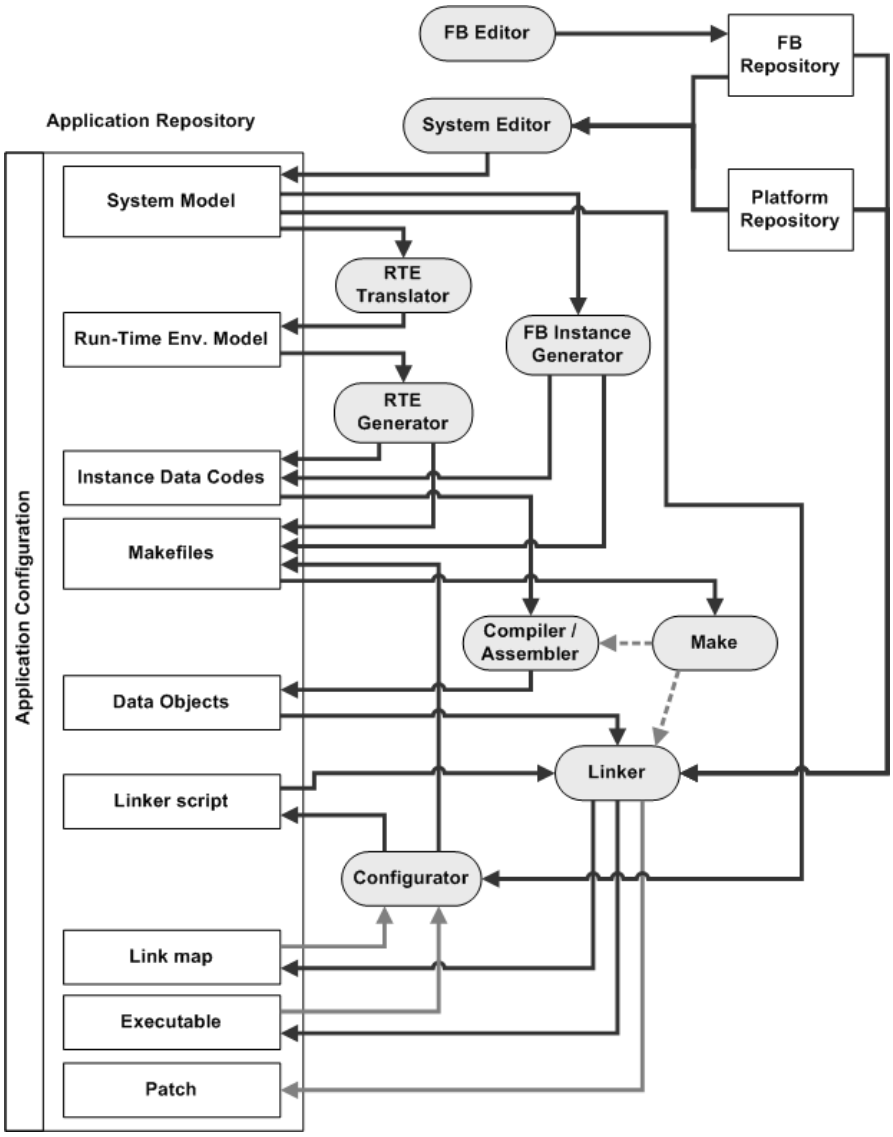


Figure 7.3: Application development tools

system model containing all used function block instances, dependencies among function blocks should be solved as another underlying function of the editor. This function is activated whenever a composite FB or modal FB is instantiated.

Managing component dependencies is a very important and complex issue in component-based applications. Because a COMDES system is statically defined, the system itself has no mechanism to manage its components at run-time, e.g. installation of a new component. Therefore, the dependencies between components have to be solved at design-time. When a developer uses the editor to specify

a COMDES model by selecting function blocks from the repository, there might be some components, such as composite function block instances, that depend on other function blocks stored in the repository. In principle, the developer should focus on whether or not the component can satisfy the application requirements, without concern for the dependency of function blocks. There should be a mechanism to help him with finding all the necessary components from the repository automatically and integrating them into a complete COMDES system model. The complete model comes from the components and their dependency components. Once all dependencies of the application are detected, the model containing all instances and all function block types used in the application can be obtained.

A component dependency graph [87] can be used to model dependencies between components in the system, so as to analyse and manage these dependencies. One way to store dependencies is using a matrix, whereby each component is represented by a column and a row in the matrix [87][88]. It is easy to understand, and existing mathematical methods can be applied to it. A matrix-based approach can be used for COMDES function blocks dependency analysis, taking into account the perspectives of both FB type and instance, where the dependencies of COMDES function block instances are represented as a matrix and the relationship between function block type and function block instance is viewed as a dependency (i.e. any FB instance depends on its type) and also represented by matrix. Therefore, a complete COMDES application can be obtained via a number of mathematical operations on matrices [89]. However, the matrix-based approach can be improved in at least one aspect: a matrix usually contains a number of zeroes (means that there is no dependency between two components), resulting in memory consumption and wasting.

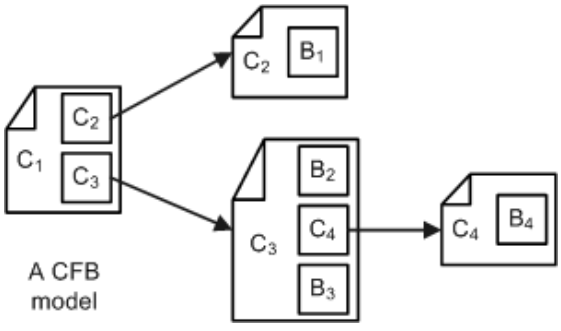


Figure 7.4: A dependency linked list

Another way to represent a component dependency graph is with linked lists. Each component has its list of adjacent (dependency) components. Such a rep-

resentation reduces the resource consumption [87]. This approach has been employed in the current COMDES toolset in order to analyse and manage the dependencies. According to the COMDES meta-model, each composite or modal component model has an implicit description of dependency, specifying which other components (both type and instance) are needed. Solving the dependency is consequently a recursive searching routine. The search starts from the system model, which is the top level of all components, and it will not stop until it reaches basic function blocks that do not depend on any other function block, as illustrated in Fig. 7.4, where the symbol C_X stands for a composite FB, whereas B_X for a basic FB. The linked list can be implemented as a file system where each node representing a component model is a file. The connection among nodes is represented as a path of the file system.

Another thing to consider is that, while components are being added into the application during the dependency resolving stage, they should be named appropriately and uniquely. The names of instances in the application model are specified by the developer. Since he has no knowledge about the instances inside a component instance, those internal instances should be named automatically. Provided that each two components are not allowed to have identical names, the rule of naming could be simply defined as *parent instance name + child instance name*, where the parent instance name is the one specified by the developer and the child instance name is the internal instance name that is shipped with the component. Otherwise a namespace mechanism must be developed.

Some MDSD development tools introduced in Chapter 2 (i.e. Cadena, GME) support model reuse feature. Thus, if an editor is developed based on such kind of tools, dependencies between components can be resolved automatically with the underlying algorithms provided by the tools. However, if we base the system model editor on an MDSD tool, which does not have such a capability (i.e. Eclipse), the *Kind-Type-Instance* pattern introduced in Chapter 4 can be employed as a model of relationships between type and instance, and the dependency is resolved using either a matrix-based or linked list-based approach that can be implemented as an extra function integrated into the editor. This function should be called when a function block type is instantiated in a COMDES system model, and should automatically create all instances that the type depends on.

The output of the editor is a validated COMDES model that contains sufficiently complete information for the next transformation steps, such as model-to-model transformation, model-to-text transformation, etc.

FB Instance Generator: The COMDES model provided by the system editor contains all information necessary for the FB instance generator to output glue

code. The dependency resolution step is significant to the glue code generation step. If dependencies were not resolved properly, glue code cannot be completely generated from a system model, due to the fact that instances required by composite or modal function blocks are not present.

It is important to note that the generated source codes are data structures representing component instances, which define the application by gluing prebuilt components together. Manual coding is not necessary, because the application-specific logic has been defined as component types, which are stored in the repository in the form of binary objects. This is in contrast with the code generator used in the individual component development stage, where the manual coding is done by a skilled programmer.

This generation is concerned with the application functionality aspect without involving any timing aspect, because the run-time environment of FBs can be different and detailed information about the environment is unknown to FBs and actors.

RTE Translator and RTE Generator: A COMDES system model will be transformed into a HARTEXμ model according to the transformation specification described in Chapter 5. The transform requires the availability of both the COMDES meta-model and the HARTEXμ meta-model. The RTE translator should take a COMDES system model as input and should output a complete HARTEXμ model. The generated HARTEXμ model will be consequently translated into timing code containing kernel instance data generated by the RTE generator.

Configurator, Compiler and Linker: The final executable of a COMDES application is said to be configured, because it consists of predefined reusable components in the form of data structures (FB instances) and executable functions (FB types).

The configurator generates configuration scripts from a validated COMDES system model by extracting information about the used FB instances, FB types and platforms. In a COMDES system model, two or more FB instances might share the same type, whereas only one type object is needed to link with instance data code. The configurator searches all types needed by instances in an application, and selects these type objects based on platform from the repository, and puts them into the configuration scripts.

Currently, the COMDES components are implemented in the C language. Therefore, existing compilers and linkers provided by the GNU tool chains can be employed to obtain the final executable. The compiler and linker are chosen according to the platform models that a COMDES model is allocated to. Ac-

cordingly, the configurator generates two scripts used in the application building process: *makefile*, which contains a configuration script specifying what components should be linked, and *linker script*, which specifies where components should be placed in memory.

The linker plays a key role in assembling component binaries together. It combines the instance data objects, the prebuilt objects, as well as a number of standard libraries and maps them to specific memory locations defined in the inker script. The standard libraries are standardized collections of include files and routines used to implement common operations, such as mathematical functions and string manipulation. In addition to the executable, a linker also outputs a *link map* that contains information about the memory allocation of component objects in the executable code.

Another significant characteristic of a COMDES system is its static configuration. This means that the system is entirely defined during the design stage: there is no dynamic memory allocation, the final code is statically linked, which not only helps to avoid any run-time overhead, but also eliminates possible run-time errors like out-of-memory and unresolved references.

Accordingly, COMDES provides a basic offline technique for software reconfiguration and makes it possible to carry out software reconfiguration by deploying only the modified executable code into a system [89][90][29], i.e. by applying a patch. Generation of the reconfigured executable code takes into account the currently running code stored in the embedded device, so the changes are minimal. In a reconfiguration process, the configurator will read the previous link map to identify changes between the current system executable and the previous executable in order to generate scripts to control the linker and the compiler that will eventually generate a patch.

7.4 COMDES development environment on Eclipse

In order to support COMDES application development with a model-driven software development approach, some factors must be considered when selecting a development platform and tools that will be used to create an environment consisting of the toolset. In terms of the adopted process of model-driven development using the COMDES DSL, facilities for meta-model definition, GUI definition, constraint definition, transformation definition and code generator definition are the basic requirements to construct the DSL, as well as the corresponding tools. In addition to the issues concerning DSL development discussed in Chapter 2, the Eclipse provides more features to support rapid tool development during the de-

velopment of the toolset.

The Eclipse is an open-source development platform comprising extensible frameworks, tools and run-times for building, deploying and managing software across its life cycle. At the fundamental level, it is an open platform for software development tools, where one tool on this platform can be easily integrated with others, since the platform provides a large number of services, APIs (Application Programming Interfaces), and frameworks common to different tools. The data needed by the tools can be easily exchanged in the scope of the workspace of the platform, which consequently helps avoid the problem of file importing and exporting between different tools. Moreover, all the tools on this platform can have unified user interface, which makes it possible to build applications using a heterogeneous set of tools while providing a set of common views to the end user.

The Eclipse is in fact a good illustration of a component-based system, whose building component is the so called *plug-in*. A Plug-in is a structured bundle of code and/or data that contribute certain functionality to the entire platform. The Eclipse core is merely an architecture for dynamic discovery, loading, and running of plug-ins. It handles the tasks of finding and running the right plug-in code. Each plug-in can then focus on doing its own task. Furthermore, a plug-in is used together with an extension mechanism, which provides flexibility in how tools are integrated. A tool mapped to a plug-in can add extensions to other tools extension points, so as to support reusability. In this manner, tools on the Eclipse can incrementally be developed and integrated with others.

The Eclipse platform can host the COMDES development toolset to form an entire development environment covering a number of model-driven development activities like modelling, analysis, code generation, building, etc. The tools can be rapidly developed and integrated based on the plug-in model because of the important advantages of the Eclipse platform: reuse of existing tools and tools development support. On one hand, architecting COMDES development tools as plug-ins to the Eclipse allow for the reuse of features from other projects in the Eclipse platform to support the COMDES development environment, which saves a considerable amount of time. On the other hand, from the viewpoint of the developer of the COMDES tools, this platform provides a number of features, such as Java development, plug-in development, debugging, etc., which are instrumental for the implementation of tools. It also offers a variety of capabilities supporting model-driven development and component-based system design. On this open platform, some baseline of tools can be generated automatically from a given model, thus saving time and effort. In addition, there are many facilities in the Eclipse platform that support user interface development. The Eclipse

workbench provides a number of extension points for adding new views, editors, wizards, preference pages, perspectives, etc. that can be reused straightforwardly for user interface development.

The following sections present implementation issues regarding COMDES development tools – from the viewpoint of modelling tools, model-to-model transformation tool, model-to-text transformation tools and integration tools. The modelling tools mainly concern the editors as well as functions behind them, involving meta-modelling, constraint specification, graphical modelling, textual programming and building. The model-to-model transformation tool refers to the tool used to transform a COMDES model into a HARTEXμ model. The code generators and the configurator belong to the category of model-to-text transformation tools, as they transform models to textual artefacts. Finally, the integration of these model-based tools is discussed.

7.4.1 Modelling tools

7.4.1.1 Meta-modelling

When using the MDSD approach, the foundation of building the development environment is provided by a meta-model that describes the possible structure of DSL models by defining the language constructs and their relationship as well as constraints. Meta-models that have been presented in Chapter 4 and 5 describe the whole COMDES framework involving DSL, component, platform, repository and run-time. They are also the basis for building the COMDES development environment, concerning the construction of modelling tools, transformation tools, generation tools as well as integration of tools supporting those steps.

For constructing a meta-model, the Eclipse Modelling Framework (EMF) project provides a meta-modelling language and facilities. In the EMF, a meta-model is described by the Ecore meta-model that is an implementation of the Meta-Object Facility (MOF). A meta-model actually extends the Ecore meta-model by instantiating classes, in the sense that a new class with new attributes in the defined meta-model is created as an instance of an existing one defined in the Ecore meta-model. After a meta-model has been defined, the meta-model is subsequently used by the EMF engine to generate a number of artefacts. These artefacts are Eclipse plug-ins including: the model implementation classes that are closely aligned with the meta-model, the basic editor that allows for modelling and the adapter classes that shield the model implementation code from the editor.

With the support of these generated artefacts, developing the COMDES meta-model becomes easier. The implementation classes allow for manipulating and

maintaining models, based on the meta-model, via programming. These classes are essentially useful for all tools that manipulate COMDES models. The generated editor makes it straightforward to test the meta-model by allowing for model creation, removal and modification, as it is tailored to the meta-model. Therefore, an iterative development process of the meta-model can be excellently supported, since in case of bug found in the meta-model, new model implementation classes and editor can be regenerated quickly. It is not necessary to manually create a testing environment for the meta-model.

However, the generated editor is very limited in functionally and usability, and thus cannot be adopted for COMDES graphical modelling. Other editors need to be created using another Eclipse project based on the generated EMF adapter classes.

7.4.1.2 Constraints

The way of specifying an embedded system in COMDES is to draw a set of domain-specific diagrams. As a consequence, the modelling of the system amounts to drawing a number of shapes and arrows and adding some accompanying text, whose intended meaning is easy to grasp. However, using diagrams has a tendency to be incomplete and imprecise, as a diagram simply cannot express the statements that should be part of a thorough specification.

The Eclipse modelling project provides solutions to specify this kind of information. As one of the solutions, the Object Constraint Language (OCL) [91] is a notational language that can be used to specify model restrictions, and can be used for the definition of constraints for a MOF-based modelling language. It offers a number of benefits over the use of diagrams to specify a system by augmenting the model with OCL expressions, resulting in a complete and precise description of the system obtained. Constraints specified in OCL are added to the meta-model so that a complete meta-model is obtained [92]. Specifically, a class in Ecore can be annotated with a number of constraints written as OCL expressions. These OCL expressions will be evaluated in the meta-model implementation classes, which are generated by the EMF code generation engine. However, practically, the default EMF code generation engine (EMF version 2.3) does not automatically generate code that parses and evaluates OCL expressions, so manual coding is required to add such functionalities into the generated classes.

However, a quite straightforward non-OCL based solution is also available on the Eclipse platform: a set of text-based specifications of constraints and a Java implementation used to check constraints. This approach can be applied to replace the OCL-based approach. A set of functions that take a meta-model class

as input and return either *true* or *false* can perfectly perform the checking task, as long as they carefully implement the specification. These functions should be integrated into the generated meta-model implementation classes. In this way, a complete meta-model implementation in Java can be obtained. However, the approach requires the specification of constraints to be well documented; otherwise, different readers might make different assumptions, which will result in an incorrect implementation.

The two approaches are in fact used together during the building of the COMDES development toolset, based on the consideration that throughout the transformation from a PIM to a PSM, the constraints involved are actually categorized into two levels, namely – platform-independent constraints and platform-specific constraints. The platform-independent constraints have no concern about which target code is going to be generated, whereas the platform-specific ones are bound to the target language, in the case of COMDES – the C language.

When modelling a system, a domain expert focuses on obtaining a correct model without any knowledge of the target code. But if he is going to use a code generator after the modelling step, he needs to know what kind of code will be generated from the model, in order to apply different platform-specific rules. For instance: a constraint like "each model has to be named" can be platform-independent. As long as the constraint is satisfied, the model is considered to be correct, which, however, is not sufficient for the code generation process. Usually, different target code will need different rules for naming variables, if the name of a model is transformed to a variable name in the generated code. For example, in the C language, the variable name must start with a letter or an underscore. Name with a number at the beginning is not allowed. Either lowercase or uppercase letter are allowed as the beginning of the name. If C variables are generated from the models, each model's name must satisfy this additional rule so that the code can be compiled. While in another target language, i.e. Erlang, the rule for variable names is stricter: all variable names must start with an uppercase letter. The constraint for C does not fit here anymore. In short, the rules have to be considered carefully in case a model is used to be transformed to different target languages.

Each of the introduced approaches can be used for either platform-independent or platform-specific constraints. However, the current COMDES toolset implementation uses the OCL-based approach to specify the platform-independent rules (see the constraints in Chapter 4) together with the meta-model. The Java approach is applied to the platform-specific rules while implementing a code generator, because there is not yet a satisfactory OCL-based solution available. All of

these rules need to be satisfied before starting a code generation process in order to produce syntactically correct code. The benefit is that the same valid PIM can be transformed to different PSMs, in the sense that the COMDES framework could be realized in programming languages other than C.

7.4.1.3 Graphical modelling support

To be able to create or understand models properly, modelling must be supported by appropriate user interface that must provide efficient means of expressing the domain concepts of the meta-model. COMDES mainly uses a number of diagrams to specify the embedded software, such as actor diagram, function block diagram, etc. With these diagrams, the meaning of an embedded system becomes obvious once the basic elements of the diagram have been understood.

As a graphical component-based DSL, the editors for COMDES are quite complicated, thus the tooling used to build them should not be overlooked. Basically, the editors have to support a number of features involving multiple kinds of components, hierarchical components, component reusability, different specification diagrams, etc., and that is why the building of all necessary editors involves a considerable amount of work.

Typically, a number of editors are required in order to cover all the levels of specification:

- System level: actor diagram
- Actor level: function block diagram (with latches)
- Function block level: composite FB diagram, modal FB diagram, and state machine FB diagram.

Among them, the editors containing actors, composite FBs or modal FBs are hierarchical in the sense that these components are compositions of other components, which should also be shown in a proper way while modelling. These features must be implemented in the COMDES editors. Most parts of a system model can be created graphically, but there are some elements that cannot (or not necessarily) be created using graphical editors. For instance, in COMDES, a signal contains one or more variables and each variable needs to be specified with a type. The signal is graphically represented as a line with an arrow in the actor diagram. However, from the user point of view, it is not usual to add a number of variables, i.e. symbol strings into a line graphically. In other words, a variable does not need a graphical representation, so adding variables to a signal does not have to be done graphically. The Eclipse provides a property sheet as complementary to

graphical editors (see Fig. 7.5), with basically two functionalities: set or display property of a model element; create model element instances. Then the creation, removal, of modification of variables can be accomplished in the property sheet of a signal that can display all contained variables as a simple list.

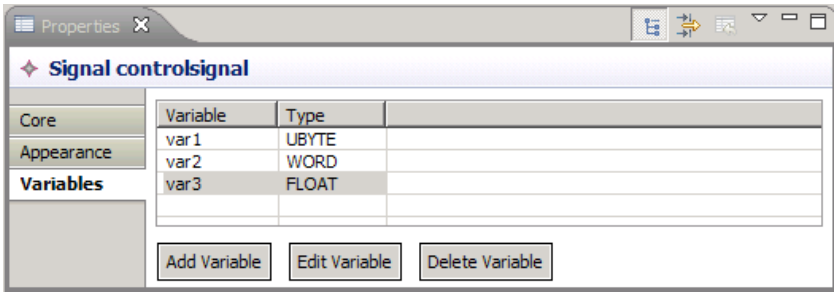


Figure 7.5: The property sheet for a COMDES signal

Another important requirement is that the component repository, as a key part of a component-based software development environment, should be accessible by users during the graphical modelling process. In order to support function block reuse, a function block type stored in the repository must be visible to the users and be instantiatable in a graphical editor. Thus, a component viewer is necessary for the graphical modelling environment. As a basic function, it lists all function block types from which the developers can build instances on an open diagram.

Besides the above mentioned basic requirements, there are additional requirements related to graphical user interface design, which will not be discussed in detail here. For one example, when considering the environment for creating or editing a domain model, only the ability of drawing the models on a diagram is not enough. An interactive environment should be provided to users, in order to make the tool really usable. It needs features like zooming, panning, context menus or buttons accompanying the diagram. These interactive components of the environment are an important aspect of editor design and require some effort as well.

The Eclipse Graphical Modelling Framework (GMF) project provides means to ease and speed up the development of the COMDES editors. It provides a generative component and run-time infrastructure for developing graphical editors, and can be used for the rapid development of standardized Eclipse graphical modelling editors. When using the GMF, a graphical model firstly needs to be defined. It contains information related to the graphical elements that will appear in the editor for modelling (see an example in Fig. 7.6). But this model does not have direct connection to the domain models for which they will provide repre-

sentation and editing. Secondly, a tooling definition model is used to design the palette and buttons that will be used for creation or deletion of the model (see an example in Fig. 7.7). Thirdly, a separate mapping model is used to link the graphical and tooling definitions to the selected domain model (see an example in Fig. 7.8). (All concrete definitions of the COMDES graphical representation models, tooling models and mapping models can be found in [93].)

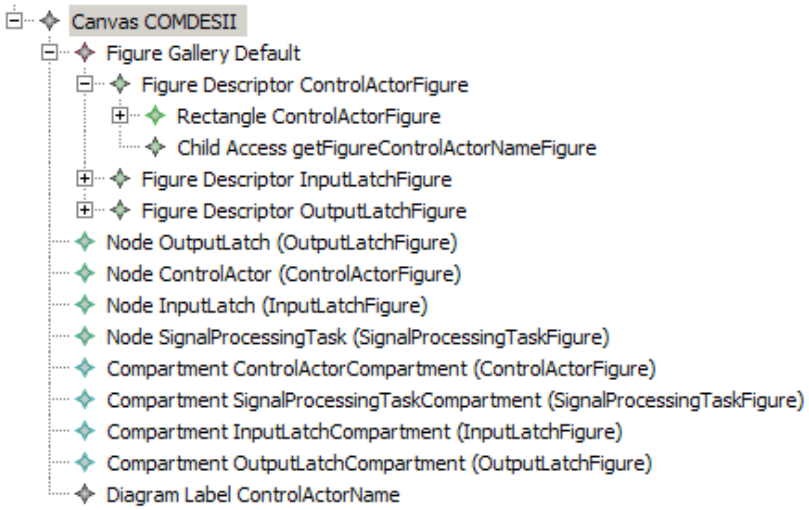


Figure 7.6: An actor graphical representation model

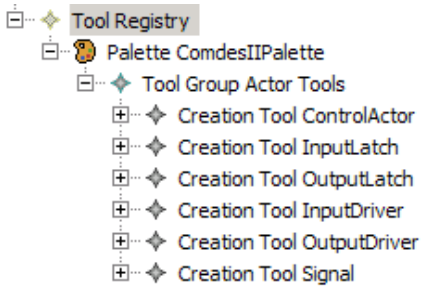


Figure 7.7: A tooling model for the system editor

Once the appropriate mappings are defined, the mapping model will be finally transformed into a generator model where implementation details can be added as needed for an editor plug-in generation. The generated editor depends on the GMF Runtime component to produce an extensible graphical editor. The runtime bridges the notation and domain model (i.e. connects the GMF generated graph and tooling code to the EMF generated meta-model implementation and

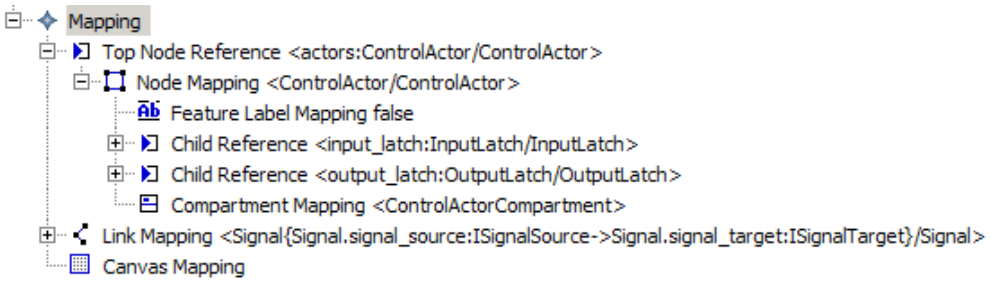


Figure 7.8: A mapping model for the system editor

adapter code), and also provides for features like diagram persistence, context menu, diagram assistants, animated zoom and layout, etc. Furthermore, these features provide a look and feel consistent with other Eclipse-based editors.

With the help of the GMF, a number of COMDES editors supporting hierarchical components and multiple levels of specification are not so hard to create as programming them from scratch, because a baseline of the editors can be generated. However, they have to be customized in order to fit all COMDES features and provide a better usability. Manual coding for the property sheet and the repository view is still required due to the fact that they cannot be modelled and generated. The complexity depends largely on the requirements of the whole user interface.

In brief, from a modelling perspective, a carefully designed combination of modelling editors, property sheets and repository viewer would offer the best solution for modelling a component-based application. Rapid development of these components can be achieved with the support of the Eclipse GMF generation ability.

7.4.1.4 Programming and building environment

An embedded development environment should not only integrate tools for modelling and generation, but also contain tools for application building, so that developers do not need to look for another programming environment to compile and build the application code. Furthermore, COMDES requires a reusable component as a compiled object that is derived from a C code; thus a C editor as well as a C development environment is needed to support the component creation.

On the other hand, from the MDSD point of view, the experiences and observations have shown [94] that many developers who master the conventional code-centric methods do not embrace the value of modelling an application; they still want the visual modelling to be integrated inside their integrated development

environment that is used to perform their daily development tasks. Developers are reluctant to leave their environment and move to a totally new modelling environment without a traditional coding and building functionality.

Therefore, a programming and building environment is useful when the domain-specific models are used to generate code in general-purpose programming languages. The Eclipse platform can satisfy such requirements via the Eclipse C/C++ Development Tool (CDT), which is a popular open source project for C/C++ development. By packaging the CDT as a part, the COMDES development environment will not only include the capabilities of modelling and generation of a COMDES application, but also support developers in viewing, editing, building and debugging the generated code.

7.4.2 Model-to-Model transformation tools

The COMDES framework is meant to give a solution to the software design of real-time embedded control systems; however it is quite complicated to use one single meta-model covering all the aspects of the domain where the system model typically encompasses a variety of aspects, i.e. functional behaviour, timing behaviour, etc. Moreover, making one big meta-model covering everything is conceivable but not practical when it becomes more complicated. To avoid such problems each aspect can be modelled using a DSL suited for the corresponding purpose.

The COMDES design employs "separation of concerns" as a design philosophy, where the timing is considered only at the actor level, whereas the functionality is modelled inside the actor. The HARTEXμ kernel is mainly used to implement the timing aspect of the system, but not for the functional behaviour of the actors. The HARTEXμ model is actually at a lower level of abstraction than COMDES, and can be considered as a separate DSL for modelling real-time systems. It plays the role of a platform for the COMDES model by focusing on when and how to run actor tasks and drivers using a set of kernel facilities such as timers, events, etc., without offering any modelling means to describe what a task does. The latter is modelled as a function block diagram involving a number of function block instances. Therefore transformation between the two kinds of models is required.

Theoretically, model transformation plays a critical role in bridging abstractions, such as PIM and PSM in the MDSD approach. Figure 7.9 gives an overview of the concepts involved in model-to-model transformation. It shows a scenario with the source model used as an input, and the target model produced as a transformation result. These two models conform to the source meta-model and the target meta-model respectively. The execution of the transformation program results in automatic creation of the target model from the source model.

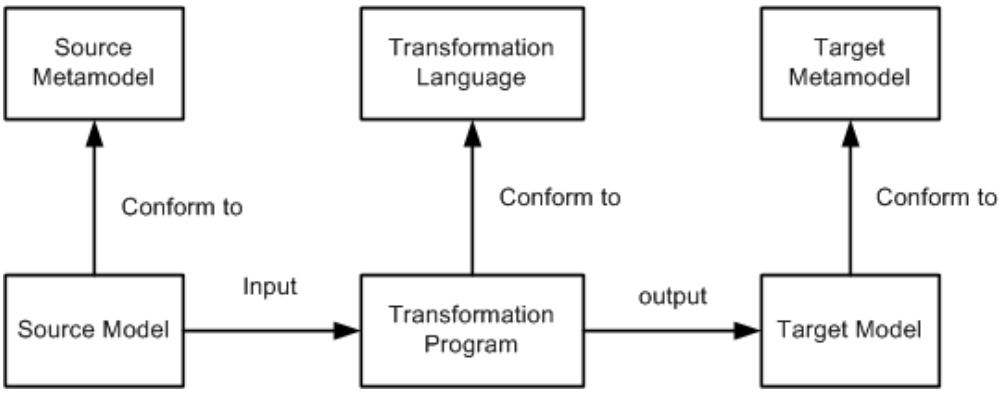


Figure 7.9: Model transformation

The transformation program, written with respect to the source and target meta-models, is executed by a transformation engine. A transformation program should implement a transformation specification, which defines the mapping between the source model and the target model. A specification contains several transformation rules. Each rule implements a small transformation step when creating the target model from the source model. The rules can be expressed either formally or informally. (An example of a transformation specification from a COMDES model to a HARTEXμ model is presented in Chapter 5.)

As a key aspect of model-driven development, the model-to-model transformation languages are provided in the Eclipse M2M project. ATL (Atlas Transformation Language) [95] is one of the components of the modelling framework. It is a model transformation language and toolkit developed by the ATLAS Group. ATL offers a language capable of expressing queries and transformations over models in the context of the MOF meta-modelling architecture. ATL provides its own meta-models defining the abstract syntaxes conforming to the MOF 2.0 meta-model. OCL is used for querying models when writing a transformation program. An execution engine and development tools are available on the Eclipse platform.

Alternatively, a general-purpose programming language like Java is sufficient to write a model-to-model transformation program, too (Fig. 7.10). A transformation tool written in Java usually depends on the implementation classes of the input and output meta-models. The meta-model implementation classes can be used to manage the input and output models. In this case, the tool has to be designed carefully, so as to manage the transformation trace.

Model-to-model transformation is still a hot topic of research. There is no or little prior experience accumulated in the first place. However, both academia and industry are investing effort into this area, and as a result, there are a number of

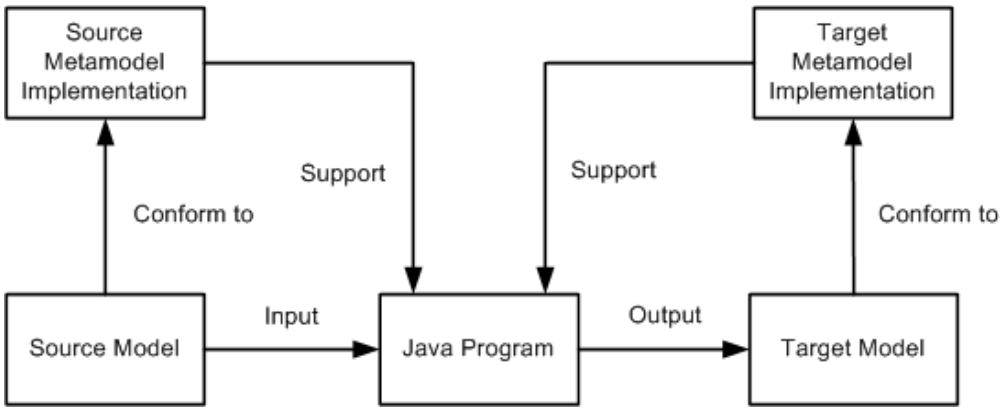


Figure 7.10: Java based transformation

approaches and tools for choice. The work [96] lists more than ten approaches with tools concerning model-to-model transformation. Trying them one by one will take a considerable amount of time. But the following paragraphs present issues that have to be considered before choosing an existing approach (i.e. ATL) or writing one on our own (i.e. Java), to perform the model transformation, in order to efficiently implement the tools of the COMDES development environment.

Firstly, in a transformation program, each rule usually addresses only one small aspect of the entire transformation. However, sometimes, a rule needs to reference certain models created by other rules. Therefore, it is necessary that the transformation engine has the ability to record the run-time footprint of a transformation execution, i.e. a transformation trace. Traceability links can be established by recoding the transformation rule and the source elements that were involved in creating a target element. The ATL provides dedicated support for tracing by creating and managing traceability links automatically. However, the Java approach requires developers to manually record information like which target model instances are mapped from which source model instances when executing a transformation rule.

Secondly, it is important for a transformation language to provide a rich set of functions or APIs for operating with the input and output models. At least, a language should provide sufficient means to navigate the source model, even though sometimes there is no explicit association between elements in the model. For example, in some cases, there is not a direct "child to parent" association defined between a child class and a parent class in a meta-model. However, special support from the language is needed in order to obtain the parent from the child, e.g. a reflective operation "`refImmediateComposite()`" can be used to return the

immediate container of the child in ATL; whereas a function call "`eContainer()`" does the same when using EMF. It may not be possible to implement some of the transformations without the support of such kind of functions.

Thirdly, in general, a complicated transformation may involve multiple source models or multiple target models, i.e. a source model could be stored in several resources, or a source model could to be linked to a number of other models. A M2M solution should be able to process all the input models regardless of their physical locations. This is especially significant for a component-based framework where typically a number of components used in an application are physically located at different repositories. The ATL is capable to deal with multiple inputs and outputs. With the help of the meta-model implementation classes generated by EMF, it is also easy to deal with such requirement in a Java-based solution.

Next, in certain scenarios, it might not be possible to construct a target model in one operation. Likewise, if there is neither a rich collection of APIs available in the transformation language, nor the capability to deal with multiple inputs and outputs, it may be necessary to transform the source model in multiple steps. In this way, a target model is constructed incrementally. Then, a number of intermediate models between the source and the target models are needed, and meta-models of these intermediate models are also required.

Finally, always bear in mind that learning a new language and tools takes time. Model transformation is a relatively young area. Although well-established standards for creating meta-models exist, there is currently no mature standard for specifying transformations. The QVT (Query/View/Transformation) standard of MDA might be a starting point. The situation is worsened by the fact that different groups are trying to develop such techniques having different views of the subject area, as a result of lack of standards. Moreover, available approaches are changing over time as they are matured. Therefore, a transformation implemented in a general-purpose language could be a better solution for the COMDES development environment, at the moment.

7.4.3 Model-to-Text transformation tools

Code generation, in the context of MDSD, is also referred to as *model-to-text transformation*, as it is a special case of model transformation. Such a name emphasizes models as the inputs of a code generation process, whereas the term "code generation" does not reveal the source of the generated code. Like code generation, the target of model-to-text transformation is just text or string. The string can be any textual artefacts, involving either code or non-code ones. Conversely, a target of model-to-model transformation is an instance of the target meta-model.

There are satisfactory solutions for code generation (not necessarily for model-to-text transformation), as well as tools available on the market. The majority of current available tools support template-based generation, where a language is specially designed to write the templates and is referred to as a *template language*. A template language usually provides basic elements for writing the generation program, such as condition, iteration, and expression. Practically, using templates helps to reduce the complexity and increase the readability of the code generation program; otherwise the program can quickly become very complex and hard to understand.

The templates that realize the transformation from platform-independent model to platform-specific model are usually derived from a reference implementation, in order to make sure that the generated source code would be compiled. As code generation is a key step when applying the MDSD approach, an existing implementation is useful as a reference for getting knowledge about what the generated code looks like and what kind of generators are needed to support this step. The Production Cell case study that was developed in the COMDES framework [97] has been used as a reference implementation during the development of the generators and the configurator within the COMDES development environment.

A template usually consists of dynamic code, which is executed at run-time to iterate over the model instances, fetch information from the source model and do computation, and static text, which is output directly to the generated target. The static code does not depend on the model and normally, it has just to be copied into the templates obtained from the reference implementation.

Some template languages are hybrid in that parts of the generation program are implemented in the specific template language, and other parts – in a general-purpose programming language. This feature is extremely useful when the template language is not sufficient to perform complicated computational tasks. The developer can use the not-so-expressive template language to navigate the model, get information from the model and then pass it to a general-purpose programming language to make complicated computation.

An advantage using the template-based approach is that during the development stage of a generator, the correction of errors found in the generated code is much easier and can be carried out more efficiently. Provided that the target code written for the reference implementation is correct, a bug in the generated code needs only to be fixed in the meta-model, the transformation rules or the generation templates. Once bugs are fixed in these places, all flawed code fragments are replaced with corrected ones after regeneration.

There are some choices for a model-to-text transformation solution from either

an Eclipse-based project (i.e. JET, Acceleo) or a non-Eclipse-based project (i.e. CodeWorker), which have been tried out during the Ph.D. project. All of the mentioned tools are template-based and consequently, can be used to develop the tools for generative purposes, i.e. FB code generation and configuration scripts generation, etc. CodeWorker is a free tool, which has been used for generation purposes in several projects executed in the MCI. As a component of the Eclipse M2T project, JET has inherent superiority over the other tools. Acceleo is an implementation of the model-centred approach, which is also able to operate on the Eclipse platform.

In addition to the above mentioned template-based tools, other similar code generation tools could be used to develop generative tools for COMDES as well. However, something worth noticing is that in the context of the COMDES framework, the code generation is very complicated and time consuming, although theoretically it looks very simple. The generation involves several kinds of components, therefore requires dozens of templates developed for all components. The computation, such as the generation of the sequence table for a composite FB, or the binary decision diagram table for a state machine FB, etc., is not trivial. Also, the generation of configuration scripts involves access to a number of component type models through the component repository model, which increases the complexity of the generation program.

Therefore, selecting the right tool to develop the COMDES code generators and the configurator would help save a considerable amount of time. Practically, the tool should provide an easy way to retrieve useful information from the source models. Textually parsing a source model based on its concrete syntax is not a good idea in the context of model-driven development. Furthermore, the tool should provide comprehensive functions to facilitate complicated computations. Otherwise, it should have the ability to integrate a general-purpose language into its template language, such that it would be possible to pass models between the general-purpose language and the template language.

In fact, the JET and the CodeWorker tools are devoted to the generative programming approach, which is not a strong model-centred approach, since it does not require a meta-model as the baseline. An XML file can be used as input for these tools and does not have to be the serialization of a model. However, even though the file could be the representation of any model, the model itself cannot be seen by the tools, which increasingly complicates the parsing part of a generator. For instance, a model could be physically stored in multiple XML files with cross-reference, which usually occurs in a system model using a number of reusable component types. The tools can only read the reference as a string,

and find the location of the referenced model or file after some operations on the strings.

As a model-centred solution Acceleo is more appropriate for COMDES, because the COMDES meta-model in Ecore is fully supported by the Acceleo. Navigation and gathering information from different places of the model is quite convenient, with the support of the meta-model. Though the model is exported as an XML file, a parser of the file in not necessary because the Acceleo can parse any model that conforms to Ecore. In this way, writing a template is totally based on the consideration of a meta-model and a model, e.g. the physical location of the model is not important, it could be in one file, in multiple files, or even could be created during run-time. This will reduce a lot of implementation time. Furthermore, the Acceleo can call services within the template language. Services are public methods defined in Java classes providing complex operations, without which it would be very complicated to realize such operations using only the limited number of template syntax elements. Integration with the Java language makes the template more powerful, as a number of predefined Java libraries, APIs of EMF and meta-model implementation classes, etc., are available for operating on source models. The services can take any model or a collection of models as parameters and return any kind of model object or a collection of model objects after certain processing. Moreover, the template language is able to process the models returned by the services, and print them with the static text. Therefore, it allows for seamless Java services and template language integration, which speeds up the development of a code generator.

The Listing 7.1 demonstrates a piece of template program used for generating all basic type FB interfaces, based on the design pattern listed in Listing 6.1, Chapter 6. The template is a part of the COMDES FB code generator. The template is written using the Acceleo template language, and it can be seen that the access to models is based on the COMDES basic FB meta-model (Fig. 4.7 and Fig. 4.9, Chapter 4), which makes the parsing of a model simple. Additionally a service `getTypeName()` is called to compute a valid name from a model, which keeps the template clean.

Listing 7.1: A basic FB interface generation template

```
<%script type="BasicType" name="basic_type_h"
  file="<%bfbtypename%>.h"%>
<%if (output_signals.nSize() > 0){%>
typedef struct {
    <%for (output_signals) {%>
    <%type%> <%name%>;
    <%}%>
```

```
} T<%self.getTypeName()%>Output;
<%}else{%>
//this fb has no outputs
<%}%>

typedef struct {
<%-- input --%>
<%if (input_signals.nSize() > 0){%>
   struct {
        <%for (input_signals) {%>
        <%type%>* <%name%>;
    <%}%>
   } input;
<%}else{%>
   //this fb has no inputs
<%}%>
<%-- parameter --%>
<%if (parameters.nSize() > 0){%>
   struct {
    <%for (parameters) {%>
        <%type%>* <%name%>;
    <%}%>
   } parameter;
<%}else{%>
   //this fb has no parameter
<%}%>
<%-- output --%>
<%if (output_signals.nSize() > 0){%>
   T<%self.getTypeName()%>Output *output;
<%}else{%>
   //this fb has no outputs
<%}%>
} T<%self.getTypeName()%>;
```

7.4.4 Tool integration

After each individual tool in the COMDES development environment has been identified and developed, another challenge would be to coordinate their operation, so as to make the tools work properly together as an integrated toolset. The tools in consideration include not only those used for modelling, model transformation and code generation that are specific for COMDES development, but also – external tools that are instrumental for the development process, i.e. analysis tools, compilers, linkers, etc.

A development environment can be divided into three parts: tools that do

the computational tasks; a coordinator that controls tool interaction; and data or models exchanged among tools. All the tools of the environment interact with other tools under certain control. The interaction could be coordinated by means of specific middleware, which carries out the data sharing and controls the execution of processes in the environment. This kind of middleware usually provides a clear separation between the computational part and the coordination part of an environment [98]. The coordination part is about the way in which tools interact (using e.g. procedure calls, remote method invocation, middleware functions, and others), while the computation part is related to tools that carry out specialized tasks. The separation of coordination and computation leads to flexible and reusable tools and environment.

Considering the MDSD approach where models are the essential elements that all tools operate on, the aim of a tool integration solution is to achieve the collaborative work between heterogeneous tools based on a variety of models. In such an environment, one source tool can send its data in the form of model to one or more other destination tools for specific service or execution. So, a proper solution for tool integration should manage tool interoperability on top of models.

Out of several aspects of an integration solution mentioned in [99], there are two that should be considered at least for the COMDES development environment: data integration and process integration. Data integration is related to enable heterogeneous tools manipulating common data, and ensures that all the information in the environment is managed as a consistent way. The data passed among tools should have a structure that could be understood by all tools in the environment. Exchange of arbitrary data is not allowed. In the COMDES development environment, each tool should operate on models, regardless of what format the data is stored in. Exchanged models could be based on different meta-models. In case the source model is not understandable for a successor, transformation facilities are used to transform the data to make it compatible with the requesting application. The shared models must be saved in the repositories.

Process integration is about the coordination among tools in the environment. It ensures that tools interact effectively in support of a defined process [99]. Different development tools are invoked such that the defined sequence of execution is automatically enacted within the integrated development environment; thus, the manual work of calling each individual tool in the development process is eliminated or reduced.

Nowadays, tool integration has become a very important issue for software development. In order to solve this problem, a number of frameworks (not necessarily specific to Eclipse) have been developed, such as ToolBus [100][98] in

Meta-Environment, ModelBus [53] in Eclipse, OTIF [101] in Model-Integrated Computing Toolsuite, etc.

ToolBus is developed by Centrum voor Wiskunde en Informatica (CWI), and has been applied in a language development framework called Meta-Environment. The goal of the ToolBus is to integrate tools written in different languages running on different machines, which is achieved by means of a programmable software bus. The ToolBus uses data representation based on term structures, and does not allow the exchange of arbitrary data. It coordinates the cooperation of a number of tools. This cooperation is described by a script that runs inside the ToolBus. The result is a set of concurrent processes inside the ToolBus that can communicate with each other and with the tools. Tools can be written in any language and can run on different machines. ToolBus forbids direct inter-tool communication. Instead, all interactions are controlled by the script that formalizes all the desired interactions among tools. Therefore, each individual tool can be replaced by another one, provided that it implements that same protocol, as expected by other tools. Thus, complete control over tool communication can be achieved. Each tool in this architecture needs to be encapsulated in a small layer of software that acts as an "adapter" between the tool's internal data formats and conventions and those of the ToolBus.

Open Tool Integration Framework (OTIF) developed by Vanderbilt University is a framework for constructing integrated tool chains. It can be used in an environment where each design tool has its own format for storing models. Models exchanged between tools can be translated from the format of one tool to the format of another tool, which makes possible to integrate external tools in an environment. Tools can be distributed across multiple machines. In the framework, an intermediate canonical model is used as a bridge between source model and target model, so that the syntactical and semantic transformations are decoupled, and moreover, the transformation process can be isolated from the details of the model representations. In the framework, workflow models are used to define tool invocation sequences. Also, a backplane incorporates a workflow engine playing the role of coordinator that enacts the workflow model and routes the messages between other components in the framework. Additionally, tool adaptors are used to couple tools to the framework by submitting/receiving the data to/from the backplane and converting the data between the canonical form and the tool-specific format.

ModelBus is a model-driven tool integration framework, which allows for building a seamlessly integrated tool environment for a system engineering process. It is dedicated to the realization of a platform offering the integration facilities needed

for applying a model-driven development approach. It provides the ability to integrate modelling tools, languages and methodologies to create fully customizable model-driven development environments. ModelBus provides a standard approach to exchange models and to execute remote modelling services. It supports a defined software engineering process that involves several tools operating on models. It enables transparent interaction between tools, and allows end users to easily assemble heterogeneous tools that interoperate without having any direct knowledge of other tools. To that end, tools use so-called adapters. The adapter is a sub-component defined in ModelBus. To communicate with other tools, a tool has to interact with its corresponding adapter plugged in the bus. When using the ModelBus, a modelling service description should be provided. The description conforms to the ModelBus meta-model [102] that contains all relevant concepts for describing modelling service interface, modelling service, model-related events and model types whose instances are defined in the description. Tool adapters are generated through the Adapter Creation Tool, based on the modelling service description. An adapter connects a tool with the ModelBus. It can register a tool to the ModelBus, lookup for available tools and respond to service invocations. Once a tool has been successfully plugged in, its functionality becomes immediately available to others as a service.

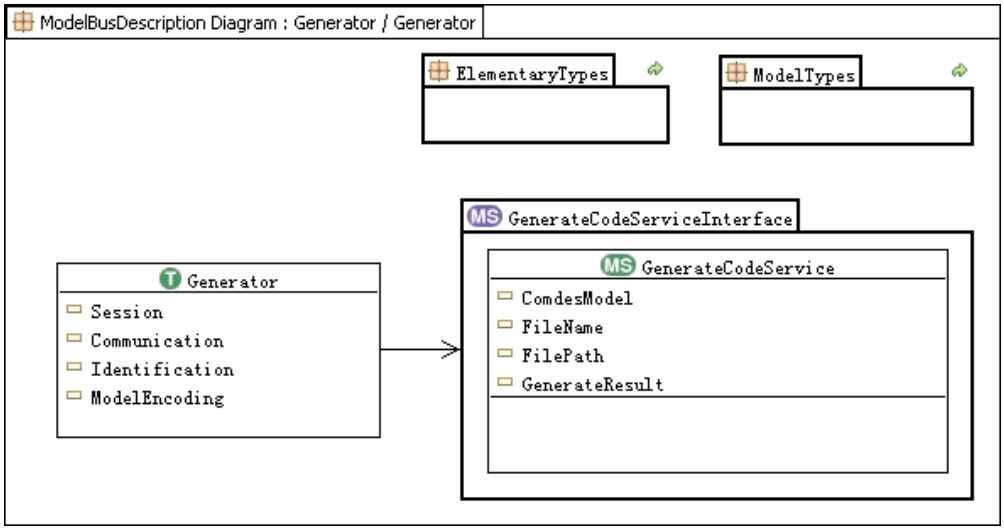


Figure 7.11: COMDES generator tool description

The above mentioned technologies deal mainly with process integration. These architectures can provide a clear separation between the computational part and the coordination part of an environment. The capability of integrating external

tools into an environment is achieved through adapters. A tool adapter interface can normally be generated automatically, given a description of tools and their interactions. OTIF and ModelBus are model-based solutions, so the model-driven development tools can exchange models through the architecture, instead of raw data.

The ModelBus is an Eclipse-based project and its meta-model is defined using the EMF project, which gives a good reason to be adopted in the COMDES development environment, since COMDES employs the EMF as a meta-modelling language. Based on the COMDES meta-model, [103] presents a ModelBus-based solution to execute all the tools in accordance with the process defined in the COMDES development environment. The description model for each of its tools has been defined (see Fig. 7.11. for an example for code generator). However, as a relatively new technology, the available ModelBus tool is not yet stable and mature enough at the moment, as of this writing. Further progress has to be made, in order to incorporate the ModelBus in the COMDES development environment.

7.5 Summary

The fundamental challenge for software technologies of the future is to provide an integrated development environment support for achieving software development automation, using an appropriate collection of tools.

This chapter presents the architecture of the COMDES development environment consisting of tools that have been specifically designed to automate embedded control system development in a model-driven fashion. The toolset supports a software development process featuring prefabricated executable components that are used to configure the executables of the target embedded system, in accordance with design models specifying its structure and behaviour.

The chapter is continued by investigating related issues and technologies used to build the environment on the Eclipse platform, concerning meta-modelling and modelling, model-to-model transformation, model-to-text transformation and tool integration. There is a broad spectrum of solutions that offer a variety of capabilities on the Eclipse. Therefore, choosing a proper one will considerably facilitate the real implementation when taking models into account.

This chapter does not contain sophisticated theoretical knowledge, but it attempts to cover the important aspects of the methodology and technology needed to implement the COMDES development environment, with respect to both model-driven and component-based development of embedded software. The technological issues discussed come largely from the experience gained during the

execution of the Ph.D. project, while implementing the COMDES development environment.

Although the discussion in this chapter is mostly related to the Eclipse platform, the issues investigated and problems discovered will hopefully offer hints to other research and development efforts dealing with software development environments based on models and components. On the other hand, some requirements concerning tool development, arising from the COMDES framework, will hopefully provoke further research and improvement of tools for model-driven development of embedded software.

Chapter 8

Demonstrations

The COMDES modelling techniques presented in Chapters 3 and 4 have been experimentally investigated using the well-known Production Cell Case Study [104]. This chapter will briefly present the software design of the Production Cell control system, based on the COMDES framework and its components, and will then discuss tool support provided by the developed toolset.

The Production Cell control system was designed and implemented in 2006 at an earlier stage of the Ph.D. project. At that time, the design was implemented mostly manually due to the lack of tool support: models and diagrams had to be drawn by hand; function blocks source code and configuration scripts were written manually. Particularly, during the coding, one had to be very careful to make sure that the code conformed to the design drawn on papers. There was tool support only for the HARTEXμ kernel, where a kernel instance could be configured in a GME-based tool used also to generate the code of kernel objects. However, automatic transformation from COMDES to HARTEXμ did not exist.

Anyway, the case study has been successfully developed using COMDES modelling and implementation techniques [97]. Throughout the Ph.D. project, it has been employed as a reference implementation when applying the MDSD approach. The reference implementation is very important when it comes to the creation of the COMDES meta-model as well as tools. It also serves a more significant purpose: it demonstrates the application and realization of the COMDES DSL. Although the reference implementation has been created manually, it exemplifies the transition from model to implementation on the respective platform. Later on, the transformations performed by tools are derived from it. For instance, a number of templates required by model-to-text transformation show great similarity to the implemented code, and can thus be extracted easily from the reference implementation.

The second part of the chapter will demonstrate how to use the COMDES

toolset to support the development of the case study, where essential features supporting component-based and model-driven software development, such as component reuse, system modelling, transformation and code generation, will be illustrated.

The last section of this chapter will present industrial experience related to a case study developed in the Zone Controller Development Division of Motorola A/S. It is about a modified version of the state machine modelling and code generation tool developed in this project, which has been refined and extended with a new code generator in accordance with industrial requirements.

8.1 Production cell case study in COMDES

8.1.1 Introduction to the case study

The Production Cell case study is a realistic industrial application, which aims to show the usefulness of formal methods for critical software systems and to prove their applicability to real-world examples. The problem addressed in the case study belongs to the area of safety-critical systems, as a number of properties must be enforced by the control software in order to avoid injury to people and damage of machines. It is a reactive system, as the control software has to react permanently to changes of the environment. A reduced version of the Production Cell plant has been adopted for this project (see Fig. 8.1 and Fig. 8.2). It has been implemented as an animated computer model controlled by a distributed control system whose design and implementation are presented in the following sections.

The simplified Production Cell consists of five machines: a feed belt, an elevating rotary table, a robot with two orthogonal arms, a press, and a deposit belt. All of these machines work jointly to process metal bricks, which are conveyed to a press by the feed belt.

The feed belt transports metal bricks to the elevating rotary table. An electric motor drives the feed belt to move or stop. There is a photoelectric sensor installed at the end of the belt, which is used to indicate if a brick has entered or left the final part of the belt.

The elevating rotary table passes the bricks from the feed belt to the arm1 of the robot. It rotates about 45 degrees and lifts to a level where the arm1 is able to pick up the brick, since the robot arm1 is located at a different level than the feed belt. There are two sensors installed on the table. The first one measures the vertical position of the rotary table, and another one measures how far the table

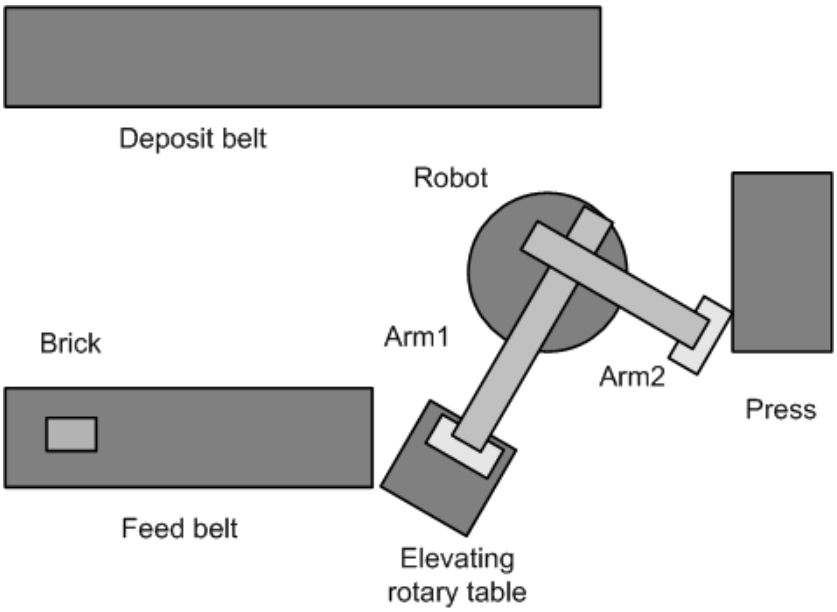


Figure 8.1: Top view of the production cell model

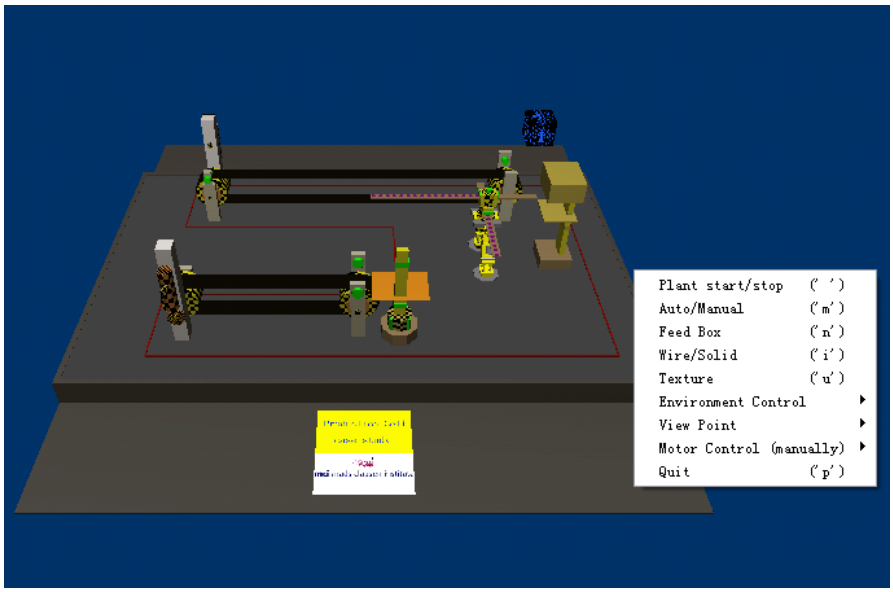


Figure 8.2: 3D view of the production cell model

has rotated. Table rotation and motion are effected by two electric motors.

The robot comprises two orthogonal arms and they are set at two different levels. Each arm can retract or extend horizontally so that it can reach the table,

press and the deposit belt. Both arms rotate jointly. To grip the bricks, each arm has an electromagnet at the end. The arm1 is responsible for taking bricks from the elevating rotary table to the press, while the arm2 is used for transporting forged bricks from the press to the deposit belt. By default, the arm2 points towards the press and the arm1 is positioned between the table and the press.

There is one sensor on the robot to measure how far the robot has rotated, and an electric motor to rotate the robot. Each arm has a sensor to indicate how long the arm has been extended, a motor to extend and retract the arm and an electromagnet to pick up and drop a brick.

The task of the press is to forge metal bricks. A plate is movable along a vertical axis. Because the robot arms are placed on different horizontal planes, the press plate has three positions. In the lower position, the press is unloaded by arm2, while in the middle position it is loaded by arm1. The brick is processed in the upper position. A sensor is used to measure the vertical position of the press plate, and an electric motor can move the press plate up and down.

The deposit belt transports the bricks unloaded by the robot arm2 out of the production cell. The belt is powered by an electric motor, which can be started up or stopped by the control program. In this simplified version, there is no sensor at the beginning and at the end of the deposit belt to indicate the coming and leaving of the bricks.

Two types of property – safety and liveness properties are considered in this system. The safety requirements are most important: if a safety requirement is violated, this might result in damage of machines, or, even worse, injury of people. A very strong liveness property for this system is satisfied, if the following requirement is fulfilled: Every brick introduced into the system via the feed belt will have been forged and will eventually be deposited out by the deposit belt.

Flexibility is another requirement taken into consideration; namely, the control software has to be open and flexible. The effort for changing the control software and proving its correctness must be as small as possible, when the control system requirements or cell configuration are changed. Obviously, a component-based design will satisfy this requirement, as shown in the next section.

8.1.2 System design specification

According to the COMDES framework, the control system specification is developed in a top-down fashion. The top level is defined in terms of actors and their interaction with each other, as well as with their environment. The structural view of the control system provides static information about the interactions between the constituent actors, as well between the actors and the environment. This view

is described by an Actor Diagram, such as the one shown in Fig. 8.3.

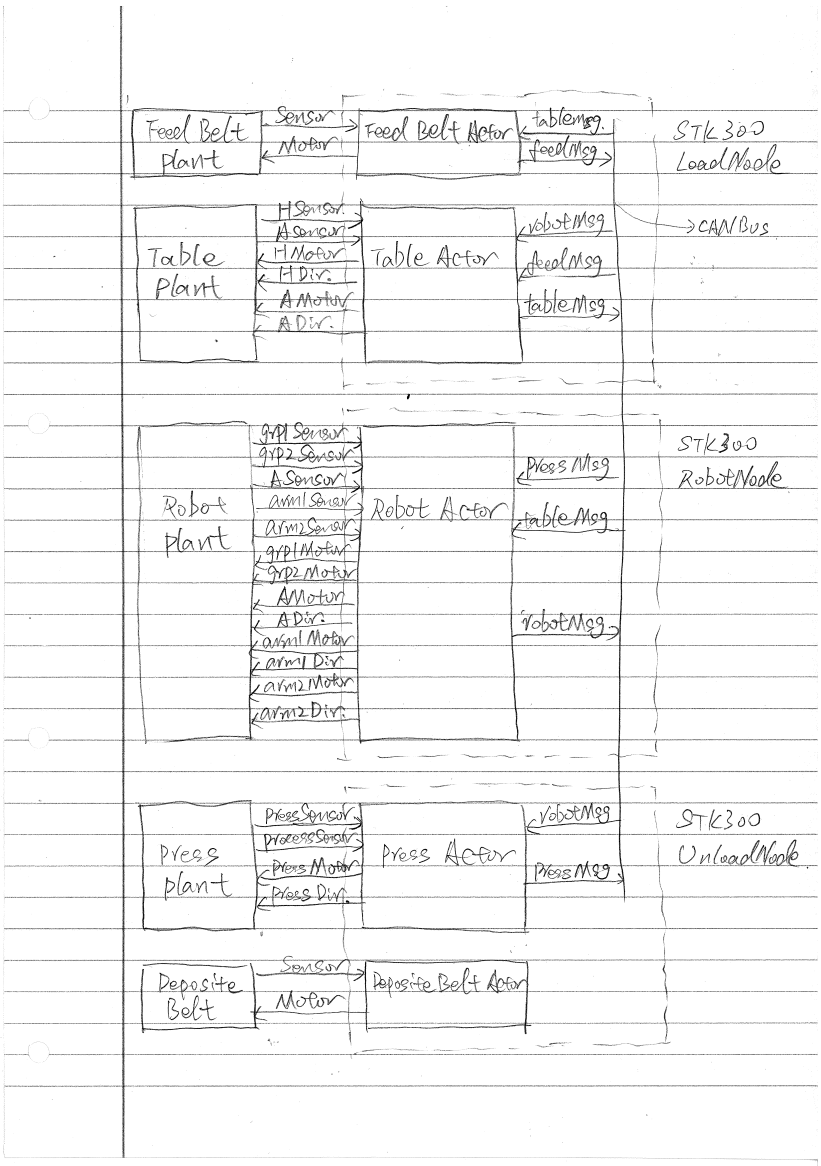


Figure 8.3: Actor diagram of control system

The Production Cell control system consists of five actors: Feed Belt Actor, Table Actor, Robot Actor, Press Actor and Deposit Belt Actor. An actor is assigned to a specific object of control in the physical environment. The internal structure of an actor is specified with a function block diagram involving constituent function blocks and signal drivers, e.g. the Table actor shown in Fig. 8.4.

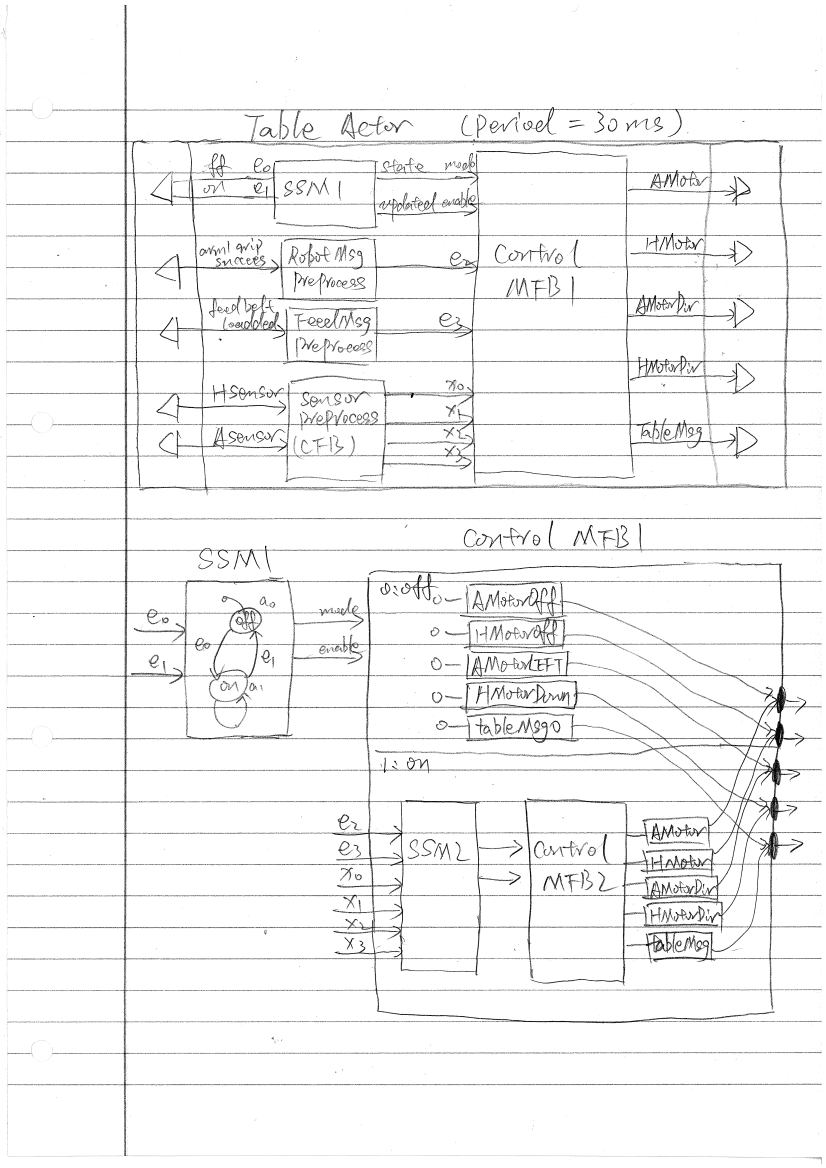


Figure 8.4: Table actor and its internal function blocks

The input signal drivers acquire the corresponding input signals from either plant or communication bus. Communicated signals (messages) are received by signal drivers and decomposed into local variables, whereas sensor signals are read by the physical input drivers. Local signals are further processed by preprocessing function blocks that are either basic or composite function blocks. Other function blocks are used to generate control signals. Output signal drivers compose signal messages and broadcast them to other actors, whereas physical output drivers

generate control signals for the plant actuators (see Fig. 8.4).

The core control part of the Table actor is a hierarchical control unit. It is composed from two controllers: top-level supervisory state machine (SSM) and modal function block (MFB), and second-level SSM and MFB. The top-level state machine has two states, which respond to the manual switch that puts the controller into either ***On*** state or ***Off*** state. It is executed when the actor is triggered by a periodic timing event. However, a transition will take place only when an on/off event is present.

The top-level MFB has two modes for both ***On*** (mode 1) and ***Off*** (mode 0). The motors of the table are switched off in mode 0. In mode 1, the MFB executes the actual control actions by invoking a sequence of function blocks, as shown also in Fig. 8.4.

The second-level state machine (***SSM2***) is the first function block to execute in mode 1. It performs the main control function of the Table controller. ***SSM2*** determines the current state using the variables provided by the input drivers as well as preprocessing function blocks, and controls the execution of the second-level modal function block (***MFB2***). The latter executes control actions based on the state indication it receives from the state machine and supplies information to the output drivers, which are then used to generate the output signals of the actor.

The Table actor model has been designed as a pattern, which has been also used with the other actors of the Production Cell control system. As a result, code manually derived from the rest of the actor models is also similar to the code of the Table actor in terms of the function block patterns used. We have experienced that this method saves a lot of development time; for instance, the time needed to implement the other four actors has been roughly half of the time spent on the first actor. On the other hand, using the framework and predefined components makes it easy to locate implementation errors. It is only necessary to check the system design model, since the implementation follows the principle: "What you design is exactly what you implement". (For the detailed design of each actor in this production cell case study, please refer to [105][106].)

8.1.3 Run-time environment and platforms

The five actors are grouped into three subsystems: Load subsystem, Robot subsystem and Unload subsystem. Each of them is set onto one physical node of the network, where these physical nodes are connected by the CAN bus.

The HARTEXμ is configured for each subsystem allocated onto one physical network node. The actors communicate inside a node and across nodes through

signal-carrying messages, using content-oriented message addressing. In this design, each actor just sends to others its own state as message contents. Fig. 8.5 shows a kernel configuration for the load subsystem developed in a GME-based kernel configuration tool, where two tasks are triggered by the same timer periodically every 30ms. Tasks exchange messages and also access resources in the hardware platform.

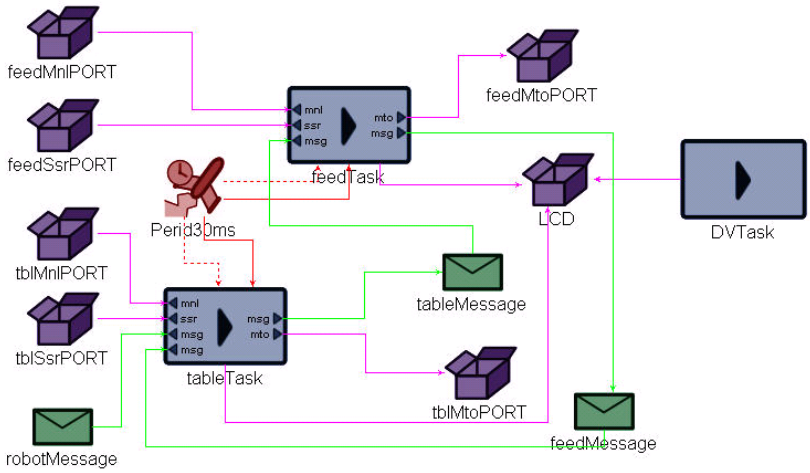


Figure 8.5: Load subsystem configuration

The microcontroller AVR ATmega128 and the STK300 Development kit (Fig. 8.6) have been used to implement the physical node. Physical nodes are connected to a Controller Area Network (CAN). The latter is characterized by a communication protocol, which provides efficient support for real-time communication with a very high level of security. That is why it has been used as a communication bus for the distributed control system of the Production Cell. Unfortunately, the Atmega128 and the STK300 do not have a CAN communication interface. Therefore it was necessary to use a locally designed extension board providing a CAN controller, as well as other peripheral devices such as local keyboard and LCD display.

WinAVR is an open-source cross-development tool-chain for the Atmel AVR series of microcontrollers, which is hosted on the Windows platform. It includes the GNU GCC compiler for the C language. It also contains all the tools for developing software for the AVR family.

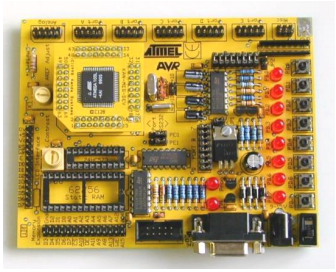


Figure 8.6: StarterKit STK300

8.1.4 Hardware-in-the-loop simulation and related experiments

The developed control system design has been ultimately tested via hardware-in-the-loop simulation involving a real-time control network and an animated computer model of the plant running in a PC. Hardware-in-the-loop simulation can be used to develop and test embedded control systems. It is particularly efficient with complex systems, whenever it is very costly or impossible to use the plant itself in the process of software development [107]. The Production Cell has been simulated on a PC, which performs the same function as the real plant: it responds to the motor controlling signals and sends out sensor signals to the distributed embedded controller, using a dedicated interface implemented with National Semiconductor process I/O boards (Fig. 8.7). The experiments have validated the developed Production Cell control system and have demonstrated the feasibility of the COMDES software design method.

8.2 Case study evolution: a tool demonstration

This section presents the tools and the process of building COMDES applications in the context of the developed Production Cell case study. It assumes that some reusable FB models, prebuilt objects and platform models have been already created with the tools and the process used for component development.

The COMDES toolset is implemented in Eclipse as a number of plug-ins integrated into the Eclipse workbench. The toolset provides an environment supporting model-driven and component-based development of embedded software using the COMDES approach. The spectrum of functionalities supported by the toolset includes:

– Design of the real-time embedded system using graphical modelling tech-

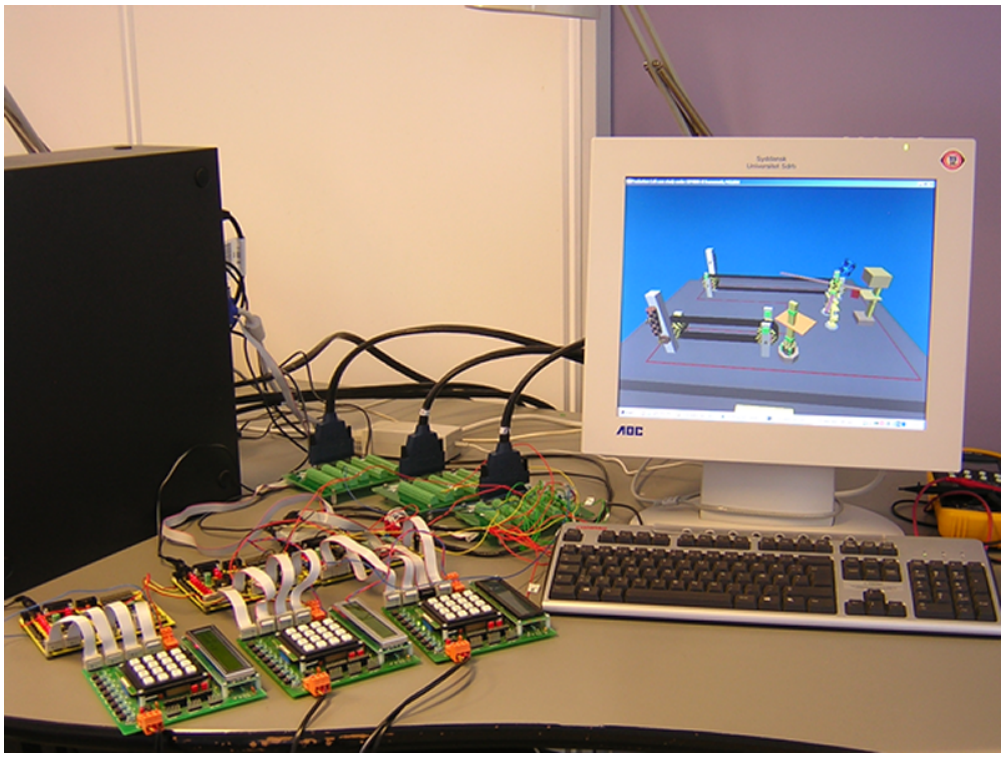


Figure 8.7: Implementation of the Production Cell case study

niques

- Reuse of predefined components
- Detection of syntax and static semantics errors
- Generation of deployable and executable code

This section introduces briefly the basic features of the development tools, and then shows how to create a model and how to get the final executable, using the main functionalities of the toolset.

In Fig. 8.8, the panel in the upper left part of the screen shows a hierarchical representation of the COMDES repository implementation and all types of resources contained in the repository, such as system model files, source code, function block objects, etc. It is the starting point for creating a COMDES system model and working with the rest of the toolset.

Another panel on the left-hand side is a repository root model viewer, which is useful for viewing all types of reusable component, whereby a type can be a function block or a platform. When modelling a COMDES system, function

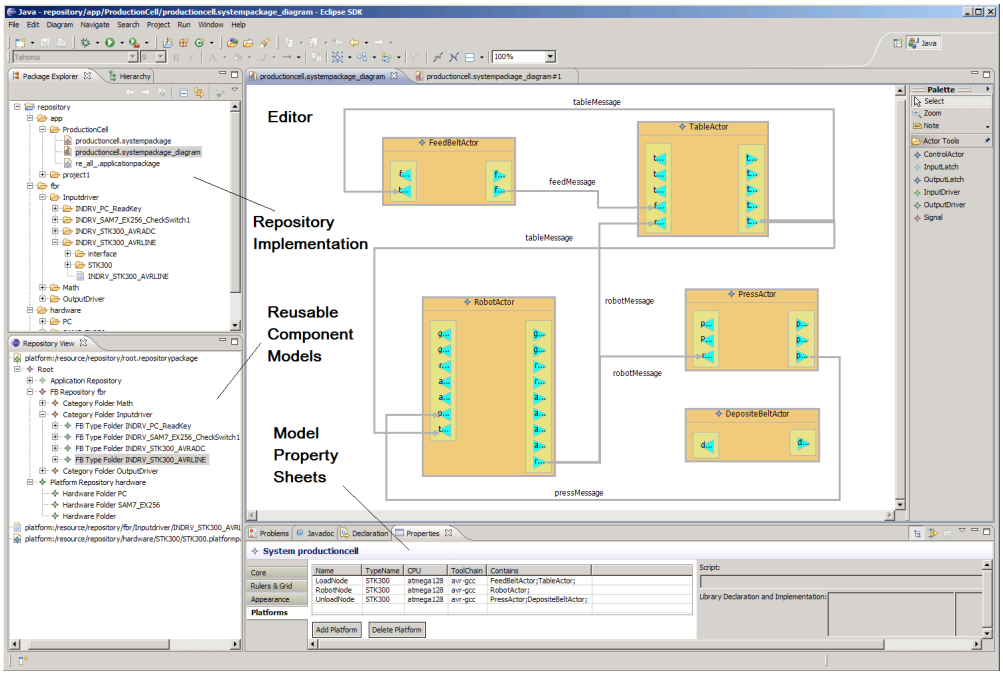


Figure 8.8: Modelling environment

block types listed in the viewer can be instantiated in the editor area, so that corresponding FB instances can be created in the model. One can invoke the action of instantiation from a popup menu when selecting a FB type model, as shown in Fig. 8.9. Alternatively, one can directly drag the type model to an open diagram, in order to create an instance from the type model. If a created instance in an application is a composite FB or a modal FB type, its constituent FB instances will also be created into the application automatically, by means of a underlying algorithm resolving the dependencies.

Just as there are different types of diagrams in COMDES, there are different types of editors. When a model is selected or created, the most appropriate editor will be used to open the model (if it is able of opening). If it is a COMDES system model, the model will be opened using a COMDES system editor (Fig. 8.8). If it is an actor model, it will be opened using an actor editor (Fig. 8.10), which has special features such as the ability to create function block instances.

At the bottom of the editor area is a view called property sheet. The content of this view depends on the model element selected in the corresponding editor. It displays attributes of a model element and allows for their modification. Thus, values of the attributes can be set or reset in the sheet. Additionally, there could be also buttons that provide convenient ways of viewing and modifying values.

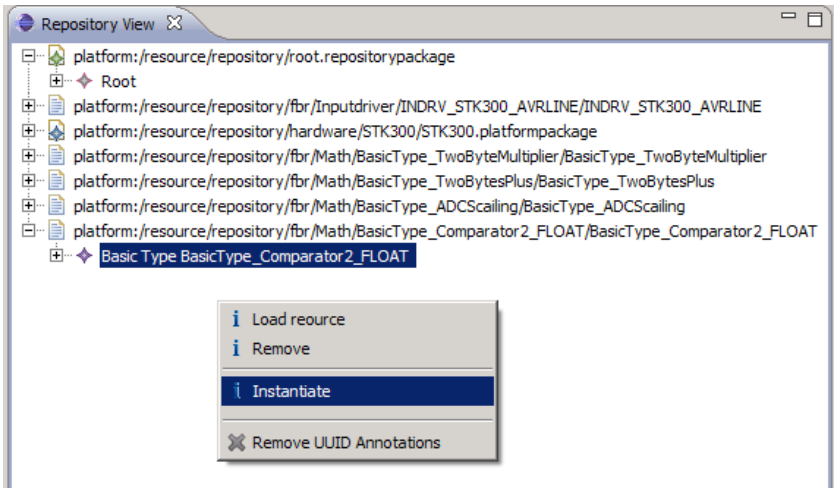


Figure 8.9: Instantiation of a reusable FB model in a repository viewer

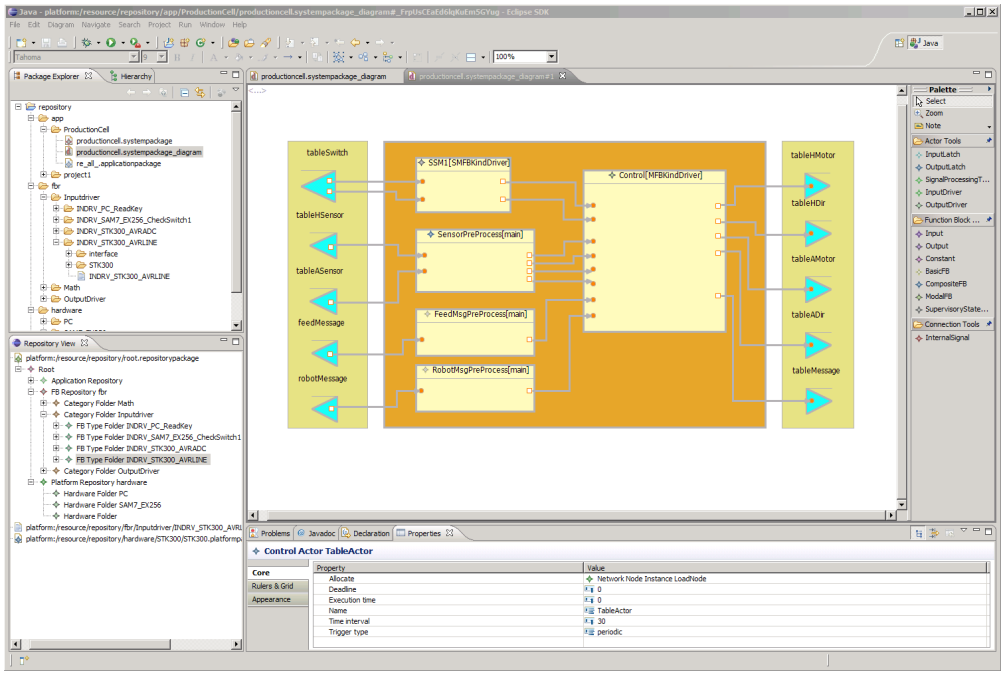


Figure 8.10: Actor editor

For instance, Fig. 8.11 lists all the network instances used in the case study as well as their attributes, when a system model is selected in the system editor, from which one can see how the five actors are allocated.

In addition to the property sheet, repository viewer and editors, the palette in

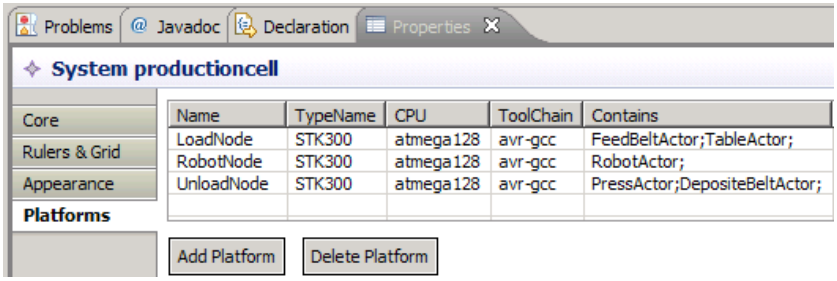


Figure 8.11: Property sheet

the right-hand side of Fig. 8.8 is also important when modelling. Buttons listed in the palette are used to create models (i.e. actor, input, basic FB instance, etc.) in a COMDES system model. The palette can be changed depending on the diagram opened in the current editor.

In addition to creating FB instances from predefined types, another feature is worth mentioning: creating FBs at design-time. When design a system model, one could occasionally find that all the predefined FB types are not the right ones for the application. Thanks to the reuse pattern presented in Chapter 4, during the system modelling stage, the buttons in the palette can be used to create a FB instance directly on an open diagram if a required FB model has not been predefined in the repository. The FB instance is constructed with inputs, outputs, parameters and functions. Furthermore, it can be exported as a predefined type for future reuse.

As shown in Fig. 8.8, the model of the Production Cell control system contains five actors that are interconnected through signals. The model of the Production Cell plant is neglected in order to keep the diagram clean (the plant model is used

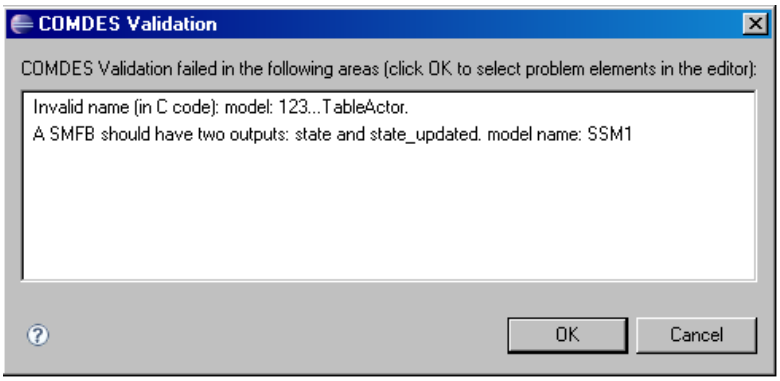


Figure 8.12: Model validation

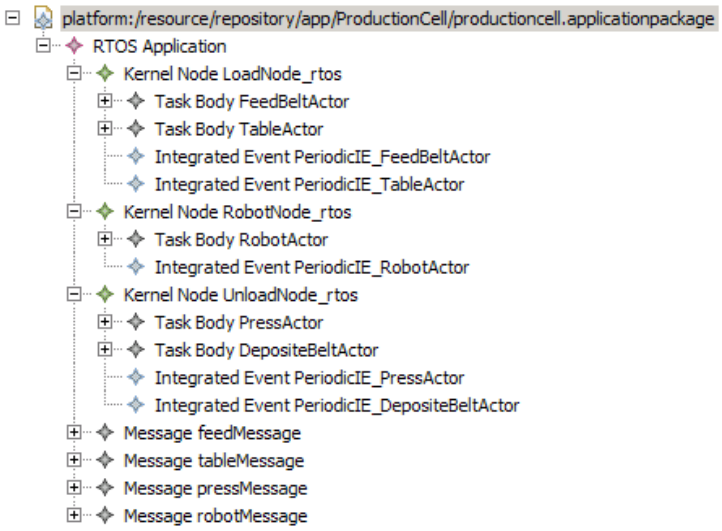


Figure 8.13: Transformed HARTEXμ model

only for the purpose of analysis of the control system model). Fig. 8.10 shows the table actor in detail. This diagram is opened by double clicking the table actor in the former figure. Function blocks or drivers used in this diagram are instantiated from predefined types listed in the repository model viewer shown in Fig. 8.9.

Once a system model has been created, it has to be validated against constraints. In case there is invalid data in the model, the validation fails resulting in the generation of a diagnostic message. Any error must be corrected in order to proceed. Fig. 8.12 illustrates a validation result with two errors found in the system model.

Next, the RTE translator can be invoked on the validated COMDES system model, in order to obtain the corresponding HARTEXμ model. In this case study, each actor is triggered periodically and has no deadline; hence each transformed task is triggered by a timer with a 30 ms period, as illustrated in Fig. 8.13. The tool for manipulating transformed HARTEXμ models has been incorporated in the COMDES development environment, since the meta-model of HARTEXμ has been defined using EMF in the Eclipse. Consequently, developers do not need to switch between different modelling tools during application development.

Finally, it is possible to generate code from the validated system models using the generators and configurator tools. As shown in Fig. 8.14, the *COMDESI-ICGenerator* is mainly intended to generate glue code (instances) from function blocks. In case there is a FB instance model without a predefined type, code for the type must be generated from the instance as well, based on the FB design

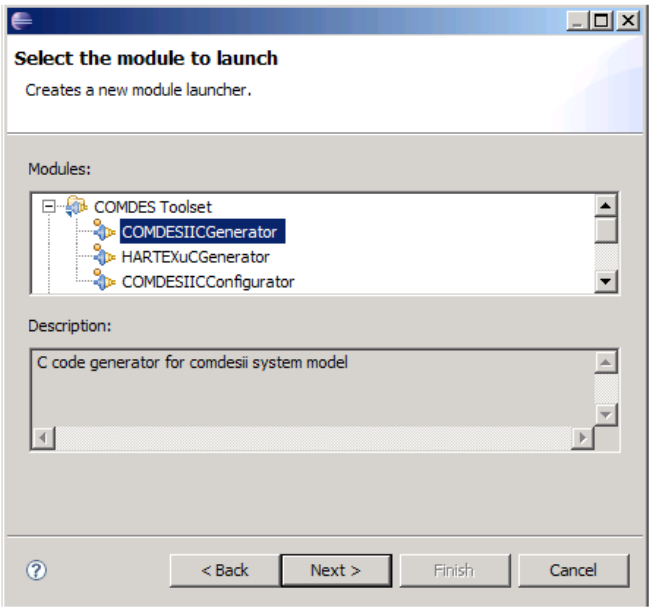


Figure 8.14: Generation tools

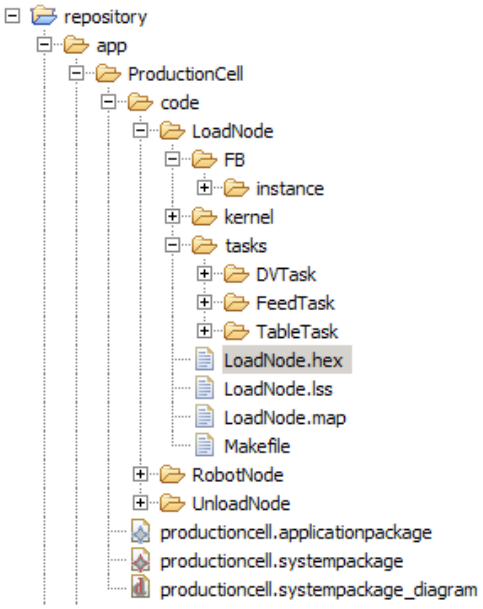


Figure 8.15: Generated executable in the application repository

patterns. The *HARTEXμCGenerator* generates code for kernel objects from the HARTEXμ model. The *COMDESIICConfigurator* takes all models (including also

a repository root model) as input to produce configuration scripts, i.e. makefiles. The scripts must specify what prebuilt FB objects from the repository need to be linked with the generated glue code. Also, they have to specify how to build a binary executable out of the generated source code as well as prebuilt objects, using a compiler and a linker that are specific to a hardware platform.

In the COMDES toolset, GNU Make is an external tool, which controls the generation of program executables from source files. Make gets its knowledge of how to build the program from the makefiles. In addition to source files, the makefile also contains commands that can instruct the compiler to produce an object file and the linker to produce an executable. As shown in Fig. 8.15, the file *LoadNode.hex* is the final executable for the Feed Belt actor and the Table actor. It is specific for the Atmega128 microcontroller and can be directly downloaded into a hardware platform, i.e. the STK300 board.

8.3 Model-driven development tools from industrial perspective

An essential part of the COMDES development environment – the state machine component development tool, has been refined and extended with a new code generator in a case study developed in Motorola A/S, in accordance with industrial requirements. The objective of this case study was to validate the Eclipsed-based technologies for model-driven tool development, and meanwhile provide a method for automatic code generation from a state machine model, offering benefits to Erlang developers in the Zone Controller Development Department of Motorola [108].

In Motorola, developers have been working with Erlang/OTP for a while now, and the need for a higher level of abstraction rather than code has surfaced on a number of occasions. The Erlang/OTP code is very clean, but sometimes it is a lot easier to communicate using pictures and models. It is always a practical problem to keep the pictures and models in sync with the code, so any tool support which can help out with that would be appreciated. Even though writing Erlang/OTP code is a lot faster than doing the same code in other languages, developers still have to write some boilerplate code to implement a component using Erlang/OTP. As a first step, a state machine model is typically conceptualized in some form before this is done. So, a method that could auto-generate code from a state machine model offers some benefits to an Erlang developer. This poses a tough requirement on any modelling tool: using the tool has to be more useful than writing the code by hand.

The design and implementation of the tool have been done in Motorola during a research visit to its Copenhagen division. It follows the concepts and technologies presented in Chapter 7, such as meta-modelling, constraints specification, graphical modelling, model-to-text transformation, etc. The meta-model of the COMDES state machine has been modified and extended in order to fit the requirements of the developer's side. For instance, they would like to add new code manually after code generation from a model, and keep the code after regeneration. Furthermore, to make auto-coding feasible, the programming of a finite state machine needs to be normalized, and developers are only allowed to put code in a number of defined sections. The result of the normalization plays the role of a generation template. In addition, a number of constraints specific to the Erlang/OTP language have been specified and implemented, substituting the original ones developed for COMDES and the C language.

From the point of view of the Ph.D. project, the implementation of the case study has resulted in validation of the presented technologies and approaches concerning model-driven development on the Eclipse platform, as well as a number of feedbacks regarding the application of the model-driven software development approach in industry.

From the point of view of Motorola, the case study is an initial step towards developing tools for model-driven development of applications in Erlang/OTP. It can be continued in several possible directions to further improve developer productivity by introducing more features. These features could also enlighten further development of the COMDES tools.

For example, one of the biggest practical problems Motorola has experienced with various model-driven approaches has been the management of different versions of a component model. Being a large organization with many development centres in the world, Motorola is challenged to share common tools and code with other teams that are not necessarily located in the same region. Typically, these teams have their own additions and changes, sharing a common code base. One problem deals with the task of being able to integrate individually created functionality, as well as that created by separate development centres, into a common code base. This is a fundamental problem, which becomes an issue of potential rework and many headaches when the same code is changed by different people. Working at the source code level, this is a merging problem which requires a significant amount of labour in order to do it consistently.

When adopting a model-driven approach, the problem does not go away – it is merely lifted to a higher level of abstraction, where models and model changes need to be merged. So far, this problem has not been solved well enough to be of

practical use. That has resulted in a situation where models are used to generate the first version of a component, and then all additions and changes are done at the source code level.

Clearly, there are two basic problems that need to be solved in order to make the model-driven approach a good fit for Motorola:

- Integration of manual changes back into the model.
- Merging a new version of a model with the source code based on a previous model plus some manual changes.

When these two problems are solved, it will become possible to solve the general problem of merging changes with the original model.

For another example, sometimes a model-driven approach has to be introduced so as to be combined with an existing code base – more often than not, this is a major practical issue since most freely written code does not easily fit within a model framework. In order to overcome this obstacle, it would be interesting to investigate the use of refactoring tools that assist the programmer in transforming a legacy code base to a format that allows for easy reverse engineering of the code into a model. For instance, Wrangler [109] is a refactoring tool providing a collection of basic refactorings to the program. With the help of the tool, the legacy code can be refactored as close as possible to the generation template. Subsequently, a text-to-model transformation engine can be applied to take the refactored program as input and produce the corresponding model.

8.4 Summary

This chapter has presented the Production Cell case study, which has been systematically developed using the COMDES framework and its toolset. The developed control system has been originally created manually using the COMDES DSL as well as reusable components. The run-time environment and platform used in the case study, including hardware and software aspects, have also been presented.

With the help of meta-models, each part of the designed system can be modelled using appropriate modelling tools, as described in the second part of the chapter. Automatic model transformations are also supported by the corresponding tools. Thus, the implementation of the case study is reduced to manual domain modelling. The rest of the application can then be generated from the models. Thus, developers can concentrate on high-level issues, such as designing domain applications, rather than manually developing the necessary code.

In another case study, an essential part of the COMDES development environment – the state machine component development tool – has been refined and extended with a new code generator, according to requirements specified by Motorola A/S, in an attempt to provide a method for the automatic generation of code from a state machine model, offering benefits to Erlang developers in Morotola. The implementation of the case study has resulted in validation of the presented technologies and approaches concerning model-driven development on the Eclipse platform, as well as a number of feedbacks regarding the application of the model-driven software development approach in industry.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

Building embedded real-time control software systems with components and models has many advantages. It promises a reduction of development costs by enabling rapid development of highly flexible and easily maintainable software systems due to the inherently reusable nature of components. It also provides application developers with a fundamentally different and higher-level methodology that makes it possible to increase software reuse, accommodate embedded applications requirements and reduce the number of errors in the resulting software.

These considerations have motivated the development of a framework – Component-Based Design of Software for Distributed Embedded Systems (COMDES), which is intuitive and easy to use by application experts, because the adopted modelling techniques reflect the true nature of embedded systems, which are predominantly real-time control and monitoring systems.

The framework provides a domain-specific modelling language specifying relevant aspects of system structure and behaviour within the domain of distributed embedded control systems operating under hard real-time constraints. In this framework, the embedded system is composed from actors, which are configured from trusted prefabricated components, such as basic, composite, state machine and modal function blocks. Actors interact by exchanging labelled messages (signals), which provides for transparent communication that is independent of the allocation of actors on network nodes. Signal-based communication is also used for internal interactions involving constituent function blocks. Consequently, actor behaviour is represented as a composition of component functions, and system behaviour – as a composition of actor functions. Different kinds of functional behaviour are treated in separation, i.e. reactive and transformational behaviour, which are delegated to separate components – supervisory state machine and

modal function blocks.

COMDES treats separately functional and timing behaviour, whereby a clocked synchronous model of execution is applied at actor and system levels, i.e. Distributed Timed Multitasking. With this model, input and output signals are latched at task (transaction) start and deadline instants, respectively, resulting in the elimination of I/O jitter at both actor task and transaction levels. The timing aspect of a COMDES system is managed by the underlying run-time environment – the real-time kernel HARTEXμ , which implements the distributed timed multitasking model of computation in the context of COMDES.

COMDES defines components as function blocks. A function block is a component class that may have multiple instances within a given configuration. Reuse of components is realized through three aspects: kind, type and instance. Function blocks are implemented following carefully developed design patterns, and are ultimately implemented as prebuilt executable objects, which are linked to build an application. Objects are physically stored in component repositories in binary format created for a specific platform. The final application implementation thus consists of prebuilt reusable components in the form of data structures and executable algorithms. The former represent component instances and the latter – component types.

This thesis presents a complete meta-model that implements the COMDES framework, including the DSL, function blocks, run-time environment, platform and repositories, in order to provide a detailed and unambiguous description of COMDES models, allowing for an automatic synthesis of systems directly from the models. The meta-model describes formally the COMDES models using class diagrams and constraint specifications, whereby various system aspects are illustrated using graphical notations that model system structure and behaviour in a natural and comprehensive way. Reusable components are formally defined in the meta-model, and the reuse of a component follows a *Kind-Type-Instance* pattern at the meta-model level, allowing for the treatment of all kinds of reusable components in a uniform way. The timing behaviour of a COMDES system is implemented through transformation from a COMDES model to a HARTEXμ model. A meta-model for HARTEXμ is specified, and the transformation rules based on source and target meta-models are defined.

COMDES development is supported by the associated engineering environment (toolset), which consists of a number of tools, such as editor, configurator and generator, etc., as well as various repositories. During operation, models are exchanged among tools, and the final output is executable code for a specific platform. The toolset can support a range of embedded targets with different

compilers, as a result of modelling the hardware and software aspects of a target platform. As long as target platforms are modelled appropriately and low-level functions are wrapped into physical drivers as predefined components, the toolset can generate complete code without requiring any manual coding work.

The toolset supports basic software development functionalities including distributed embedded control system modelling and target executable code generation. From the viewpoint of the classical waterfall development process, the toolset can be used for the design and implementation phases. However, the design and the implementation steps are actually tightly integrated because the implementation is generated directly from models constructed in the design phase.

As a component-based framework, the development in COMDES encompasses two processes: developing reusable components and assembling software systems from software components. Specifically, the toolset supports both the definition of components and the usage of predefined components. The model of a component can be defined with component development tools prior to an application development process. Next, the source code of the component can be generated from the models based on design patterns. Finally, a prebuilt object is derived and stored in the component repository, which is thus available for application development.

With tool support, system modelling is performed in the following sequence: defining the system structure by specifying constituent actors and signals; defining timing behaviour by setting the corresponding attributes of system actors; defining functional behaviour (reactive and transformational) by adding FB models i.e. state machine FB, Modal FB, etc., into actors; defining deployment by allocating the actors to physical platforms. Then the complete system can be implemented by transforming models into code and integrating different reusable component objects into one executable system.

Developing such an engineering environment and the associated tools is a highly complex engineering task. The main challenge is to find a proper solution that is sufficient to implement the COMDES engineering environment. Therefore, this study has been accomplished by performing a survey of the use of different existing model-driven development platforms and tools that can be used as a foundation for building specialized tools for component-based development of embedded software, in order to develop a viable toolset for the COMDES framework. During the survey, a number of key design issues have been identified, which has been instrumental for the development of the COMDES toolset.

As a result, the Eclipse platform has been chosen to implement the software engineering environment, due to its open source property and strong support by constituent model-driven development frameworks, such as Eclipse Modelling

Framework (EMF), Graphical Modelling Framework, etc. It provides a language for meta-modelling, flexible constraint definition and supports strong GUI definition. Parsing models is quite easy using either tailored API or reflective API under the EMF framework. There are also a number of tools available for code generation. Although model reuse is not supported by default, this feature can be implemented in the meta-model following the reuse pattern. Tools can be easily built and integrated into the Eclipse platform through its plug-in mechanism. Moreover, it allows for collaboration between heterogeneous development tools by providing a tool integration solution. Therefore, it is possible to build a seamlessly integrated tool environment for a given development process.

The Eclipse world provides a broad spectrum of model-driven solutions that offer a variety of capabilities. During the study, various solutions have been tried out in order to rapidly implement all the COMDES tools. This thesis presents some of these solutions from the viewpoint of modelling tools, model-to-model transformation tools, model-to-text transformation tools and integration tools. It focuses on a number of possible technologies concerning both model-driven and component-based development, based on the hands-on experience gained during the project.

A prototype implementation of the COMDES engineering environment has been presented in the thesis. Hopefully, it will be useful for both research and industry, and it will serve to advance awareness about the state of the art and provide insights on possible avenues of research and development, regarding embedded software development tools and environments operating on models and components.

9.2 Summary of the Ph.D. project

Component-based design methods for embedded software development usually follow a generative approach, which was originally investigated in the context of executable models and rapid prototyping systems. It is also widely used with industrial automation systems supporting standards like IEC 61131-3 and IEC 61499. This approach can be characterized as computer-aided generation of embedded software out of application models specified in terms of components that are defined at the conceptual modelling and source code levels. This approach requires the complete generation and compilation of executable code, which has to be subsequently downloaded into the target system [12].

Conversely, COMDES emphasizes the use of prefabricated and validated (or trusted) components during the development of embedded systems. In particular,

the components are implemented as prebuilt binary objects saved in the component repository, which makes it possible to configure applications using only glue code, i.e. instance data. This approach requires no generation of executable code from applications. Furthermore, it supports system reconfiguration, which is achieved by updating data structures, whereas executable codes remain unchanged.

The main results obtained during the Ph.D. project are summarized below:

- Meta-models of COMDES-II (a component-based software framework for embedded real-time control systems) have been developed, concerning the definition of a domain-specific language, components, platform, repository, etc., which can be used to specify the structure and behaviour of real-time embedded control systems. The meta-models are complete with respect to the information needed to systematically translate domain-specific models into source code (Chapter 4).

- Meta-models of the runtime environment of COMDES-II applications – the HARTEXμ kernel – have been defined as well. Rules specifying the transformation from a COMDES-II model to a HARTEXμ model have been specified, which makes it possible to transform a platform-independent application model into a platform-specific run-time model with appropriate tool support (Chapter 5).

- Generic design patterns for COMDES-II executable components have been specified at the source-code level, which provide for reusability and reconfigurability of components and component-based applications, and support automatic code generation out of COMDES-II models (Chapter 6, [60]).

- The COMDES-II software design method has been experimentally validated via a case study – the Production Cell Case Study, including modelling, component development, configuration and experimental validation of a distributed real-time control system, as well as formal verification of the developed system using UPPAAL (Chapter 8, [97]).

- Tools supporting the COMDES software development process have been identified and the functionalities of each tool have been defined, including system specification, component specification, component code generation, run-time model transformation, application configuration, etc. (Chapter 7, [90], [86], [110]).

– A number of platforms supporting model-driven software development have been investigated in view of the tool support needed to build the COMDES-II software development environment, covering all aspects of the envisioned software development process. As a result, the Eclipse platform has been chosen to host the COMDES-II toolset (Chapter 2, [111]).

– A number of model-driven tool development issues and technologies based on the Eclipse platform have been investigated, concerning the requirements of COMDES-II toolset. A final solution to the development of the COMDES-II development environment has been given based on the selected technologies (Chapter 2, Chapter 7, [111], [86], [108], [110]).

– A prototype version of a software engineering environment integrating software development tools has been implemented as exchangeable modules within the Eclipse platform supporting embedded software development under the COMDES-II framework (Chapter 7, Chapter 8).

9.3 Future work

Unsurprisingly, all tools have some potential for improvement. Obviously, the usability of the toolset is coupled with the complexity of the offered functionality. As a research work, the current prototype implementation naturally demonstrated some weaknesses concerning user friendliness, documentation, and stability. However, the toolset has some potential for improvement with respect to e.g. stability, speed, window management, integrated help features, etc. Additionally, for efficient support of industrial software development using COMDES, it is important to have more practical functionalities, i.e. generation of documentation, version control, refactoring, model debugging, etc. An industrial version of the COMDES development environment is being developed at the Mads Clausen Institute, University of Southern Denmark, based on the results of this project.

The application models should be proven correct with respect to the required functional and timing behaviour. Accordingly, system behaviour can be analysed using appropriate techniques and tools, e.g. UPPAAL and Simulink, following semantics-preserving transformations from system design models to the corresponding analysis models. Research on transformation from COMDES design models to UPPAAL analysis models has been carried out jointly by the Centre for Embedded Software Systems, Aalborg University and the Mads Clausen Institute, University of Southern Denmark, as a part of the MoDES project. However, more time is needed to develop a translation tool that performs such a transformation,

so as to preserve the semantics of the original COMDES design model [112]. In the foreseeable future, a tool-chain integrating analysis technologies into the current development environment will make it possible to fully support both modelling and analysis of component-based real-time systems. This will lead to a higher level of software quality, especially if analysis is carried out at an earlier stage of development, which will ultimately result in design methodology for embedded systems that are correct by construction.

This project is about developing tools supporting a DSL that can be used for a formal and unambiguous description of embedded applications and for automation of embedded software development, by generating executable code from models. However, as software itself, the structure and behaviour of each tool in the environment should also be specified precisely using some kind of specialized modelling language, taking advantage of the technology of domain-specific modelling. Unfortunately, such a domain-specific language for tool development has not been found during the project. A lot of related research publications just depict tools that support a certain modelling and generation technique in some kind of graphical notation like rectangular boxes and arrows accompanied by explanation in human language (unfortunately, this thesis did the same). Such kind of description, no matter how carefully written, is subject to interpretation and occasional misunderstanding. So, it would be nice to find a way of describing the tools used in the area of MDSD intuitively and unambiguously. At least, as a popular modelling language, UML is a better choice than human language, i.e. component diagram for structure, activity for behaviour, etc., although it is not a language dedicated to tool development. Then, the models of tools can be associated with the meta-models of the supported DSL, so as to enable automatic generation.

Furthermore, the purpose of the toolset is to eventually generate software. But the toolset itself is software that has to be coded manually, due to lack of a proper tool modelling technique. Thanks to the Eclipse platform employed in this work, part of the toolset can be generated from COMDES meta-models resulting in some reduction of the development effort, but a fully automated solution does not exist yet. Hopefully, when modelling techniques for different application domains come to maturity, more research efforts will be launched for model-driven tool development, which constitutes a specific domain as well. At that time, it would be nice to have a technology that unifies both the development of domain applications and associated tool development, where all necessary tools can be generated automatically once a DSL is defined, leading to further improvement of productivity.

Glossary

ADC Analog-Digital Converter, 102
API Application-Programming Interface, 49
ATL ATLAS Transformation Language, 50

BDD Binary Decision Diagram, 141
BFB Basic Function Block, 89
BON Builder Object Network, 41

CALM Cadena Architecture Language with Meta-modelling, 42
CAN Controller Area Network, 102
CBD Component-Based Development, 6
CCM CORBA Component Model, 42
CDT C/C++ Development Tool, 177
CFB Composite Function Block, 68
COMDES Component-based Design of Software for Distributed Embedded Systems, 21
CWI Centrum voor Wiskunde en Informatica, 186

DSL Domain-Specific Language, 16
DSM Domain-Specific Modelling, 16
DSML Domain-Specific Modelling Language, 39
DTM Distributed Timed Multitasking, 66

EET Extended Event Trace, 34
ELF Executable and Linking Format, 152
EMF Eclipse Modelling Framework, 43
eMOF essential MOF, 50

FB Function Block, 57
FBD Function Block Diagram, 91

GME Generic Modelling Environment, 39
GMF Graphical Modelling Framework, 50
GReAT Graph Rewriting and Transformation, 40

HAL Hardware Adaptation Layer, 82

IE Integrated Event, 123
ISIS Institute for Software Integrated Systems, 39

JDT Java Development Tools, 51
JET Java Emitter Templates, 51
JMI Java Metadata Interface, 46

M2M Model-to-Model, 50
M2T Model-to-Text, 51
MDA Model-Driven Architecture, 14
MDSD Model-Driven Software Development, 14
MFB Modal Function Block, 60
MIC Model-Integrated Computing, 39
MOF Meta-Object Facility, 14
MSC Message Sequence Chart, 34

OCL Object Constraint Language, 38
OMG Object Management Group, 14
OS Operating System, 31
OTIF Open Tool Integration Framework, 186

PDE Plug-in Development Environment, 51
PECT Prediction-Enabled Component Technology, 11
PIM Platform Independent Model, 15
PLC Programmable Logic Controller, 10
POU Programming Object, 28
PSM Platform Specific Model, 15
PWM Pulse-Width Modulator, 102

QVT Query/View/Transformation, 38

RTE run-time environment, 163
RTOS real-time operating system, 159

SAnToS The Laboratory for Specification, Analysis, and Transformation of Software, 42
SDM Story Driven Modelling, 45
SDU University of Southern Denmark, 112
SLC State Logic Controller, 138
SMFB State Machine Function Block, 94
SSD System Structure Diagram, 33
SSM Supervisory State Machine, 60
STD State Transition Diagram, 34

TGG Triple Graph Grammars, 46

UML Unified Modelling Language, 14

XMI XML Metadata Interchange, 38
XML Extensible Markup Language, 49, 182

Bibliography

[1] Thomas A. Henzinger and Joseph Sifakis. The Embedded Systems Design Challenge. In *Proceedings of the 14th International Symposium on Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science*, pages 1–15. Springer, August 2006.

[2] Radu Cornea, Nikil Dutt, Rajesh Gupta, Ingolf Krueger, Alex Nicolau, Doug Schmidt, Sandeep Shukla, and Eep Shukla. FORGE: A Framework for Optimization of Distributed Embedded Systems Software. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2003.

[3] Colin Atkinson, Christian Bunse, Hans-Gerhard Gross, and Christian Peper. *Component-Based Software Development*, volume 3778 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.

[4] D. Mcilroy. Mass-Produced Software Components. In Peter Naur and Brian Randell, editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968.

[5] Clemens Szyperski, Dominik Gruntz, and Stephan Mure. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2nd edition, December 2002.

[6] Katharine Whitehead. *Component-Based Development: Principles and Planning for Business Systems*. Addison-Wesley Professional, May 2002.

[7] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer, 1st edition, April 2001.

[8] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, March 2000.

[9] Rob van Ommering. Koala, a Component Model for Consumer Electronics Product Software. In *Development and Evolution of Software Architectures for Product Families*, volume 1429/1998 of *Lecture Notes in Computer Science*, pages 76–86. Springer Berlin/Heidelberg, January 1998.

[10] Bruno Bouyssounouse and Joseph Sifakis. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *Lecture Notes in Computer Science*. Springer, 2005.

[11] Anders Möller, Mikael Åkerholm, Johan Fredriksson, and Mikael Nolin. Software Component Technologies for Real-Time Systems - An Industrial Perspective. In *WiP Session of Real-Time Systems Symposium (RTSS)*, December 2003.

[12] Christo Angelov, Krzysztof Sierszecki, and Nicolae Marian. Component-Based Design of Embedded Software: An Analysis of Design Issues. In Nicolas Guelfi, Gianna Reggio, and Alexander B. Romanovsky, editors, *Scientific Engineering of Distributed Java Applications, 4th InternationalWorkshop (FIDJI 2004)*, volume 3409 of *Lecture Notes in Computer Science*, pages 1–11, Luxembourg-Kirchberg, Luxembourg, November 2004. Springer.

[13] Kurt C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, April 2003.

[14] Damir Isovic and Christer Norström. Components in Real-Time Systems. In *Proceedings of the The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA 2002)*, pages 135–139, March 2002.

[15] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew Black, Peter Müller, Christian Zeidler, Thomas Genssler, and Reinier Van Den Born. A Component Model for Field Devices. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer-Verlag, July/August 2002.

[16] Andree Blotz, Franz Huber, Heiko Loetzbeyer, Alexander Pretschner, Oscar Slotosch, and Hans-Peter Zaengerl. Model-Based Software Engineering and Ada: Synergy for the Development of Safety-Critical Systems. In *ADA Deutschland Tagung*, Jena, Germany, March 2002.

[17] Franz Huber Sascha, Franz Huber, Sascha Molterer, Andreas Rausch, Bernhard Schätz, Marc Sihling, and Oscar Slotosch. Tool Supported Specification and Simulation of Distributed Systems. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 155–164. IEEE Computer Society, 1998.

[18] Franz Huber, Bernhard Schätz, Alexander Schmidt, Er Schmidt, and Katharina Spies. AutoFocus - A Tool for Distributed Systems Specification. In *Proceedings FTRTFT96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. Springer Verlag, 1996.

[19] Christo Angelov and Krzysztof Sierszecki. A Software Framework for Component-Based Embedded Applications. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 655–662, Busan, Korea, November/December 2004.

[20] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, pages 199–208, Daegu, Korea, August 2007. IEEE Computer Society.

[21] Markus Völter and Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 1st edition, May 2006.

[22] Mark Dalgarno. Model-Driven Software Development - Ready for Prime Time? *MSDN architecture newsletter*, 2, April 2007.

[23] The Object Management Group. MDA Guide Version 1.0.1, June 2003.

[24] The Object Management Group. A Proposal for an MDA Foundation Model, V00-02, ormsc/05-04-01.

[25] Steve Cook. Domain-Specific Modeling and Model Driven Architecture. *MDA Journal*, pages 2–10, January 2004.

[26] D. Djuric, D. Gaševic, and V. Devedžic. The Tao of Modeling Spaces. *Journal of Object Technology*, 5:125–147, November-December 2006.

[27] The Object Management Group. Meta Object Facility(MOF) Specification, Version 1.4, Aprial 2002.

[28] S. Gérard, D. Petriu, and J. Medina. MARTE: A New Standard for Modeling and Analysis of Real-Time and Embedded Systems. In *19th Euromicro Conference on Real-Time Systems (ECRTS 2007)*, Pisa, Italy, July 2007.

[29] Krzysztof Sierszecki. *Component-Based Design of Software for Embedded Systems*. PhD thesis, University of Southern Denmark, Soenderborg, Denmark, 2007.

[30] Xu Ke. *Model-Based Design and Analysis of Embedded Software*. PhD thesis, University of Southern Denmark, Soenderborg, Denmark, 2008.

[31] Christo Angelov, Xu Ke, and Krzysztof Sierszecki. A Component-Based Framework for Distributed Control Systems. In *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2006)*, pages 20–27, Dubrovnik, Croatia, August/September 2006. IEEE Computer Society.

[32] Bernhard Schätz, Tobias Hain, Frank Houdek, Wolfgang Prenninger, Martin Rappl, Jan Romberg, Oscar Slotosch, Martin Strecker, and Alexander Wisspeintner. CASE Tools for Embedded Systems. Technical Report TUM-I0309, Technische Universität München, July 2003.

[33] Hanne R. Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[34] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The Generic Modeling Environment. In *Proceedings of the Workshop on Intelligent Signal Processing (WISP 2001)*, 2001.

[35] Gabor Karsai, Miklos Maroti, Akos Ledeczi, Jeff Gray, and Janos Sztipanovits. Composition and Cloning in Modeling and Meta-Modeling. *IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling)*, 12:263–278, 2004.

[36] Aniruddha Gokhale, Douglas C. Schmidt, Balachandran Natarajan, and Nanbor Wang. Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. *Communications of the ACM*, 45(10):65–70, 2002.

[37] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

[38] Jonathan Sprinkle, Aditya Agrawal, Tihamer Levendovszky, Feng Shi, and Gabor Karsai. Domain Translation Using Graph Transformations. In *Tenth IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 159–168, Huntsville, AL, April 2003.

[39] Georg Jung and John Hatcliff. A Type-centric Framework for Specifying Heterogeneous, Large-scale, Component-oriented, Architectures. In *Proceedings of the 6th international conference on Generative programming and component engineering (GPCE 2007)*, pages 33–42, Salzburg, Austria, 2007. ACM.

[40] Georg Jung, John Hatcliff, Adam Childs, Matt Hoosier, Jesse Greenwald, and Alley Stoughton. *Overview of Cadena's Architecture Definition Language and Meta-modeling Framework.* International Summer School on Tool-based Rigorous Engineering of Software Systems (STRESS 2006) Lecture Slide, 2006.

[41] Adam Childs, Jesse Greenwald, Georg Jung, Matthew Hoosier, and John Hatcliff. CALM and Cadena: Metamodeling for Component-Based Product-Line Development. *Computer*, 39(2):42–50, 2006.

[42] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science*, pages 361–375. Springer Verlag, 2006.

[43] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. *Theory and Application of Graph Transformations*, 1764:296–309, 2000.

[44] Felix Klar, Alexander Königs, and Andy Schürr. Model Transformation in the Large. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC-FSE 2007)*, pages 285–294, Dubrovnik, Croatia, 2007. ACM.

[45] Carsten Amelunxen, Alexander Königs, and Tobias Rötschke. MOSL: Composing a Visual Language for a Metamodeling Framework. In J. Howse

J. Grundy, editor, *IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2006)*, pages 81–84. IEEE Computer Society, 2006.

[46] MetaCase. Upgrading a DSM and code generation tool. *Embedded Systems Europe*, October 2006.

[47] Juha pekka Tolvanen, Risto Pohjonen, and Steven Kelly. Advanced Tooling for Domain-Specific Modeling: MetaEdit+. In J. Sprinkle, J. Gray, M. Rossi, Tolvanen, and J.-P., editors, *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007)*, Technical Reports, TR-38. University of Jyväskylä, Finland, 2007.

[48] MetaCase. *The Graphical Metamodeling Example*, in MetaCase document no. GE-4.5, 2nd edition, February 2008.

[49] MetaCase. http://www.metacase.com/, 2009.

[50] MetaCase. *Integrating with other environments.* MetaCase, 2009.

[51] Sanna Sivonen. Domain-specific modelling language and code generator for developing repository-based Eclipse plug-ins. Technical Report VTT PUBLICATIONS 680, VTT Technical Research Centre of Finland, 2008.

[52] Xu Ke and Krzysztof Sierszecki. Generative Programming for a Component-based Framework of Distributed Embedded Systems. In *Proceedings of the 6th OOPSLA Workshop on Domain Specific Modeling*, Portland, Oregon, USA, October 2006.

[53] Aitor Aldazabal, Terry Baily, Felix Nanclares, Andrey Sadovykh, Christian Hein, and Tom Ritter. Automated Model Driven Development Processes. In *Proceedings of the ECMDA workshop on Model Driven Tool and Process Integration*, pages 361–375. Fraunhofer IRB Verlag, June 2008.

[54] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776, December 1997.

[55] Jie Liu and Edward Lee. Timed Multitasking for Real-Time Embedded Software. *IEEE Control Systems Magazine*, 23:65–75, 2002.

[56] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. Event-driven Programming with Logical Execution Times. In *Hybrid Systems: Computation and Control, 7th International Workshop (HSCC 2004)*, volume 2993 of *Lecture Notes in Computer Science*, pages 357–371, Philadelphia, PA, USA, March 2004. Springer.

[57] Hans Hansson, Mikael Åkerholm, Ivica Crnkovic, and Martin Törngren. SaveCCM – A Component Model for Safety-Critical Real-Time Systems. In *Proceedings of the 30th EUROMICRO Conference (EUROMICRO 2004)*, pages 627–635. IEEE Computer Society, 2004.

[58] R. W. Lewis. *Modelling Control Systems Using IEC 61499: Applying Function Blocks to Distributed Systems.* The Institution of Engineering and Technology, July 2001.

[59] Paul Caspi. Some Issues in Model-Based Development for Embedded Control Systems. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems, IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*, volume 225 of *IFIP*, pages 9–13, Braga, Portugal, 2006. Springer.

[60] Christo Angelov, Xu Ke, Yu Guo, and Krzysztof Sierszecki. Reconfigurable State Machine Components for Embedded Applications. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2008)*, pages 51–58, Parma, Italy, September 2008. IEEE Computer Society.

[61] Xu Ke, Paul Pettersson, Krzysztof Sierszecki, and Christo Angelov. Verification of COMDES-II Systems Using UPPAAL with Model Transformation. In *Proceedings of the 14th IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008)*, pages 153–160, Kaohisung, China, August 2008. IEEE Computer Society.

[62] Krzysztof Sierszecki, Christo Angelov, and Xu Ke. A Run-Time Environment Supporting Real-Time Execution of Embedded Control Applications. In *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2008)*, pages 61–68, Kaohsiung, China, August 2008. IEEE Computer Society.

[63] Christo Angelov, Krzysztof Sierszecki, Nicolae Marian, and Jinpeng Ma. A Formal Component Framework for Distributed Embedded Systems. In

Proceedings of CBSE 2006, volume 4063/2006 of *Lecture Notes in Computer Science*, pages 206–221, 2006.

[64] Christo Angelov, Krzysztof Sierszecki, and Yu Guo. Formal Design Models for Distributed Embedded Control Systems. In *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)*, pages 43–57, Denver, Colorado, USA, 2009.

[65] Axel Jantsch. *Modeling Embedded Systems and SoC's: Concurrency and Time in Models of Computation*. Morgan Kaufmann, 1st edition, June 2003.

[66] William Henderson, David Kendall, and Adrian Robson. Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems Computing Minimal Response Times and Reducing Jitter. *Real-Time Systems*, 20(1):5–25, January 2001.

[67] Søren Top, Hans Jørgen Nørgaard, Brian Krogsgaard, and Bo Nørregaard Jørgensen. The Sandwich Code File Structure: An architectural support for software engineering in simulation based development of embedded control applications. In *Proceedings of IASTED International Conference on Software Engineering (SE 2004)*, pages 196–201, Innsbruck, Austria, February 2004.

[68] Søren Top, Hans Jørgen Nørgaard, Brian Krogsgaard, and Bo Nørregaard Jørgensen. Object Oriented C++ Programming in SIMULINK(r): A reengineered simulation architecture for the control algorithm code view. In *Proceedings of Nordic MATLAB Conference 2003*, pages 79–84, 2003.

[69] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[70] Johan Bengtsson and Wang Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lecture Notes on Concurrency and Petri Nets*, number 3098 in LNCS, pages 87–124, 2004.

[71] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide.* Addison-Wesley Professional, August 2003.

[72] Kleanthis Thramboulidis, G. Doukas, and A. Frantzis. Towards an Implementation Model for FB-Based Reconfigurable Distributed Control Applications. In *7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 193–200, Vienna, Austria, May 2004. IEEE Computer Society.

[73] Axel Jantsch and Ingo Sander. Models of Computation and Languages for Embedded System Design. *IEE Proceedings on Computers and Digital Techniques*, 152(2):114–129, March 2005.

[74] Mo YongTeng. *Design Pattern (C#/Java).* TsingHua University Press, 2006.

[75] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, September 2001.

[76] Christo Angelov, Ivan Ivanov, and Alan Burns. HARTEX - a Safe Real-Time Kernel for Distributed Computer Control Systems. *Software: Practice and Experience*, 32(3):209–232, March 2002.

[77] Christo Angelov and Jesper Berthing. A Jitter-Free Kernel for Hard Real-Time Systems. In Zhaohui Wu, Chun Chen, Minyi Guo, and Jiajun Bu, editors, *Embedded Software and Systems, First International Conference (ICESS 2004)*, volume 3605 of *Lecture Notes in Computer Science*, pages 388–394, Hangzhou, China, December 2004. Springer Berlin / Heidelberg.

[78] Christo Angelov and Jesper Berthing. Distributed Timed Multitasking - A Model of Computation for Hard Real-Time Distributed Systems. In Bernd Kleinjohann, Lisa Kleinjohann, Ricardo Jorge Machado, Carlos Eduardo Pereira, and P. S. Thiagarajan, editors, *Proceedings of the 5th IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*, volume 225 of *IFIP*, pages 145–154, Braga, Portugal, October 2006. Springer.

[79] Christo Angelov and Jesper Berthing. A Timed Multitasking Architecture for Distributed Embedded Systems. In *Proceedings of the IEEE Second In-*

ternational Symposium on Industrial Embedded Systems (SIES 2007), pages 102–109, Lisbon, Portugal, July 2007.

[80] Jesper Berthing. *Component-Based Design of Safe Real-Time Kernels for Embedded Systems*. PhD thesis, University of Southern Denmark, Soenderborg, Denmark, 2008.

[81] Arnold S. Berger. *Embedded Systems Design: An Introduction to Processes, Tools and Techniques*. CMP Books, 1st edition, December 2001.

[82] Christo Angelov, Krzysztof Sierszecki, and Nicolae Marian. Design Models for Reusable and Reconfigurable State Machines. In Laurence Tianruo Yang, Makoto Amamiya, Zhen Liu, Minyi Guo, and Franz J. Rammig, editors, *Embedded and Ubiquitous Computing*, volume 3824 of *Lecture Notes in Computer Science*, pages 152–163, Nagasaki, Japan, December 2005. Springer Berlin / Heidelberg.

[83] Rolf Drechsler and Jochen Römmler. Implementation and Visualization of a BDD Package in JAVA. Technical report, GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 2002.

[84] Tool Interface Standard (TIS). Executable and Linking Format (ELF) Specification Version 1.2, May 1995.

[85] John R. Levine. *Linkers and Loaders*. Morgan Kauffman, 1st edition, October 1999.

[86] Yu Guo, Krzysztof Sierszecki, and Christo Angelov. COMDES Development Toolset. In *Proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008)*, pages 233–238, Malaga, Spain, September 2008.

[87] Magnus Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. PhD thesis, Uppsala University, Uppsala, Sweden, 2000.

[88] Bixin Li. Managing Dependencies in Component-Based Systems Based on Matrix Model. In *Proceedings of Net.Object.Days 2003*, pages 22–25, 2003.

[89] Yu Guo. COMDES Function Blocks Design and Configuration. Master's thesis, University of Southern Denmark, Soenderborg, Denmark, June 2005.

[90] Yu Guo, Krzysztof Sierszecki, and Christo Angelov. A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems. In *Proceedings of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008)*, pages 1315–1320, Turku, Finland, July/August 2008. IEEE Computer Society.

[91] Jos Warmer and Anneke Kleppe. *Object Constraint Language, The: Getting Your Models Ready for MDA*. Addison Wesley, 2nd edition, September 2003.

[92] Christian W. Damus. Implementing Model Integrity in EMF with MDT OCL. Eclipse Corner Articles, February 2007.

[93] Fangjian Hu. COMDES-II Development Editing and Repository Environment. Master's thesis, University of Southern Denmark, Soenderborg, Denmark, June 2008.

[94] Gary Cernosek. Next-generation model-driven development. IBM Software Group, 2004.

[95] ATLAS group. *ATL:Atlas Transformation Language ATL Starter Guide version 0.1*, December 2005.

[96] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[97] Yu Guo, Feng Zhou, Nicolae Marian, and Cristo Angelov. Hardware-in-the-Loop Simulation of Component-Based Embedded Systems. In *Proceedings of the 8th International Workshop on Research and Education in Mechatronics (REM 2007)*, Tallinn,Estonia, June 2007.

[98] Jan A. Bergstra and Paul Klint:. The Discrete Time ToolBus – A Software Coordination Architecture. *Science of Computer Programming*, 31(2-3):205–229, 1998.

[99] Ian Thomas and Brian A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, 9(2):29–35, 1992.

[100] Hayco de Jong and Paul Klint. ToolBus: The Next Generation. In *Formal Methods for Components and Objects, First International Symposium*, volume 2852 of *Lecture Notes in Computer Science*, pages 220–241. Springer, 2003.

[101] Gabor Karsai, Andras Lang, and Sandeep Neema. Design Patterns for Open Tool Integration. *Journal of Software and System Modeling*, 4(2), May 2005.

[102] Prawee Sriplakich. *ModelBus – An Open and Distributed Environment for Model Driven Engineering*. PhD thesis, University Pierre and Marie Curie, France, 2007.

[103] Shen Wei. Tool Integration for COMDES Development Environment. Master's thesis, University of Southern Denmark, Soenderborg, Denmark, June 2008.

[104] A. Burns. How to Verify a Safe Real-Time System The Application of Model Checking and a Timed Automata to the Production Cell Case Study. Technical report, Real-Time Systems Research Group, Department of Computer Science, University of York, 1998.

[105] Xu Ke and Yu Guo. Case Studies of Component-based Design for Distributed Embedded Systems. Technical report, Mads Clausen Institute, University of Southern Denmark, Soenderborg, Denmark, June 2006.

[106] Feng Zhou. Component-based Design of Embedded System – Production Cell Case Study. Master's thesis, University of Southern Denmark, Soenderborg, Denmark, June 2006.

[107] Martin Gomez. Hardware-in-the-Loop Simulation. *Embedded Systems Design*, 14(13), December 2001.

[108] Yu Guo, Torben Hoffmann, and Nicholas Gunder. Autocoding State Machine in Erlang. In *Proceedings of the 14th International Erlang/OTP User Conference*, Stockholm, Sweden, November 2008.

[109] Huiqing Li, Simon Thompson, George Orosz, and Melinda Toth. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *Proceedings of the Seventh ACMSIGPLAN Erlang Workshop*, September 2008.

[110] Kebin Zeng, Yu Guo, and Christo Angelov. Graphical Model Debugger Framework for Embedded Systems. In *Accepted for presentation to the 13th DATE Conference and Exhibition: Design, Automation and Test in Europe (DATE2010)*, Dresden, Germany, March 2010.

[111] Yu Guo, Krzysztof Sierszecki, and Christo Angelov. Model-Driven Development of Domain-Specific Applications: Tool Support. In *Proceedings of*

11th Symposium on Programming Languages and Software Tools and 7th Nordic Workshop on Model Driven Software Engineering (SPLST 2009 and NW-MODE 2009), pages 225–239, Tampere, Finland, August 2009.

[112] István Knoll, Anders P. Ravn, and Arne Skou. Semantics for communicating actors with interdependent real-time deadlines. *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 29–35, 2009.

Appendix: Model-Driven Software Development in Industry

Question

Is the Model-Driven Development approach company-widely used to develop software products?

Say, given a business domain, first model the system using UML or some domain-specific language. Then generate part or all of code from the model. If bugs are found, fix the model instead of fix the code, as the new code will be generated again. Is this approach popular? Why?

Answers

The answers are not ordered. The appendix only shows the answers that have been approved by the people who answered the question. They have agreed that their answers with their names, and professional headlines can be listed here.

Probal DasGupta, Software Engineering, Enterprise Systems, Business Agility Executive: The goal of MDA (Model Driven Architecture) is to enable the solution design to occur in a PIM (Platform Independent Model) where your ideas are not constrained by hardware/software considerations; therefore, a pure business person, equipped with and trained on the right BPM tool, can produce the required business model of the target solution. After a complete and fully validated business model is saved in the BPM world, tools can be used for converting the PIM (Platform Independent Model) into a PSM (Platform Specific Model) that deals with hard implementation details like the target hardware/-software platform and generates the source code. Instead of maintainable source code, you get a maintainable business model. The biggest benefit of this approach is BUSINESS AGILITY. An organization will be able to react much faster to

internal and external changes because I.T. will be closely aligned with Business, and not perpetually 6 months behind it. This is the goal (and the dream?) of the MDA world, which is closely aligned with the goal and the dream of the business process management world – which is why, in recent years, omg.org (the proponent of MDA technology) and bmpi.org have merged and become one organization. I do agree with previous answers (a) that the toolset is in most cases not there yet; and (b) that it is better used in Europe than in the United States. Full and complete implementations of MDA do exist, as I discovered during my visit to CeBIT 2006 - the world's largest ICT fair held in Hannover, Germany every year in March. The Gartner Group has called MDA an "outsourcing buster" solution. One of its reports states that while outsourcing to offshore organizations typically saved companies 35% in ultimate costs, a full-fledged adoption of MDA should save organizations 45% or more, because they would not require companies to maintain large armies of programmers anywhere (whether in the U.S. Or offshore). So MDA/MDD is the technology of the future that will come into its own and has a good chance of becoming the prevalent software engineering paradigm. New IDEs, Agile methodologies (SCRUM/XP) cannot compete with MDA, because they tie the organization down to PSMs (platform specific models) that require I.T. experts, and makes organizations more dependent on technologists. MDA proposes make organizations depend only on "technology", not "technologists". In an ideal MDA world, software talent will be used for developing wonderful new tools that made the MDA concept more and more viable, but they will not be involved in business programming. Only be eliminating the "technologists" from the chain of delivery and replacing them with the proper "technology" that is handled by the business persons themselves, can true business agility be achieved – which is extremely important for companies to maintain market leadership and general excellence. I strongly suspect we shall see greater adoption of MDA concepts in the immediate future.

Brian Shannon, Software Engineer and Architect: This approach is generally not used because of several reasons. The first is that while UML models can express the structure of classes, interfaces, etc., the implementations and algorithms or business logic of those classes are, generally speaking, not represented in UML. Some basic implementations can be generated (e.g. Plain Old Java Object [POJO] classes), and in DSLs this can be expanded, but ultimately in non-trivial applications, there will be custom code that has to be written.

So, after generating the skeletons of each entity in the model, developer still needs to fill those skeletons with implementations. This leads to making changes

to the model in the code, because when writing code, as we all know, the reality of the situation and/or design sets in and changes need to be made. This leads to the need to maintain and keep in sync both a code base and a UML model, which is quite a chore without too many advantages.

Also, as you mention with regard to maintaining the code, unless the code is only and always generated from the model, and the model completely represents the business logic and implementation needed, you will not be able to do this without some creative thinking. There are techniques such as Aspect Oriented Programming (AOP) that one could use and even some great frameworks out there than allow for loosely coupled components that can be swapped out easily, but it seems that the effort required to keep the model in sync with the code and generate the proper code would probably be more effort than just updating the code itself especially with modern IDEs and their ability to refactor existing code/design quickly.

If using MDD, a balanced approach might be best. Consider the parts of the application that would benefit from this (POJOs and well-defined DSL implementations), but also perhaps generate the initial code base from the model and then just start from there. It can save some of the initial tediousness of translating the model to code.

Dennis Andrews, VP of Sales, Artisan Software Tools - Mission and Safety Critical Engineering Solutions: Model Driven Development is most widely used within: Aerospace, Automobile, Defence, Medical, Network Communications industries.

An MDD solution should generate bug free code, if you find a bug, then it may be several things. Most MDD tools will not allow you to perform a build without doing a series of static checks.

Even when using MDD, some code is still hand written. The hand written code come from any legacy code that has been integrated into the model or the transition code within a state chart.

Some MDD solutions do allow users to modify portions of the generated code. These changes can automatically be reflected back into the model.

Every MMD solution offers varying degrees of control over code generation. This control allows users to set properties or apply rules, which will ensure that the generated code adheres to specific guidelines.

UML has evolved to provide a baseline for many domain specific languages: Business Processes, Systems Engineering and Defence. Supporting a broader spectrum of Domain Specific Languages using one base language supports a more

seamless flow from Requirements to Business Processes to System Architectures to Software Designs to Code.

MDD is most popular among people who know the technology and have used it effectively to increase quality and productivity. This number is growing.

There are many reasons for conflicting responses to your question, however. There is a broad spectrum of solutions which offer a variety of capabilities. In addition, the deployment of MDD is often uneven within industries and even companies. All these factors can lead to misconceptions.

Dan Rosanova, Principal Consultant at Nova Enterprise Systems: To a certain extent I think it is. I generally start with a domain analysis and eventually create my business objects from that. I actually like to use Use Cases and then circle the nouns.

I like to keep my generated code separated from my manual code. Since I work in C# I benefit from partial classes, but there are other techniques you can use as well. The big problem with generation is that you often become tied to it and it can hold back development. I certainly think that for a large portion of your development this is a great idea. I think a lot of developers don't really get UML so it's harder for them to work that way.

Niresh Agarwal, Principal Software Engineer/Tech Lead at Teradyne: I agree with Vasilij and Dan. And I do think that though the technology is not there yet, it is going to be tomorrow's high level language (not necessarily UML, but some modelling approach), where future programmers will program their designs into models and auto code generation tools will generate efficient and custom code particular to the needs of that application.

Todd Hansen, Software Development Consultant: If you're working in the Microsoft side of the development world, then there actually is a tool that works quite well for doing exactly as you suggested. As of Visual Studio 2005 (and 2008), you can create class diagrams within Visual Studio – as part of the project – and any changes made to the diagram will be reflected in the code ...and vice versa. It actually works really well, and I've used it on several projects.

Mark Meyer, Principle/Real-time Embedded Software Engineering Consultant at Differential Designs, Inc.: MDD (or Model Based Development) is currently being used in the automotive industry in designing and developing electronic modules for use in vehicles. You first model it then autocode it

(generate code). In this environment, timing and memory constraints are critical. Yet, its meeting those constraints while use is increasing.

Venkat Pula, Modeling Tools Sales Specialist: It depends on the nature of the bug. If you are referring to a design or model bug, it makes sense to fix the model, and regenerate the code. If the bug is computational in nature, then, it makes sense to fix the bug in code itself.

The other aspect to think about is your UML tool's capability to update its model when the code is changed. If your tool supports such a capability, then, you can fix the problem in either side. For example, Telelogic's UML tool, Rhapsody, supports DMCA (Dynamic Model Code Associativity) where any changes occur in model are automatically updated in generated code, and vice versa. Hope this info helps.

Chris Lema, Software product development professional (VP/CTO): Is this approach popular? It depends on the computing sector. In the embedded software space, companies have been using technology products to produce code generated solutions like you are describing. Products like those from MetaCase allow users to do exactly that. In the enterprise software space however, there are fewer tools that have gone the whole distance. Many prefer to stop before real logic is generated – because describing that logic is difficult in languages like UML. That said, a few companies have built complete tools that use formal logic languages and a visual model compiler to do the very thing described in your question, though predominantly in Europe rather than in the US. In the end, as others have shared, the popularity of this approach is limited by the maturity of the toolsets available. Further, the requirement to think and design at a higher level of abstraction is a harder skill to manage/train and this may also be a factor.

Richard Tabor Greene, Professor of Knowledge and Creativity Management at Kwansei Gakuin University: The problem is UML just models the mess that is there now in the world as present business processes – automating a mess in software form is immensely wasteful and problematic over the long term.

In my old book Global Quality (at amazon.com) the chapter on Taguchi shows my approach at Xerox PARC – namely:

1. find what in the business process causes its outputs to displease customers receiving them

2. improve the process to reduce or eliminate those displeasers

3. rigorously eliminate all wastes in the process, that is, all steps that do not directly contribute value to end users of the outputs of the process

4. develop software to automate those parts of the process not better do-able by persons

5. measure to make sure that the final customers sat with the new version software enabled is higher than the customers sat with the old version of the process.

By directing software at a version of a business process first cleaned up in total quality ways, a much more cogent, lean, and focussed-on-customer version of the business process gets automated into software form.

Vilas Prabhu, Consultant Chief Architect: I am aware of 'elaborationist' MDD (generating part of code) being put in practice, by many a organisation for their product development. This is mainly because of economy of scale provided by this approach. For product companies it makes sense to use this approach because its easy to churn out various product versions (targeted at different technologies), which are only slightly different than other versions, while maintaining the standardisation across versions.

My observation is that it is not a very popular approach for general purpose software development in a non-product scenario. When you are doing bespoke software development, economy of scale is of no use. However you may still want to use MDD for quality reasons. But current tooling is esoteric and not standardised, so decision makers deem it to be risky and don't allow it to be used. They rely on good resources to provide good quality, instead.

Prashant Hegde, Systems Architect at Honeywell: Theoretically, it should work fine. Practically, there are issues. From the UML point of view, it uses three main diagrams for the code generation. Object diagrams, class diagrams and StateCharts/Diagrams. Other UML constructs are not very useful from code generation perspective.

Now, if you want to generate 100% code generation, you need to model everything(to the lowest level detail) in the modelling tool. With this approach you get into the same issues as the code maintenance. Take, for example, the case of Rhapsody, it allows writing code within the model to achieve 100% code generation! Frequently the code generated from the model will be ugly and not readable. The other concern you may have is - performance of the generated code.

My take is use the modelling tool for what they are - as a vehicle for conveying design and architecture. And may be for generating the skeleton code.

Guilherme Vieira, Consultant: Actually MDD represents the current generation of best practices and tools to delivery a fully IT solution.

The team should use Model Driven Development since the beginning when the primary target is to identify and track down the needs and potential issues.

A domain model(using UML) must be build by businesses analysts and users/-customers in a collaborative way on the early stages of the whole process focusing in delivering a strong and well done artefact to guide the team to build (from ground up sometimes) a solution which will achieve all the expectations.

After that model construction phase the team will be able to start the development stage which can be or be not supported by automatic generated code. But as you may know must obey the given model rules.

In my humble opinion UML 2 already go through. Really nice by the way. But I guess it's time to another language take over and start a new era in the development process. I really think it is a open window! And that new standard could come up from anywhere.

There is a very large number of active technologies nowadays in Java community and they deserve a more dynamic and reliable "way" of MODEL and BUILD applications.

Just my 2 cents.