Mälardalen University Press Dissertations
No. 170

SOFTWARE AND HARDWARE MODELS IN COMPONENT-BASED DEVELOPMENT OF EMBEDDED SYSTEMS

Luka Lednicki

2015

University of Zagreb
Faculty of Electrical Engineering and Computing

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISBN 978-91-7485-180-9
ISSN 1651-4238
Printed by Arkitektkopia, Västerås, Sweden

Mälardalen University Press Dissertations
No. 170

SOFTWARE AND HARDWARE MODELS IN COMPONENT-BASED DEVELOPMENT OF EMBEDDED SYSTEMS

Luka Lednicki

Akademisk avhandling

som för avläggande av teknologie doktorsexamen i datavetenskap vid Akademin för innovation, design och teknik kommer att offentligen försvaras tisdagen den 27 januari 2015, 13.00 i Gamma, Högskoleplan 1, Västerås.

Fakultetsopponent: Professor Martin Törngren, KTH Royal Institute of Technology

University of Zagreb
Faculty of Electrical Engineering and Computing

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

Abstract

As modern embedded systems grow in complexity component-based development is an increasingly attractive approach to make the development of such systems simpler and less error prone. In this approach software systems are built by composing them out of prefabricated software components. One of the challenges for applying component-based development to embedded systems is the tight coupling between the software and the hardware platform. To take full advantage of the component-based approach in the embedded domain, the development process has to provide support for describing and handling this coupling.

The goal of this thesis is to provide advancements in development of embedded component-based systems by using a combination of software and hardware models. To achieve the overall research goal, three different aspects are investigated: (i) how to provide support for integration of sensors and actuators in component-based development, (ii) how to utilize a combination of software and hardware models in development of distributed systems, and (iii) how to analyze extra-functional system properties using models of both software and hardware. The thesis goal is addressed by following contributions: (i) a component-based model which allows describing sensors and actuators, and how they are connected to the processing nodes and software components, (ii) a method for automatic synthesis of code for communication with sensors and actuators, (iii) a framework for automatic generation of distributed communication in component-based models and (iv) a compositional model-level analysis of timing and processing node utilization for component-based applications. These contributions are evaluated in separation, by applying prototype tools to either example systems, case-studies, or test scenarios.

ISBN 978-91-7485-180-9
ISSN 1651-4238

University of Zagreb

MÄLARDALEN UNIVERSITY SWEDEN

This thesis is presented in partial fulfillment of international dual doctoral degree at Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, and School of Innovation, Design and Engineering, Mälardalen University, Sweden.

Main advisors
Prof. Ivica Crnković, Mälardalen University, Sweden
Prof. Mario Žagar, University of Zagreb, Croatia

Co-advisor
Associate Prof. Jan Carlson, Mälardalen University, Sweden

Faculty examiner
Prof. Martin Törngren, KTH Royal Institute of Technology, Sweden

Examining committee
Associate Prof. Tomáš Bureš, Charles University in Prague, Czech Republic
Associate Prof. Željka Car, University of Zagreb, Croatia
Dr. Roland Weiss, ABB Corporate Research, Germany

To my family

Abstract

As modern embedded systems grow in complexity component-based development is an increasingly attractive approach to make the development of such systems simpler and less error prone. In this approach software systems are built by composing them out of prefabricated software components. One of the challenges for applying component-based development to embedded systems is the tight coupling between the software and the hardware platform. To take full advantage of the component-based approach in the embedded domain, the development process has to provide support for describing and handling this coupling.

The goal of this thesis is to provide advancements in development of embedded component-based systems by using a combination of software and hardware models. To achieve the overall research goal, three different aspects are investigated: (i) how to provide support for integration of sensors and actuators in component-based development, (ii) how to utilize a combination of software and hardware models in development of distributed systems, and (iii) how to analyze extra-functional system properties using models of both software and hardware. The thesis goal is addressed by following contributions: (i) a component-based model which allows describing sensors and actuators, and how they are connected to the processing nodes and software components, (ii) a method for automatic synthesis of code for communication with sensors and actuators, (iii) a framework for automatic generation of distributed communication in component-based models and (iv) a compositional model-level analysis of timing and processing node utilization for component-based applications. These contributions are evaluated in separation, by applying prototype tools to either example systems, case-studies, or test scenarios.

Prošireni sažetak

Programski i sklopovski modeli u razvoju ugradbenih sustava utemeljenih na programskim komponentama

Povećanjem složenost modernih ugradbenih sustava u njihovom razvoju se sve češće pokušava primijeniti pristup temeljen na programskim komponentama. Koristeći ovaj pristup sustavi se izrađuju sastavljanjem programskih komponenata – već postojećih elemenata namijenjenih višestrukom korištenju. Ovakav način razvoja može, između ostalog, znatno skratiti vrijeme izrade sustava, smanjiti količinu grešaka u sustavu te učiniti sustave predvidljivijima. Jedan od aspekata ključnih u razvoju ugradbenih sustava je visok stupanj povezanosti programske podrške i sklopovlja. Iako postoje mnoge metode koje omogućuju modeliranje sklopovlja i olakšavaju razvoj programske podrške specifične za sklopovlje, one su rijetko integrirane s metodama razvoja temeljenim na komponentama.

Cilj ovog rada je unaprijediti proces razvoja ugradbenih sustava temeljen na programskim komponentama koristeći kombinaciju programskih i sklopovskih modela. U okviru rada istražena su tri aspekta razvoja: (i) kako pružiti podršku za integraciju senzora i aktuatora u razvoju temeljenom na programskim komponentama, (ii) kako unaprijediti razvoj raspodjeljenih sustava koristeći modele programske podrške i modele sklopovlja te (iii) kako analizirati svojstva sustava upotrebom spomenutih modela. Cilj rada ostvaren je sljedećim doprinosima:

- **Doprinos 1** – Model ugradbenog sustava temeljen na programskim komponentama koji uključuje senzore i aktuatore,
- **Doprinos 2** – Metoda sinteze programskog koda iz predloženog komponentnog modela,

- **Doprinos 3** – Metoda automatskog generiranja komunikacijskih komponenata u modelima raspodjeljenih ugradbenih sustava,
- **Doprinos 4** – Metoda kompozicijske analize vremenskih karakteristika i opterećenja procesnih čvorova u sustavima temeljenim na programskim komponentama.

Doprinos 1 proširuje postojeći komponentni model ProCom mogućnošću opisivanja senzora i aktuatora. Osim samog opisa senzora i aktuatora, novi model omogućuje i specificiranje načina na koji su ovi uređaji spojeni na procesne čvorove sustava. Postojeće programske komponente modela ProCom proširene su opisom ovisnosti o senzorima i aktuatorima. Veze između programske podrške i sklopovlja opisuju se pridruživanjem elemenata modela sklopovlja ovisnostima programskih komponenata. Predloženi model promiče ponovno korištenje jednom razvijenih elemenata jasnim razdvajanjem onih elemenata koji su neovisni o kontekstu od elemenata koji su specifični za pojedini sustav. U svrhu evaluacije model je implementiran u okviru prototipnog alata i primjenjen na realistični primjer, čime je pokazano da je pristup prikladan za modeliranje stvarnih sustava.

U okviru doprinosa 2 razvijena je metoda koja na temelju modela predloženog u doprinosu 1 automatski stvara programski kôd za komunikaciju programskih komponenata sa senzorima i aktuatorima. Metoda definira dvije grupe elemenata programskog koda: *ulazne elemente* i *izlazne elemente*. Ulazni elementi podobni su za višestruko korištenje te se pridružuju elementima modela koji su neovisni o kontekstu. Izlazni elementi koda stvaraju se automatski, na temelju elemenata modela specifičnih za sustav. Dobivena kombinacija ulaznih i izlaznih elemenata pruža potpunu komunikaciju između programskih komponenata i sklopovlja. Doprinos je evaluiran primjenom prototipnog alata na realistični primjer. Rezultati evaluacije pokazali su da generirani kôd pruža točnu implementaciju komunikacije. Evaluacija je također obuhvatila mjerenje povećanja vremena izvođenja i zauzeća memorije dobivenog kôda.

Doprinos 3 čini razvojni okvir koji omogućuje automatsko stvaranje komunikacije između čvorova raspodijeljenih aplikacija temeljenih na programskim komponentama. Komunikacija se implementira dodavanjem programskih komponenata u modele aplikacije specifične za pojedine čvorove sustava. Ove komponente stvaraju se na temelju modela aplikacije neovisnog o sklopovlju sustava i modela koji opisuje sklopovlje.

Razvojni okvir podijeljen je u module, međusobno povezane samo unaprijed definiranim sučeljima, čime se olakšavaju prilagođavanje metode različitim komponentnim modelima i nadogradnja okvira novim mogućnostima. Razvojni okvir je primijenjen na normu IEC 61499 te je implementiran prototipni alat. Evaluacija doprinosa izvršena je primjenom razvojnog okvira na prošireni primjer i na dvije studije slučaja. Modeli dobiveni automatskim stvaranjem bili su u skladu s očekivanim rezultatima, dok je dobivena komunikacija ispravno implementirala funkcionalnost sustava.

Doprinos 4 ovoga rada je metoda analize svojstava sustava izrađenih normom IEC 61499. Predložena metoda analize podjeljena je na dva dijela: analizu najdužeg vremena izvođenja i analizu opterećenja procesnih čvorova. Izračun najdužeg vremena izvođenja dijelova aplikacije temelji se na konceptu kompozicije komponenata. Analiza opterećenja procesnih čvorova omogućena je proširenjem analize najdužeg vremena izvođenja, u isto vrijeme uzimajući u obzir i modele sklopovlja. Za obje metode analize implementirani su prototipni alati. Isti alati su upotrijebljeni u svrhu evaluacije, primjenjujući ih na primjere preuzete iz dva postojeća alata za razvoj sustava pomoću norme IEC 61499. Dijelovi analize također su evaluirani primjenom prototipnog alata na skup testova. Evaluacija je pokazala da je predložena analiza primijenjiva na stvarne sustave te da algoritmi analize imaju visoku učinkovitost. Rezultati testova pokazali su ispravnost pojedinih dijelova analize.

Acknowledgements

I would like to start the acknowledgements with two persons who gave me the opportunity to be a PhD student, my advisors Mario Žagar and Ivica Crnković. Thank you for guiding me, pushing me forward, and providing both professional and personal support whenever I needed it. An equal amount of thanks also goes to my co-advisor Jan Carlson. All discussions, advice and detailed comments have certainly made this a better thesis, and made me a better researcher.

During these years of doctoral studies I have worked with many people from both Croatian and Swedish universities. With your willingness to help, friendship, and remarkable coffee-drinking skills you have made the university more than just a work place. Thank you for that! I will not try to list you all here, you know who you are!

To my parents Blanka and Damir, and sister Iva, thank you for providing endless support and encouraging me to achieve more through all my life. Without you I surely wouldn't be here, writing acknowledgments for a doctoral dissertation.

Lastly, Anna, you were by my side through all these years, you supported me when I needed it, and you made my life more interesting and fun than I could have imagined! Thank you!

Luka Lednicki
Västerås, December 2014

This work was partly supported by the Unity Through Knowledge Found through project project DICES, Swedish Foundation for Strategic Research via research centre PROGRESS and through project RALF3, and ABB Software Research Grant Program through project ASSIST.

Contents

Chapter 1

Introduction

Almost all modern technology, from factories and vehicles, to consumer electronics and household appliances, is in some level supported by embedded computer systems. The increased usage of embedded systems has resulted in rapid growth of their complexity. An example of this can be seen in the automotive industry. Premium vehicles often have a hardware platform consisting of more than 100 processing units connected with multiple communication networks, running several thousands of software functions, and interact with the environment using numerous sensors and actuators [64]. Development of embedded systems is also complicated by the specifics of the embedded domain. The context they are used in often requires them to not only provide correct functionality, but to also deliver the functionality in a specific time. These systems also usually execute on platforms with limited resources, e.g. low processing power and memory capacity. As a result of these factors, it is getting harder to develop embedded software and ensure that the implementation satisfies its requirements. One of the possibilities for alleviating development of such complex systems is by applying model driven [7] and component-based [1, 17, 60] approaches for their development.

Model driven development advocates building systems by modeling them, and providing implementation by transforming models to executable code. This approach provides multiple benefits. The models provide a view of a system which is more abstract compared to executable code. Thus, much of the complexity of implementation can be hidden, and systems can be easier to develop and understand.

The models used to create a system can also be used to analyze its various properties. Because it is applied to an abstract view of a system, model-level analysis can provide high efficiency, and is applicable in early stages of development, when the full implementation is not yet available. The ability to detect potential problems early can reduce the risk of costly redesign late in the development. A negative side of analysis on such abstract level is a possible decrease in the accuracy of results.

A system can often be described by more than one model. As an example, it is common to describe system functionality in a platform-independent manner, and provide the platform-specific details in a separate model. This allows each model to more clearly represent a specific aspect of the overall system. Also, besides just modeling software, some approaches include the ability to use models for describing the hardware platform that the software can be deployed to. However, having multiple coexisting models introduces the need of keeping them synchronized, and utilizing a combination of multiple models is not always trivial.

Model driven development is often combined with aforementioned component-based software engineering. In component-based software engineering systems are built by composing them out of software components – units of software which conform in their syntax and semantics to a component model.

The main characteristic of software components is that they are developed with reuse in mind. Component should implement a functionality which is independent to the context of a specific system, encapsulate that functionality, and provided it to the rest of the system only through a well-defined interface. The interface should also explicitly express all dependencies of a component, i.e. what functionality the component requires from the rest of the system. The benefit of component communicating only through a well defined interface is twofold. First, by having all components implement an interface conforming to same syntax and semantics it can be assured that different components can be composed together and exchange information. Second, the system developer can use components in a *black-box* manner, meaning that during component composition only components interfaces are used, while the complexity of the implementation stays hidden.

One problem which arises when applying component-based development in the embedded system domain is the tight coupling of software and hardware. The limited processing and memory resources often prevent use of extensive hardware abstraction layers usually present in the desktop domain. Because of this, the code implementing software components can become highly dependent on the hardware it communicates with. If not addressed properly, such dependencies can limit the potential of component reuse.

Besides promoting reuse, the previously mentioned encapsulation of component functionality also facilitates prediction of system properties. As the implementation of a component is isolated from the rest of the system, and ideally all components dependencies are explicitly expressed by the component interface, it becomes easier to predict properties of a single component. Then, in the same way that system functionality is built by composing functionality of components, the properties of a system can potentially be composed from properties of individual components. The increased predictability is one of the main benefits the component based approach can introduce to the development of embedded systems. It is, however, not trivial to define context-independent description of component properties and provide methods for composition of such properties, especially when considering the aforementioned tight coupling of software and hardware.

In this thesis we investigate how current state-of-the-art of component-based development process can be advanced by leveraging software and hardware models of component-based systems. In the rest of this section we describe the research questions that have guided the research, present the main contributions of the thesis, list the publications which have been used as the basis for the thesis, describe the methodology used during research and in the end give an outline for the rest of the thesis.

1.1 Research questions

The overall goal of this research is to provide advancements in development of embedded component-based software systems by leveraging a combination of software and hardware models. To achieve this goal, we have investigated three aspects of component-based development for embedded systems consisting of software and hardware: (i) support for integrating sensors and actuators while modeling and deploying component-based systems, (ii) development of distributed systems and (iii) analysis of extra-functional system properties. The conducted research has been guided by three research questions, which are presented below.

Interaction with the environment using IO devices such as sensors and actuators is one of the key characteristics of embedded systems. The two most common ways for handling this interaction in component-based development are to either make it a part of software component code, or let it be handled outside of the component model. Having the communication with IO devices encapsulated in software components hides the dependencies of components on the devices, and makes such components highly dependent on a specific hardware configuration. This breaks the principle of components having all dependencies expressed on the interface level and reduces the potential of component reuse. On the other hand, completely removing sensors and actuators from component models prevents the component-based design to cover the whole system and hides the effects of IO devices during system analysis. Such approach can thus reduce applicability of the component-based approach and predictability of systems. Therefore, we have defined the following research question:

Research Question 1: *How can we improve the support for integration of sensors and actuators in component-based development for embedded systems, so that dependencies to these devices are more easily manageable?*

Some component models provide support for modeling distributed applications by providing two complementary sets of models: platform-independent models which abstract away from the details of communication between platform nodes, and platform-specific models which contain components that implement the distributed communication. Although the ability to model applications on these two levels can help coping with the complexity of distributed systems, manually keeping the two models synchronized is a time consuming and error-prone activity. Lack of established methods for keeping these models up-to-date in when using a component based approach led us to the second research question:

Research Question 2: *How can we enhance development of distributed component-based systems in order to reduce the effort of synchronizing platform-independent and platform-specific models?*

Embedded systems often have to satisfy domain-specific requirements such as real-time constraints. A common way of verifying if these requirements are met is through analysis. The fact that component-based systems are often built using abstract models provides a possibility for analysis to also be performed using these models. However, exploiting the full potential of model-level analysis in component-based systems is not trivial. This led us to define the following final research question:

Research Question 3: *How can we utilize software and platform models to efficiently analyze extra-functional properties of component-based systems in early stages of development?*

1.2 Contributions

We have addressed the research questions presented in the previous section by the following four contributions.

Research contribution 1 (RC1): *Component-based embedded system model with integrated sensors and actuators.*

We have extended the ProCom component model [53] with an ability to describe IO devices (i.e. sensors and actuators) that are part of the hardware platform, and means to specify how these devices are connected to the processing nodes of the platform. The new model lets software components state their dependencies on specific types of IO devices on the level of component interface, and allows these dependencies to be propagated through component hierarchy. During the deployment phase, the dependencies of software components can be mapped to the available IO devices. The proposed approach promotes reuse by having context-independent model elements loosely bound and clearly separated from the system-specific ones. A part of the contribution is also a prototype modeling tool implementing the approach. The contribution was evaluated by applying it to a realistic example, proving that the models adequately describe the system.

Research contribution 2 (RC2): *Code synthesis method based on the proposed component model.*

We have developed a method that provides automatic generation of platform specific code for transfer of data between software components and IO devices (i.e. sensors and actuators). The generation leverages the platform model which describes IO devices, processing nodes, how the two are connected, and how the software components communicate with IO devices. The method defines how to specify reusable code segments for context-independent model elements, and generates glue-code connecting these segments into a system-specific solution. The generation method is implemented for the ProCom component model and supported by a prototype tool. The synthesis method has been evaluated in combination with RC1 by applying the prototype tool to a realistic example. The results of the evaluation proved that the synthesized code correctly implements system functionality. A part of the evaluation was also measuring the overhead of the generated code in terms of execution time and memory footprint.

Research contribution 3 (RC3): *Method for automatic generation of communication components in distributed embedded systems models.*

We have defined a framework which supports automatic generation of communication between nodes of distributed component-based applications. The generation uses platform-independent software models and models of the platform to insert and configure components which implement the required communication. The framework introduces adaptability and extensibility by separating generation into different loosely bound modules connected only through well-defined interfaces. The method has been applied to the IEC 61499 standard and implemented in a prototype tool. The contribution has been evaluated by applying it to an extended example, and two case-study systems. The evaluation showed that the generated models were in line with the expected results, and that the generated communication correctly implements the functionality of the case-study systems.

Research contribution 4 (RC4): *Method for compositional timing and utilization analysis of component-based systems.*

We have developed a novel method for timing analysis of component-based embedded software systems built using the IEC 61499 standard. The analysis method relies on the concepts of component composition and hierarchy to provide efficient calculation of worst-case execution time for composite components and applications. By extending the timing analysis to use a combination of software and platform models, we have enabled analysis of processing resource utilization. Both timing and utilization analysis have been implemented in a prototype tool. The prototype tool was used to evaluate the analysis using a set of example models taken from two IEC 61499 development tools, and a set of test scenarios. The evaluation using example models proved that the analysis is applicable to realistic systems, and that the performance of the analysis algorithms is high. Applying the analysis on the set of test scenarios has validated the correct behavior of parts of the analysis method.

1.3 Research methodology

The research method applied to the work presented in this thesis generally aligns to the engineering version of the scientific method given by Basili [5]: *"observe existing solutions, propose better solutions, build/develop, measure and analyze, and repeat the process until no more improvements"*. The overall research has been conducted according to following steps:

1. Investigation of state-of-the-art and state-of-the-practice in the field of component-based software engineering for embedded systems.

2. Definition of a concrete research problem, followed by an in-depth literature review and definition of a research goal.

3. An iterative process of: (i) development of a theoretical research result, (ii) implementation of a prototype tool, (iii) evaluation of the method using the prototype tool.

4. Validation of the method using a case-study or a set of tests.

As we have addressed multiple concrete problems, all the steps have been performed for each problem separately.

The research started with a survey of component-based models developed for use in the embedded system domain [19]. As a part of the survey, we have identified a lack of support for expressing the interaction with the hardware platform in such development approaches, with the automatic generation of hardware-specific code as a possibility of improvement. The research continued with a more thorough analysis of existing approaches for modeling hardware. Based on this, we have defined a method to model sensors and actuators in component models for embedded systems. The modeling method was then used to provide the method of automatic generation of sensor- and actuator-specific code.

The part of the research that addresses system analysis started with a project which included transfer of research applied conducted in scope of the ProCom component-model to the more industrial setting of the

IEC 61499 standard [15]. Based on the existing model-level analysis for ProCom [13] and an investigation of other existing methods for both model- and code-level timing analysis we proposed a novel method for timing analysis which could take advantage of the specifics of component-based development approach. The results of this research were then combined with the results of the research on hardware modeling, and the new analysis method was extended to take into account models of both software and hardware. As we have identified the lack of support analysis of cyclic execution paths to be a problem in current model-level analysis methods, we have also extended the timing analysis with an ability to analyze such constructs.

During the research on model-level analysis, we also identified a possibility of enhancing the analysis method by performing the analysis on platform-specific models which include inter-node communication components. As we did not find a component-based development framework which would allow generating such model from platform-independent models and model of the platform, we extended our research to provide such a method.

For each new method we defined, we developed the theoretical contributions in parallel with a prototype tool. The tools were used to evaluate the applicability of the methods, and iteratively updating the methods using the insights gained by the evaluation.

The presented work is validated through persuasion, examples and evaluation, as defined by Shaw [54]. When possible, a *slice of life* example is used, rather than a *toy example.* To some extent, the developed tools are been applied to the examples to provide a more detailed information about the performance of proposed methods, rather than just test their applicability.

1.4 Publications

This section first presents the main publications contributing to this thesis, and then lists the rest of the related publications.

1.4.1 Main contributing publications

The publications presented in this section constitute the basis for this doctoral thesis. For all these publications I am the main author of both contributions and text, while other coauthors contributed with smaller amounts of text, and valuable discussions and reviews.

- **Paper A:** *Adding Support for Hardware Devices to Component Models for Embedded Systems*, Luka Lednicki, Juraj Feljan, Jan Carlson, Mario Žagar, The Sixth International Conference on Software Engineering Advances (ICSEA), 2011.

  This paper introduces a possibility of modeling sensors and actuators, describing how they are connected with processing nodes of the platform, and specifying their interaction with software components, providing contribution RC1. The work presented in this paper is the basis for a part of Chapter 3 where the approach for modeling sensors and actuators is described.

- **Paper B:** *Automatic Synthesis of Hardware-specific Code in Component-based Embedded Systems*, Luka Lednicki, Ivica Crnković, Mario Žagar, The Seventh International Conference on Software Engineering Advances, 2012.

  In this paper the results of Paper A have been used to define a method for synthesis of code that connects software components to sensors and actuators. The method specifies how to define reusable code elements and allows automatic generation of glue-code between these elements, resulting in contribution RC2. The part of Chapter 3 that concerns code synthesis is based on this paper.

- **Paper C:** *A Framework for Generation of Inter-node Communication in Component-based Distributed Embedded Systems*, Luka Lednicki, Jan Carlson, IEEE International Conference on Emerging Technology and Factory Automation, 2014.

  This paper presents an extensible framework which allows automatic generation of communication between nodes of distributed systems by adding communication components to platform-specific system models, providing contribution RC3. Paper C is used as the basis for Chapter 4.

- **Paper D:** *Model Level Worst-case Execution Time Analysis for IEC 61499*, Luka Lednicki, Jan Carlson, Kristian Sandström, The 16th International ACM Sigsoft Symposium on Component-based Software Engineering, 2013.

  The contribution of this paper is a novel method for compositional worst-case execution time analysis of component-based software, giving the foundation for the thesis contribution RC4. This paper is used as the basis for a part of Chapter 5, where the model level timing analysis for IEC 61499 is introduced.

- **Paper E:** *Device Utilization Analysis for IEC 61499 Systems in Early Stages of Development*, Luka Lednicki, Jan Carlson, Kristian Sandström, IEEE International Conference on Emerging Technology and Factory Automation, 2013.

  In this paper we extend the analysis method described in Paper C to utilize models of hardware platform to provide a novel method for analyzing device utilization, adding to contribution RC4. The paper is used as the basis for the part of Chapter 5 describing processing node utilization analysis.

- **Paper F:** *Handling Cyclic Execution Paths in Timing Analysis of Component-based Software*, Luka Lednicki, Jan Carlson, The 40th Euromicro Conference on Software Engineering and Advanced Applications, 2014. (short paper)

  This paper further extends the analysis method presented in Paper C with the ability to analyze software models of component-based applications which contain cyclic execution paths, further extending contribution RC4. Paper F is the basis for the analysis of cyclic paths in IEC 61499 given in Chapter 5.

All of the main contributing papers are used as a partly basis for the background presented in Chapter 2 and the related work described in Chapter 6.

1.4.2 Other related publications

Peer reviewed publications

- *15 Years of CBSE Symposium: Impact on the Research Community*, Josip Maras, Luka Lednicki, Ivica Crnković, Proceedings of the 15th ACM SIGSOFT Symposium on Component-based Software Engineering, 2012.

- *Towards Automatic Synthesis of Hardware-specific Code in Component-based Embedded Systems*, Luka Lednicki, Ivica Crnković, Mario Žagar, Proceedings of the 38th Euromicro Conference on Software Engineering and Advanced Applications, 2012. (short paper)

- *PRIDE – an Environment for Component-based Development of Distributed Real-time Embedded Systems*, Etienne Borde, Jan Carlson, Juraj Feljan, Luka Lednicki, Thomas Leveque, Josip Maras, Ana Petričić, Séverine Sentilles, 9th Working IEEE/IFIP Conference on Software Architecture, 2011.

- *Support for Hardware Devices in Component Models for Embedded Systems*, Luka Lednicki, International Doctoral Symposium on Software Engineering and Advanced Applications, 2011.

- *DICES: Distributed Component-based Embedded Software Systems*, Mario Žagar, Ivica Crnković, Darko Stipaničev, Maja Štula, Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, Annual of the Croatian Academy of Engineering, 2010/2011.

- *A Component-Based Technology for Hardware and Software Components*, Luka Lednicki, Ana Petričić, Mario Žagar, 35th Euromicro Conference on Software Engineering and Advanced Applications, 2009. (short paper)

- *Using UML for Domain-specific Component Models*, Ana Petričić, Luka Lednicki, Ivica Crnković, Fourteenth International Workshop on Component-oriented Programming, 2009.

- *Uniform Treatment of Hardware and Software Components*, Luka Lednicki, Jan Carlson, Mario Žagar, 8th Conference on Software Engineering Research and Practice in Sweden, 2008.

Technical reports

- *Feasibility of Migrating Analysis and Synthesis Mechanisms from ProCom to IEC 61499*, Jan Carlson, Luka Lednicki, MRTC report, 2012.

- *Classification and Survey of Component Models*, Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, Ivica Crnković, MRTC report, 2009.

1.5 Thesis outline

In this section we give an outline to the rest of the thesis chapters.

Chapter 2 – Background
The chapter gives an overview of the existing concepts and research needed to understand the contributions presented in the thesis. The overview consists of descriptions of the ProCom component model and the IEC 61499 standard.

Chapter 3 – Sensors and actuators in component-based development
This chapter first discusses effects that sensors and actuators have on component-based development process, and then introduces an extension to the ProCom component model which allows modeling of sensors and actuators, and in the end describes the method for synthesis of sensor- and actuator-specific code. The work presented in this chapter is based on Paper A [43] and Paper B [41] of the main contributing publications of the thesis, and [42]. This chapter updates the modeling approach described in the publications, introduces a common example model for the modeling and synthesis contribution, and extends the evaluation of the approach.

Chapter 4 – Automatic generation of inter-node communication
We describe a framework for automatic generation of communication in distributed component-based applications. In the chapter we first introduce a model which captures communication requirements of applications. After that we describe the process of automatic generation of distributed communication. In the end we exemplify the generation on a simple model. The contributions of this chapter are based on Paper C [36] of the main contributing publications. Apart from improved descriptions, this chapter does not add to the work presented in the Paper C.

Chapter 5 – Model-level timing and utilization analysis
In this chapter we present a novel model-level analysis method for analysis of component-based systems. First, we introduce a method for

compositional analysis of worst-case execution time of component-based software applications. We then describe how the compositional analysis can be applied to applications which contain cyclic execution paths. We complete the chapter with a method for analysis of processing node utilization which is based on the compositional worst-case execution time analysis. This chapter is based on the main contributing publications Paper D [40], Paper E [39] and Paper F [37]. The descriptions and formal definitions presented in the publications are improved, and new formal definitions are added to the previous work.

Chapter 6 – Related work

We provide an overview if existing research that is related to the work presented in this thesis.

Chapter 7 – Conclusion

This chapter concludes the thesis.

Chapter 2

Background

This chapter describes the ProCom component model and the IEC 61499 standard. The presentation of ProCom is kept on a higher level, sufficient to understand the work presented in Chapter 3, and mainly describes how some of the common component-based development concepts are implemented by this component model. The IEC 61499 standard is described on a more detailed level, as the contributions in Chapter 4 and Chapter 5 are more closely connected to the details of the standard.

2.1 ProCom

ProCom [10, 11, 53] is a component model for development of safety-critical distributed embedded real-time systems. One of the main aims of ProCom is to provide component-based approach that covers the whole development process, including design, analysis and deployment of systems. Development of ProCom systems is supported by the open-source PRIDE tool [8]. The rest of this section gives an overview of various parts of the ProCom approach.

2.1.1 Components

ProCom has a strong notion of components, which are rich and reusable design-time entities. Each component implements a specific functionality, encapsulates it, and provides it only through a well-defined interface.

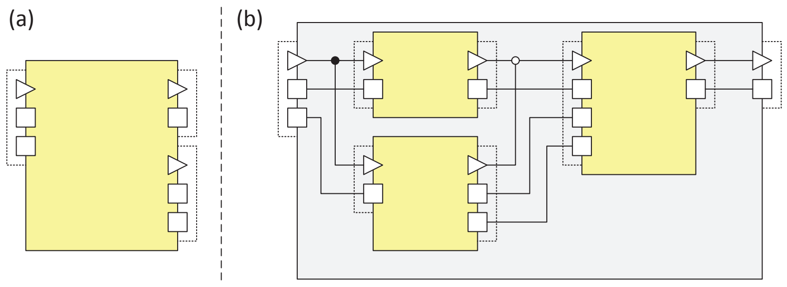


Figure 2.1: (a) ProSave component interface. The triangles and squares represent trigger and data ports, respectively. While the input ports are located on the left border of the components, the output ports are located on the right side. Port groups are denoted by dashed lines encompassing the ports. (b) An example of a composite ProSave component.

Besides their functionality, components can also store various other elements, for example behavior models or property values. Syntax and semantics of components is defined on the interface level. This allows treatment of components in a black-box manner, promoting their reuse. Although ProCom provides two modeling levels, ProSave and ProSys, each with different levels of component granularity, for the purpose of this thesis we will only concentrate on the lower, ProSave layer.

On the ProSave level, systems are modeled by connecting components in a pipe-and-filter manner, with clear distinction between the data and control flow. On the interface level, ProSave components[1] consist of one input port group, and zero or more output port group. Each group contains one trigger port and an arbitrary number of data ports. The trigger ports of a component are used to activate the execution of the component (input trigger) or send activation signals to other components (output trigger), while the data ports are used to send or receive data. The graphical representation of a ProSave component on the interface level is given in Figure 2.1 (a).

[1]For the purpose of this thesis, the notion of ProSave *service*, and the ability of components to provide multiple input port groups, will be disregarded.

2.1.2 Component types

Based on how they implement their functionality, ProSave distinguishes two types of components: primitive components and composite components. The implementation of primitive components is defined by code, with each component defining an entry function in C language which will be executed when the component is triggered. Composite components implement their functionality using interconnected subcomponents. In this way, functionality provided by a composition of components can be encapsulated and presented in a black-box manner, as a component, and reused on higher levels of hierarchy. Figure 2.1 (b) shows an example of the interface and implementation of a ProSave composite.

2.1.3 Semantics

All types of ProSave components adhere to same semantics, defined on the level of the component interface [59,66]. The components are passive, meaning that they never start execution unless activated by a signal at their input trigger port. Once triggered for execution, components follow a read-execute-write pattern. First, the values at the input ports of the interface are read and transferred to the component's internal implementation. Then, the functionality of the component is executed. During the execution, the output data ports are updated with new values, and activation signals are sent through output trigger ports. Output of the data and the triggering signal for each group is performed as an atomic operation. When all output triggers of a component have been activated, the component stops its execution and remains inactive until the next activation of its input trigger port.

2.1.4 Platform modeling and deployment

ProCom supports modeling of execution platforms and how software systems are deployed to those platforms [14]. Models of hardware platforms consist of physical nodes and network connections between them. Physical nodes represent concrete hardware processing units. Deployment of software systems to the platform is done by mapping the software components to the physical nodes[2].

[2]In the work presented in this thesis, the ProCom concept of *virtual nodes* is disregarded.

2.1.5 Analysis

One of the main aspects of ProCom is aim to support analysis of system properties. The support for attaching attributes or models, which can be used for analysis, is given by an extensive framework for specification of extra-functional properties [52]. Besides providing support for attaching extra-functional properties to components, various analysis techniques have been developed in the context of ProCom, for example analysis of timing properties [13, 44], and resource usage analysis [51, 65].

2.2 IEC 61499

The IEC 61499 standard [28, 61, 67, 70] is proposed as a successor of the IEC 61131-3 standard [27], which is widely used in industry, especially for development of industrial automation systems. The new standard addresses some of the problems of IEC 61131-3, mainly focusing on enhancing reusability, configurability, interoperability, reconfiguration and support for distributed systems. Development using IEC 61499 is supported by several tools, for example 4DIAC [56], Function Block Development Kit [24], nxtStudio [47] and ISaGRAF [26]. The details of the component-based approach of IEC 61499 are described in the following sections.

2.2.1 Components

The software components of the IEC 61499 standard are called *function blocks*. As is common in component-based approaches, the functionality of a function block is presented by the *function block interface*, which hides the details of the implementation. Similarly to ProCom (described in Section 2.1), the IEC 61499 function block interface explicitly separates event and data inputs and outputs. Event inputs and outputs are used to specify the execution flow, but do not provide any means for exchanging data between function blocks. All data transfers are done by data inputs and outputs.

Relations between event and data ports can be described by *WITH qualifiers*. A WITH qualifier can be defined on either a combination of one input event port and a set of data inputs, or one output event port and a set of data outputs. Defining a WITH qualifier on an event input port and a set of data input ports describes which data inputs will

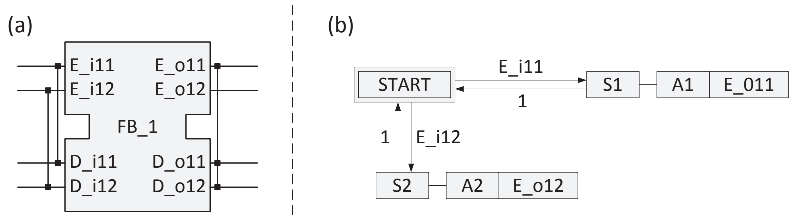


Figure 2.2: (a) An IEC 61499 function block interface. (b) A basic function block ECC.

be sampled together with the event port. A WITH qualifier combining an event output port with a set of data output ports shows which data outputs will be updated with new values together with an output at the event output.

Example: Figure 2.2 (a) shows an example of a function block interface, defined for function block *FB_1*. The figure shows a function block with input event ports *E_i11* and *E_i12*, output event ports *E_o11* and *E_o12*, data inputs *D_i11* and *D_i12*, and data outputs *D_o11* and *D_o12*. The WITH qualifiers are represented by connecting the ports with vertical lines, marking each port belonging to a WITH qualifier by a black rectangle. As an example, the WITH operator defined on the outputs of *FB* implies that when an event is generated on *E_o11*, the values on ports *D_o11* and *D_o12* will also be updated.

2.2.2 Component types

Considering their implementation, function blocks can be of three possible types: Basic function block, Service interface function block and Composite function block.

A *basic function block* (BFB) is implemented by means of an *Execution Control Chart* (ECC) and one or more algorithms. The ECC is an automaton consisting of states and guarded transitions. Each state can be associated with zero or more actions. An action can specify one or more algorithms which will be executed once the state is reached, and output event ports that will be activated. We differentiate between two types of states: stable states in which execution of the ECC stops until a new event arrives at an input port, and transitional states which do not require an event for ECC to move to another state. Execution of basic function blocks is strictly event driven – it can only start when an event is received at one of the input ports, and once the execution stops it will not continue until the next event arrives. One execution cycle of a function block is called a *run*. A single run can traverse more than one ECC state in case the ECC contains transitional states, and thus result in an arbitrary number of algorithm executions and output events.

Example: An example of a basic function block ECC is given in Figure 2.2 (b). It contains states *START*, *S1* and *S2*, with *START* being the initial state. The transition from *START* to *S1* is guarded by the input event *E_i11*. When this state is reached, algorithm *A1* is executed, and an event is generated at output event port *E_o11*. The transition to state *S2* is guarded by the input event *E_i12*, and it executes algorithm *A2* and outputs an event at *E_i12*. Both *S1* and *S2* are transitional states – there are transitions back to *START* that are not guarded by any event, depicted by a *1* as the transition condition.

Service interface function blocks (SIFB) are designed to be used as interfaces to external hardware or services. The functionality of this element is not specified by the standard, and although they can contain a sequence diagram describing their behavior, the functionality might not be fully documented. Unlike basic function blocks, the service interface function blocks can start their execution without the arrival of an input event (active execution).

A *composite function block* (CFB) has an implementation defined by a function block network (defined below), with additional connections between the ports of the enclosing interface and the ports of the function blocks in the network. As the composite function block can contain active service interface function blocks, composites can also be active, i.e. start their execution without receiving an input event.

A *function block network* (FBN) defines the internal structure of a composite function block or a whole application. A function block network consists of a set of function blocks of arbitrary types (BFB, SIFB or CFB) and connections between the ports of these function blocks. As a result of the separation of event and data ports, the flow of control and data are clearly distinguished.

Example: An example of a composite function block, named *FB_4*, can be seen in Figure 2.3. Its internal function block network contains three function blocks, *FB_1*, *FB_2*, and *FB_3*.

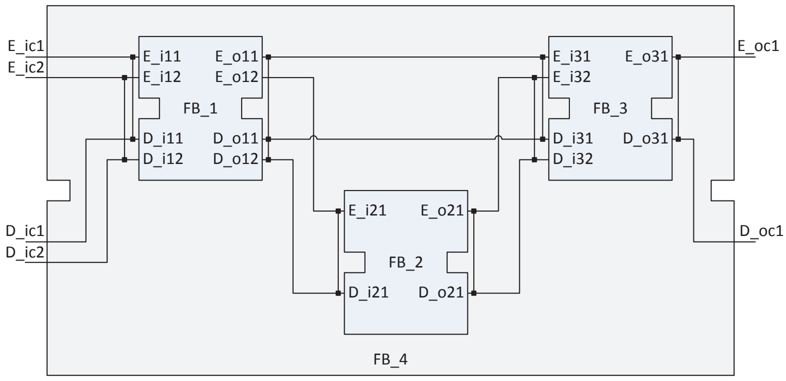


Figure 2.3: A composite function block with an internal function block network.

2.2.3 Semantics

Unlike the previously described ProCom component model, the semantics of IEC 61499 function blocks is not unambiguously defined on the level of component interface [57, 58]. While basic function blocks are strictly passive, starting their execution only by receiving signals to their input event ports, and following a read-execute-write pattern, service interface function blocks can also start their execution by means of an internal trigger. Both of these function block types, regardless of their triggering, are executed as atomic units: no other function block will interrupt their execution. As composites can contain both these types of function blocks, they can also be either passive or active components. The definition of composites also does not clearly define the atomicity of their execution. The standard does not specify if it should be allowed to interrupt execution of a composite function block by some function block outside of the composite. As a result, different IEC 61499 implementations provide different semantics for composites, and it is harder to argue about properties of function block compositions. Although not defined by the standard itself, a formal definition of execution semantics [63] has been published in the research community.

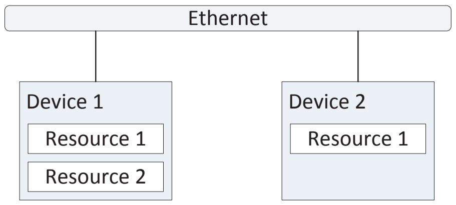


Figure 2.4: Example model of the platform in IEC 61499.

2.2.4 Platform modeling and deployment

In IEC 61499 the platform is represented by *devices*. A device is an independent physical entity capable of performing one or more specified functions. Each device contains one or more *resources*, which are functional units with independent control of operation.

Communication networks between devices are modeled by *network segments*. One device can be connected to more than one segment, and there is no limit to the number of devices that can be connected to a single network segment.

Example: Figure 2.4 shows an example of platform model consisting of two devices connected via an Ethernet network segment. *Device 1* has two resources, and *Device 2* with only one resource.

Applications are deployed to the platform by mapping its function blocks to the resources contained in the devices. Since all function blocks, including composites, are atomic units of deployment, each function block can be mapped to only one resource.

Each resource in a distributed IEC 61499 system contains a local model of the application, containing only the subset of the application's function blocks which are mapped to that particular resource. This local model can however contain additional function blocks which are not visible on the application level. It is a common practice to use this option to implement functionality that is specific to the current mapping of the application, such as adding inter-resource communication function blocks.

2.2.5 Analysis

Analysis was not one of the original main concerns of IEC 61499, and analysis of IEC 61499 models has not yet been fully explored [67,70]. One of the obstacles for performing analysis on the level of IEC 61499 models is also the previously mentioned ambiguity of execution semantics. Some of the analysis techniques developed for IEC 61499 are described as part of the related work, in Section 6.3.

The IEC 61499 standard allows attaching properties to modeling elements in form of attributes. The types of attributes are not defined in advance – they can hold any information about functional or extra-functional properties of the model elements. The standard also supports inheritance of attributes, for example function block instances inherit attributes from their respective function block types.

Chapter 3

Sensors and actuators in component-based development

One of the main characteristics of embedded systems is their interaction with the environment using sensors and actuators. The communication between software components and such IO devices can be as simple as writing a value to a hardware pin or a port, or as complex as invoking a service on a remote device. In all cases, this interaction introduces dependencies of software on the underlying hardware used to interact with the environment. The same interaction can also affect behavior and properties of software components, as, for example, properties like data acquisition time can vary for different types of sensors. Failure to adequately express this interaction can result in models that do not capture all system properties, and reduce the potential of component reuse, thus hindering the full utilization of the component-based approach in the embedded system domain.

In this chapter, we will first describe the effects that sensors and actuators can have on model-driven component-based development process, and then present our approach for modeling sensors and actuators in component-based systems and a method which allows automatic synthesis of code based on the proposed models. The chapter continues with description of a prototype tool which implements the presented methods and provides information about the evaluation, and ends with a summary. The work presented in this chapter is based on Paper A [43] and Paper B [41] of the main contributing publications of the thesis, and [42].

3.1 Effects of sensors and actuators on component-based development

When applying a model-driven component-based software engineering approach to embedded systems, the dependencies of software components on sensors and actuators, as well as communication between the software and hardware, impacts multiple aspects of the development process. In this section we will give an overview of some of them.

When *designing* a system, the developer specifies the system using models of (i) the software layer of the system, as a composition of software components, and (ii) the hardware platform layer, as a composition of nodes that the system will be deployed to. The former requires means to manage interaction with sensors and actuators in the software layer, i.e. to specify which components communicate with IO devices, and how. The latter calls for an ability to describe actual instances of sensors and actuators, and how they are connected to processing nodes of the platform. Additionally, if we want to accommodate to the component *reuse* concept, it must be ensured that the components dependent on sensors and actuators can be deployed to various hardware platforms.

During system *deployment* the components of the software layer are allocated to the elements of the platform layer that will execute them. In this phase, there is a need to identify the dependencies of the software components on sensors and actuators in order to ensure that the platform elements targeted for deployment satisfy these dependencies.

To guarantee that the system satisfies its requirements, the developer can perform various types of *analysis*. Such analysis can be used to check both functional and extra-functional properties of a system. For the results of the analysis to be valid, the effects of interaction of software components with sensors and actuators must be taken into account.

Finally, in the *synthesis* phase system models are used to generate executable code. In this phase, it must be ensured that the generated code reflects the specifics of the platform and correctly implements the communication with sensors and actuators.

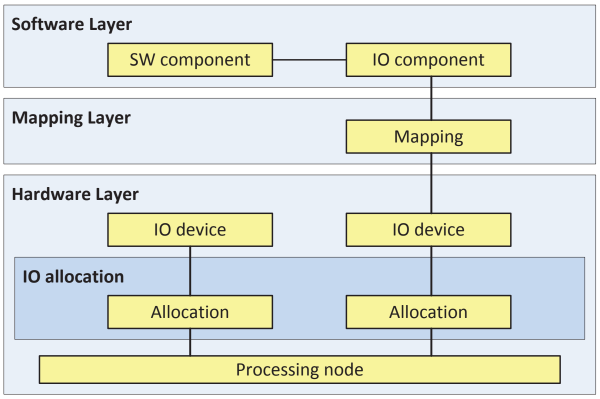


Figure 3.1: Overview of the layered model for describing IO devices in component based development.

3.2 Modeling sensors and actuators in component-based approaches

To address the problem of specifying the interaction of software components with sensors and actuators we have defined a model which allows incorporation of sensors and actuators into component models, and applied it to ProCom.

To limit the amount of coupling between software components and IO devices we have separated the model into three layers: *software layer*, *hardware layer* and *mapping layer*. With this separation the software and the hardware of the system can be described independently, making the descriptions suitable for reuse in different scenarios. When developing a concrete system, the independent software and hardware layers can be connected using the mapping layer. The same principle of loose binding that is introduced by the mapping layer is also used in the hardware layer, where connections between processing nodes and IO devices are defined by IO allocations. An abstract overview of these three layers is given in Figure 3.1.

The IO device modeling approach introduces a clear distinction between *types* and *instances* for both hardware and software entities. Types are entity definitions that are context-independent. They can be reused in different settings or stored in repositories for future use. To use a type in a concrete system, an instance of that type needs to be created. In this case instances are not copies of the entity, but rather representatives of the general entity in a specific context. For example, when describing an IO device, the description actually defines a type. When creating a system containing the IO device, a developer creates a new instance of that type. In some cases, instances can also refine properties of the type depending on the usage context.

The next three sections give a detailed description of the model layers, referencing the metamodel given in Figure 3.2. The metamodel is defined using Ecore, a language for describing metamodels provided by the Eclipse Modeling Framework [55].

3.2.1 Software layer

To enable describing interaction of component-based applications with IO devices, we have introduced a new type of component – *IO component*. This component type is derived from ordinary software component, providing the same interface and adhering to the same execution semantics. The difference between normal software components and IO components is in their internals. Device components do not provide an ability for developers to explicitly specify their realization, and do not contain any device specific code (e.g. code for actual communication with hardware). This is because they inherit their realization from IO devices once the two are mapped together (described in detail in Section 3.2.3).

To be able to use device components in composites, and still treat composites in a black-box manner, components have to specify their device dependencies on the interface level. Therefore we have also extended the component interface with a device dependency list, containing references to device components. In case of device components this list always contains only one entry, referencing the device component itself. Primitive components which are implemented by code always have an empty IO dependency list. For composite components, the device dependency list matches the combined dependencies of all of its subcomponents. It should be noted that primitive and composite types of components are not covered by the meta-model in Figure 3.2.

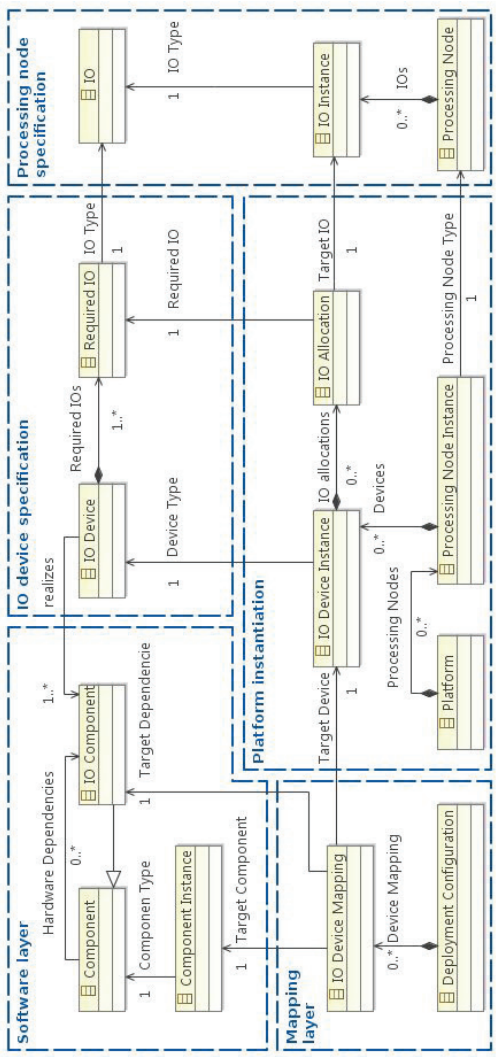


Figure 3.2: Ecore metamodel that defines the model for including sensors and actuators in component models.

3.2.2 Hardware layer

The hardware layer allows describing processing nodes that runnable code can be deployed to, IO devices such as sensors and actuators, and platforms which consist of instances of processing nodes and IO devices. As reuse is one of the main characteristics of component-based approach, the hardware layer model is also defined in a manner which promotes reuse. It consists of three separate logical parts: *processing node specification*, *IO device specification* and *platform instantiation*. In the following sections we will go into the details of these three.

Processing node specification

Processing nodes describe hardware which can be a target for deployment of executable code, for example microcontroller or ECUs. They are reusable model entities as they only describe types of nodes, and do not contain any information about how nodes are used or configured in a particular system.

Processing nodes define a list of inputs and outputs they provide using *IO instances*. Each IO instance references an IO type (e.g. one-bit digital I/O, serial communication port, analog input, etc.).

Processing nodes and their inputs and outputs can also be characterized by extra-functional properties such as their processing power, available memory, behavioral models, delays for input and output functions and other similar attributes.

IO device specification

IO Devices represent sensors and actuators that can be connected to processing nodes in order to interact with the environment. Like processing nodes, device definitions are reusable and define a type of sensor or actuator, rather than an actual instance.

Each IO device references an IO component type for which the device can be used as realization. It should be noted that one IO component type can be referenced by many different devices. For example, a temperature sensor device can be referenced by two different temperature sensor implementations (i.e. IO devices). However, a device component in the software layer does not depend on these particular implementations.

Similarly to the list of inputs and outputs provided by processing nodes, devices define a list of inputs and outputs that they require for communication.

Devices also support defining attributes that describe their extra-functional properties.

Platform instantiation

As opposed to processing nodes and devices, platforms represent particular hardware configurations which are not aimed for reuse. They are defined as sets of processing node and device instances, each instance referencing its type.

Besides defining instances of processing nodes and devices, the platform instantiation also describes how each device instance is connected to a processing node instance. This is done using *IO allocation* model entities. Each IO allocation references an IO instance of a processing node and a required IO of a device, denoting to which inputs or outputs, and how, a sensor or actuator is connected. Once that allocation of inputs and outputs is defined, the platform can also be checked to determine if IO requirements of all device instances are fulfilled by the processing nodes they are attached to.

3.2.3 Mapping layer

As already stated, the software and hardware model layers are defined in manner which reduces their interdependency to a minimum, therefore promoting reuse of structures defined in them. The mapping layer allows defining a set of *IO device mappings* which reference an IO component in the software layer and an IO device instance in the hardware layer. Each mapping denotes which sensor or actuator will be used as implementation for an IO component. By this, reusable units of the two layers are put in the context of a system.

A mapping between an IO component and an IO device can be created only if the type of the device instance references the type of the IO component instance. This constraint ensures that a system is deployed in a valid way.

The approach supports IO device mappings to be defined even in early stages of system development, before the full implementation is complete. This, for example, allows early analysis of system properties using the combination of software and hardware layers.

3.2.4 Example

To illustrate use of the proposed modeling approach, it will be demonstrated on an example. The example model, shown in Figure 3.3, describes a simple system that consists of an IO component for output to a display, a GDM2004 display device and an RCM2200 microcontroller. It should be noted that the example does not capture all functionality or hardware provided by the display or the microcontroller, but only the parts needed to demonstrate the approach.

The IO device component for communication with a hardware display provides interface for showing text messages. For this, the component receives as input the message text, and position, in rows and columns, where the message should appear.

The GDM2004 display provides only textual output capabilities, having four lines with 20 character columns each. The characters are sent to the display one at a time by writing data to a data register, with each character printed at the position defined by the inner state of the display, described by a cursor. The position of the cursor can be set at any time by writing data to a control register, and is increased by one column after each character write.

For the purpose of the example, the RCM2200 microcontroller will provide two ports, PA and PE, which can be either used for parallel 8-bit input or output, or as 8 separate single-bit inputs or outputs.

Processing node specification defines reusable model elements which describe processing nodes, IO types, and IO instances provided by processing nodes. In this example, processing node specification consist of a single processing node type, the *RCM2200* microcontroller, and two IO types: *OneBitIO* which sends or receive s single bit value from a hardware pin, and *IO8BitPort* which can be used to send or receive eight bits in parallel through eight hardware pins. To describe the communication capabilities of the microcontroller, *RCM2200* contains four IO instances: *PE0*, *PE4* and *PE5* of type *OneBitIO*, and *PA* of type *IO8BitPort*.

IO device specification provides reusable definitions of IO devices, together with IOs that devices require for communication. IO device specification for the example consists of an IO device *GDM2004 Display*. The device defines four required IOs: *registerSelect*, *rw* and *enable* of

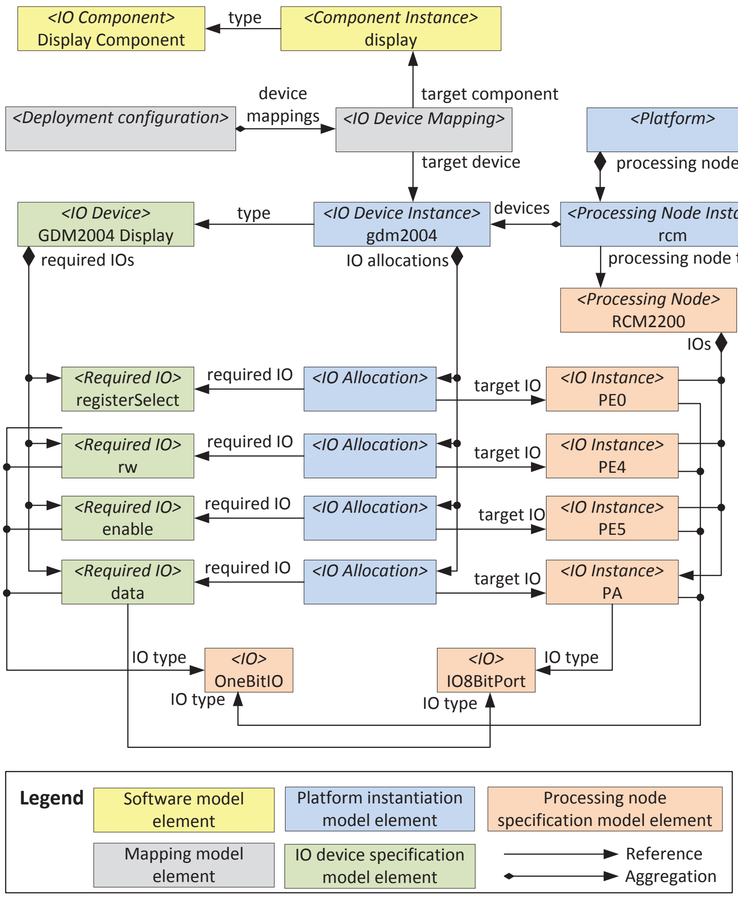


Figure 3.3: Example system, consisting of an IO component for display output, a GDM2004 display, and an RCM2200 microcontroller.

type *OneBitIO*, and *data* of type *IO8BitPort*.

Platform specification describes instances of processing nodes and IO devices, and how the two are connected. The top-level model element of this part of the code, a platform, consists of an instance of the *RCM2200* processing node, named *rcm*. The processing node instance in turn contains an instance of *GDM2004 Display* IO device, named *gdm2004*. The connection between the IO device instance and the processing node instance is defined by four IO allocations contained by *gdm2004*. These allocations connect *registerSelect*, *rw*, *enable* and *data* required IOs of the device to *PE0*, *PE4* *PE5* and *PA* IOs provided by the node, respectively.

The software layer of the example, which describes software components and connections between them, consists of a single IO component type, *Display Component*, and an instance of that component type named *display*.

The software and the hardware layers of the model are connected by the mapping layer. In this example, the mapping layer consists of a deployment configuration, which contains only one device mapping. The mapping connects the *display* IO device component instance and the *gdm2004* IO device instance.

3.3 Automatic synthesis of executable code

Code synthesis is a common way to provide executable code from software models. Generation of efficient code, compared to interpretation-based model execution, is especially valuable in embedded systems domain, where processing resources and available memory are scarce. In component-based systems, synthesis most often implies creating glue-code which connects reusable code fragments defined and encapsulated by software components. In this section we define how similar synthesis principles can be combined with the previously presented meta-model which includes IO devices, in order to automatically generate code for communication with sensors and actuators. The synthesis is based on two principles – (i) definition of context-independent reusable code elements, and (ii) automatic generation of system-specific code based on a model of the system. The result of the synthesis is a combination of code elements that gives a system-specific deployable functionality. An overview of the generation process is given in Figure 3.4.

The synthesis method is based on the C programming language, as C is the language still most common language used in the embedded domain. However, the principles used in this solution are not limited to C and could also be implemented in other programming languages.

As already mentioned, the synthesis method separates predefined reusable code elements, to which we refer to as *input code*, and generated system-specific code elements which we call *output code*.

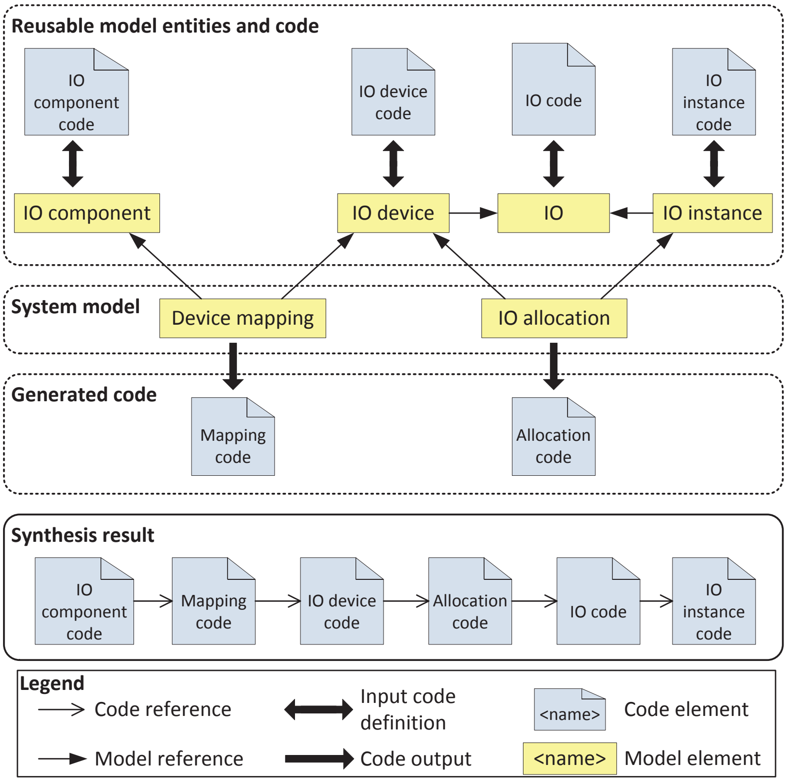


Figure 3.4: Overview of the synthesis process.

The following sections give a detailed description of input and output code. The code element descriptions are complemented by Figure 3.5, which gives an overview of all the elements, the parts that the elements consist of, and the relations between these parts. The descriptions also include code examples, based on the example system given in Section 3.2.4.

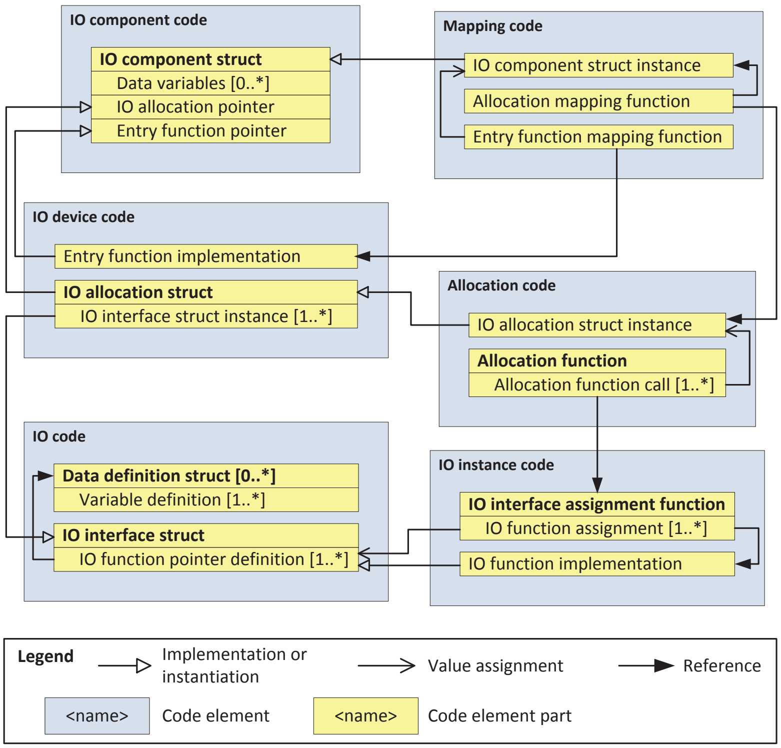

IO component code
IO component struct
Data variables [0..*]
IO allocation pointer
Entry function pointer
Mapping code
IO component struct instance
Allocation mapping function
Entry function mapping function
IO device code
Entry function implementation
IO allocation struct
IO interface struct instance [1..*]
Allocation code
IO allocation struct instance
Allocation function
Allocation function call [1..*]
IO code
Data definition struct [0..*]
Variable definition [1..*]
IO interface struct
IO function pointer definition [1..*]
IO instance code
IO interface assignment function
IO function assignment [1..*]
IO function implementation
Legend
Implementation or instantiation
Value assignment
Reference
<name>
Code element
<name>
Code element part

Figure 3.5: Overview of the code elements used and generated by the synthesis, and relations between them.

3.3.1 Input code definition

Input code elements are defined for elements of the model which are intended to be reused. These are IO components in the software layer, IO devices and IOs in the IO device specification of the hardware layer, and IO instances in the processing node specification part of the hardware layer. Each input code element is named after the model element it is attached to. We will now give a detailed description of the input code elements.

IO component code

IO component code is used to provide a common interface which allows connecting components in the software layer to various implementations of an IO device. As this code contains no implementation of functionality, it leaves the software layer completely independent of platform-specific implementation.

IO component code consists of a definition of a structure that includes: (a) definitions of zero or more variables to be used for communicating data to and from an IO device, and (b) pointers for mapping and allocating appropriate platform-specific functionality to the software component layer. While for storing IO allocation data a void pointer is used, device mapping is done via a pointer to an entry function which implements the IO device-specific functionality.

It should be noted that IO component code only defines a signature (list of parameters and the return type) of the entry function, while the implementation is defined in the IO device code. The single input argument to the entry function is the IO component structure, which is used by the entry function implementation to assign or read data values, and use appropriate platform IO functions. A description of how these pointers are assigned is given in Section 3.3.2.

Example: The input code for the *Display Component* defined in the example system can be seen in Listing 3.1. The IO component structure begins at line 1. Lines 2 and 3 contain data variables that will be used for communication between the component and the IO device. The pointer for referencing allocation data is defined on line 6, and line 7 defines the pointer for referencing the IO device entry function implementation.

```
1 typedef struct DisplayComponent {
2     int row;
3     int column;
4     char* text;
5
6     void * ioAllocation; // Pointer to IO allocation
7     void (*entryFunction) (struct DisplayComponent*);
          // Pointer to device specific entry function
8 } DisplayComponent;
```

Listing 3.1: IO component input code for *DisplayComponent*.

IO code

IO code has a role similar to IO component code. It too contains no implementation, but only defines an interface for communicating with an IO device using a specific IO type. The main element of IO code is the *IO interface structure*. This function contains pointers (signatures) for one or more functions that will provide platform-specific implementation of communication. The number of functions can differ for different kinds of IOs, and some of the functions can also be used for configuration of the communication channel, rather than the actual communication. IO code can also contain definitions of data structures used as arguments of the communication functions in case the arguments are not basic C types. How an instance of the IO interface structure is used to allocate IO devices to IOs is described in Section 3.3.2.

Example: The IO code defining the interface for *OneBitIO* is shown in Listing 3.2. Lines 2 to 5 of the listing define the data structure for configuring the port. The IO interface structure that defines pointers for functions used for interaction with the port is located at lines 8 to 12. The interface structure defines three functions, one for writing data to the port, one for reading data from the port, and one for configuring the port using a data structure defined in lines 2 to 5.

```
 1  // Data definition
 2  typedef struct  {
 3    int isOutput;
 4    int isOpenDrain;
 5  } OneBitIO_configData;
 6
 7  // IO interface structure
 8  typedef struct  {
 9    void (*writeData) (int);
10    void (*readData) (int*);
11    void (*configure) (OneBitIO_configData*);
12  } OneBitIO;
```

Listing 3.2: IO input code for *OneBitIO*.

IO instance code

IO instance code gives a platform-specific implementation for the IO communication functions defined in the interface structure of the IO code. For IO instance code to be valid it must implement all the functions defined in the interface structure. IO instance code also defines an IO interface assignment function that receives an IO interface structure, and assigns function implementations to the function pointers of the structure. How this function is used for the actual allocation is described in Section 3.3.2.

Example: Listing 3.3 shows the IO instance code for microcontroller port *PE0*. Lines 2 to 12 contain functions that implement controller-specific communication through *OnBitIO*. These functions adhere to function signatures defined in the IO interface structure provided in the previous section. The IO interface assignment function, which assigns function definitions contained in lines 2 to 12 to an instance of IO interface structure is located on lines 16 to 20.

```
 1  // START functions implementing IO
 2  void PE0_writeData(int value) {
 3    BitWrPortI(PADR, &PEDRShadow, value, 0);
 4  }
 5
 6  void PE0_readData(int* value) {
 7    BitRdPortI(PADR, &PEDRShadow, value, 0);
 8  }
 9
10  void PE0_configure(OneBitIO_configData* data) {
11    BitWrPortI(PEFR, &PEFRShadow, 0, (1<<0));
12    BitWrPortI(PEDDR, &PEDDRShadow, data->isOutput,
          (1<<0) & PEDDRShadow);
13  }
14  // END functions implementing IO
15
16  // Assigning functions for allocation
17  void allocate_PE0(OneBitIO* allocation) {
18    allocation->writeData = PE0_writeData;
19    allocation->readData = PE0_readData;
20    allocation->configure = PE0_configure;
21  }
```

Listing 3.3: IO instance input code for *PE0*.

IO device code

IO device code provides implementation for a specific kind of sensor or actuator. This includes the protocol used to communicate with the IO device, possible adaptation of data, and calls to IO functions.

The IO device code provides a device specific implementation for a IO component using an entry function. This function has a signature matching the signature of the entry function definition in the IO component code (described in Section 3.3.1. The IO requirements of a device are captured by an IO allocation structure definition. The structure contains one member for each required IO, using IO interface structures defined by the IO code for member types.

```c
// Structure for IO allocation
typedef struct GDM2004_allocation{
  OneBitIO registerSelect;
  OneBitIO rw;
  OneBitIO enable;
  IO8bitPort data;
} GDM2004_allocation;

// Implementation of the entry function
void entry_GDM2004(DisplayComponent* instanceData) {
  int i;
  GDM2004_allocation* alloc = (GDM2004_allocation*)
      instanceData->ioAllocation;

  GDM2004SetPosition(instanceData->column,
      instanceData->row, alloc);
  for (i=0; i < strlen(instanceData->text; ++i) {
    GDM2004PrintChar(instanceData->text[i], alloc);
  }
}

void GDM2004SetPosition(int column, int row,
    GDM2004_allocation* alloc) {
  // Implementation skipped
}

void GDM2004PrintChar(char c, GDM2004_allocation*
    alloc) {
  alloc->registerSelect.writeData(1);
  alloc->rw.writeData(0);
  alloc->enable.writeData(1);
  Delay_60usec();
  alloc->data.writeData(c);
  Delay_60usec();
  alloc->enable.writeData(0);
  Delay_60usec();
}
```

Listing 3.4: IO device input code for *GDM2400*.

As it will be described in the following sections, an instance of this structure will be referenced by the allocation pointer in the IO component code, allowing calls to platform-specific IO functions.

Example: The IO device code for *GDM2004 Display* is given in Listing 3.4. This code provides the GDM2004 LCD display specific implementation for *Display Component*. The definition of the IO allocation structure for the device starts at line 2. The structure contains four members, one for each IO required by the device. Lines 10 to 18 contain the the GDM2004-specific implementation of the *GDM2004 Display* entry function, matching the signature of the function pointer defined in the IO component code. On line 12, the allocation data is extracted from the IO component structure of the *Display Component* instance. This data is then passed to the device-specific functions on lines 14 and 16. How the IO interface structures of the allocation data are used for communication can be seen in the *GDM2004PrintChar* function which starts on line 26.

3.3.2 Output code generation

Generated output code creates connections between previously described input code elements, allowing them to form a system-specific functionality. How the input elements should be connected is defined by system-specific model elements: IO allocations and device mappings. Output code for the two is generated in separation, and the details of the generation are given in the following sections.

IO allocation code

IO allocation code provides connections between instances of IO devices and IOs provided by processing nodes. It enables IO devices to make calls to low-level IO functions, while abstracting the specifics of IO communication. IO allocation code generation starts with creating an instance of the IO allocation structure for each IO device instance of the platform model. The exact type of the structure is determined by the IO device referenced by the instance. A unique name for each structure instance is inferred from the name of the device instance. Is should be noted that one device can have multiple allocations, one for each required IO, but still only one IO structure instance, covering all the allocations, is generated.

The generation continues with creating an allocation function which assigns appropriate IO functions to function pointers in the IO allocation structure instances. The allocation function contains a call to an IO interface assignment function (defined by the IO instance input code) for each allocation element. The exact function name to be used is determined by the IO instance referenced by the allocations. The argument for the function call, an IO allocation structure instance and its IO interface structure member, are determined by the device instance element which is the parent of the allocation model element, and the required IO referenced by the allocation.

Example: The allocation code, shown in Listing 3.5, starts by creating a new instance of the IO allocation structure for *GDM2004Device* named *gdm2004*. The allocation function which calls an appropriate IO interface assignment functions for each IO required by the device instances begins on line 3. The body of this function is generated based on the IO allocations defined in the system model. As the only device instance in the example is *gdm2004*, the function only assigns the members of its IO allocation structure.

```
1  GDM2004_allocation gdm2004;
2
3  void doIOAllocation() {
4    allocate_PE0(&(gdm2004.registerSelect));
5    allocate_PE4(&(gdm2004.rw));
6    allocate_PE5(&(gdm2004.enable));
7    allocate_PA(&(gdm2004.data));
8  }
```

Listing 3.5: Allocation output code for the example system.

Device mapping code

Device mapping code connects IO software component code to the platform-specific code. Similarly to the allocation code, the generation begins with creating one instance of the IO component structure for each instance of an IO device component, using the IO component referenced by the instance to determine the exact structure type.

After this, an allocation mapping function is created. The purpose of this function is to connect the IO allocation structure instances of the IO allocation code with the previously generated IO component structure instances. For each device mapping, one statement in the function body is created. In each of these statements, an IO allocation structure is assigned to an ioAllocation void pointer of an IO component structure instance. The structure instances to be used are defined by the IO component instance and IO device instance referenced by the device mapping.

The final step of device mapping code generation is creation of a function which is responsible for mapping entry functions defined by IO device code to the IO component instances. Each statement of this function contains an assignment of an entry function to the entry function pointer of an IO component structure instance, one for each device mapping model element. The structure instance to be used is determined by the IO component instance referenced by the mapping. The entry function for the assignment is selected by the IO device type reference of the IO device instance which is referenced by the mapping.

Example: The code for device mapping is given in Listing 3.6 and starts with a definition of *DisplayComponent* IO component structure named *display*. IO allocation mapping is performed by a function beginning at line 3. As again the only device instance in the system is *gdm2004*, the only line of this function assigns its IO allocation structure to the IO allocation structure pointer of *display*. The function that implements mapping of device entry functions starts on line 7.It assigns the entry function of *GDM2004 Device* to *display*.

Once defined, the three functions defined by the IO allocation and mapping output code can be called during system initialization to bind together the input code to form system-specific functionality.

```
1  DisplayComponent display;
2
3  void doAllocationMapping() {
4    display.ioAllocation = &gdm2004;
5  }
6
7  void doEntryMapping() {
8    display.entryFunction = entry_GDM2004;
9  }
```

Listing 3.6: Mapping output code for the example system.

3.4 Implementation and evaluation

The IO device modeling and code synthesis approach proposed in this chapter has been implemented as a prototype tool. The following sections first describe the details of the implementation, and then provide information about how the tool was used to evaluate the approach.

3.4.1 Implementation

The prototype tool for modeling of sensors and actuators and automatic code synthesis of code for communication using these devices was implemented in the context of the ProCom component model. To provide integration of IO device modeling with ProCom, the prototype tool was build as a plug-in for PRIDE [8] – an Eclipse based integrated development environment for ProCom. This approach also allowed use of Eclipse Modeling Framework [55] to define meta-model for modeling IO devices, and to create Java code for model elements and visual model editors. Although most of the new model elements were defined as an addition to ProCom ones, the standard ProCom meta-model had to be updated to allow definition of IO components and IO dependencies inside the software model.

The part of the tool for automatic code synthesis relies on Eclipse Modeling Framework [55] for model traversal and Java for code generation. The tool relies on universally-unique identifiers to link reusable model elements with files with pre-defined input code elements, and generates the mapping and allocation code. The only element left out of automatic synthesis process is a call from standard ProCom component entry function to the IO device entry function mapped by the synthesis process, as it would require making changes to standard ProCom synthesis.

3.4.2 Evaluation

The aim of evaluation of work presented in this chapter was to (i) test whether the method can be used to model a realistic system and generate correct executable code, and (ii) measure the overhead of the generated code in both program size and execution time compared to handwritten code. For this purpose we have used the previously described example containing a GMD2004 display connected to a RCM2200 microcontroller. As current implementation of ProCom synthesis does not provide synthesis of code RCM2200, the test have been executed on the generated code in isolation, by directly calling the mapped IO device entry function.

For the first part of evaluation was preformed by first creating the system model, and then and synthesizing the communication code code. The aim of the test was filling out all lines of display with text. As a result, the display was showing expected output, confirming that the model and synthesized code are valid.

Overhead of the generated code was measured by comparing four different implementations of functionality which initializes communication with the display and fills one line of the display with an output. The first implementation used a pre-existing hand-written library for communication with the display. The second implementation used code synthesized by the method proposed in this chapter, with input code elements based on the code from the library used in the first implementation. For the third implementation, the code from the first implementation was optimized by composing all the code needed for outputting one line to the display into one function, thus removing all unnecessary code and function calls. The fourth implementation also used synthesized code, but the input code elements were based on the third, optimized implementation.

While the deployed program size measurements took into account the whole program, including the program initialization and code stack of RCM2200, the execution time measurement included only the function call for outputting a line on the display. To perform more accurate measurements of short time intervals, the execution time was measured for 3000 consecutive calls to the output function, and calculated by dividing the accumulated time by the number of function calls. Each execution time measurement was repeated 10 times to account to possible execution anomalies. However, as the program executed on the microcontroller without possible preemptions, all measurements for 3000 output function calls were within a range of 5 milliseconds, which is a negligible difference when scaled to execution of one function call. The results of the measurements, together with the calculated overhead, are given in Table 3.1.

The results show that execution time overhead for the synthesized code compared to the hand-written one is 59.74% for the first and second implementation alternative (based on existing library code), and 41.57%

Table 3.1: Execution time and code size of measured programs.

Value	Hand-written	Synthesized	Overhead
Execution time (ms)	4.58	7.32	59.74%
Program size (bytes)	33100	34851	5.29%

Value	Hand-written optimized	Synthesized optimized	Overhead
Execution time (ms)	3.78	5.36	41.57%
Program size (bytes)	32714	35194	7.58%

for the third and fourth implementation alternative (based on optimized code). Program size overhead is 5.29% when comparing the library-based alternatives, and 7.58% for the optimized ones. Such results are expected, as the synthesized code introduces more function calls, requires de-referencing of various function and data pointers, and contains more statements than the hand-written code. It should be noted that the execution time and program size overhead was measured on isolated communication with the display, and would have less impact if included as only a part of a more complex system. Because of this, we argue that the performance and memory footprint of the synthesized code is still acceptable, while allowing faster and less error-prone development.

3.5 Summary

In this chapter we have presented a method for modeling sensors and actuators in component-based development, and a method for automatic synthesis of code for communication with such IO devices in a component-based environment. The model supporting IO devices is divided into multiple loosely-bound layers, and has a clear distinction between types and instances, with the aim of promoting reuse of model elements in different scenarios. The presented code synthesis method relies on the proposed models, and defines how to specify pre-defined input code elements for the reusable model entities, and how to generate glue-code between these elements based on the system configuration. The presented methods have been exemplified, and evaluated using a prototype tool.

Chapter 4

Automatic generation of inter-node communication

A common approach for facilitating development of distributed applications is by providing means to describe them using platform-independent[1] models. In this way a developer can focus on providing correct functionality while abstracting the complexity of communication between distributed nodes. However, using such an approach still requires implementing this communication before an executable system is deployed. This can be done by either transforming the platform-independent models to directly to platform-specific executable code, or by introducing intermediate platform-specific models. Although in component-based approaches the former is more common, it leads to having properties of the platform-specific implementation available only in late stages of system development. Thus, such properties can be unavailable during most of the development process. This can be avoided by using platform-specific models, which can be available early in the development process. However, in current component-based approaches, such models most often need to be generated and updated by hand, resulting in increase of development time and risk of introducing errors. An alternative to

[1]In the context of this chapter, we refer to platform-independence only in the context of communication between distributed processing nodes.

manual creation of platform-specific models is generating them automatically. To account for the possibility of inconsistencies between the platform-specific and platform-independent models, approach for automatic generation should also address co-existence of the two by providing means for synchronization.

An example of an approach which supports separation of platform-independent and platform-specific models can be seen in the IEC 61499 standard [28, 67, 70]. The standard allows platform-independent development of distributed applications, abstracting some of the additional complexity of communication between distributed nodes. While some development tools [26, 47] automatically generate code for this communication when deploying an application to hardware, inter-node communication can also be implemented by manually adding specialized communication components on the model level.

In this chapter we present a framework for automatic generation of inter-node communication on the level of software models, and provide an implementation of the framework for the IEC 61499 standard.

An overview of the approach is depicted in Figure 4.1. The generation is performed by utilizing platform-independent software models, platform models, and models of mapping between the two. Based on these models, the generation method first determines distributed-communication requirements of an application, determines how these requirements can be satisfied, and creates communication components in the platform-specific models. In the end, information about the generated communication is propagated back to the platform-independent model in form of annotations to connections between components.

The approach allows developers to model distributed functionality without having to manually implement the details of inter-node communication, and provides means to keep platform-independent and platform-specific models synchronized through annotations to model elements. The communication components can be generated even in the early stages of development, before the system is fully implemented, which makes it easier to explore different allocation options. To increase flexibility and make it easier to apply the method do different component frameworks, the generation is separated into multiple phases, each with clearly defined inputs and outputs. The framework also distinguishes between the generic generation mechanism and generation of protocol-specific elements, allowing the method to be easily extended with support for different protocols.

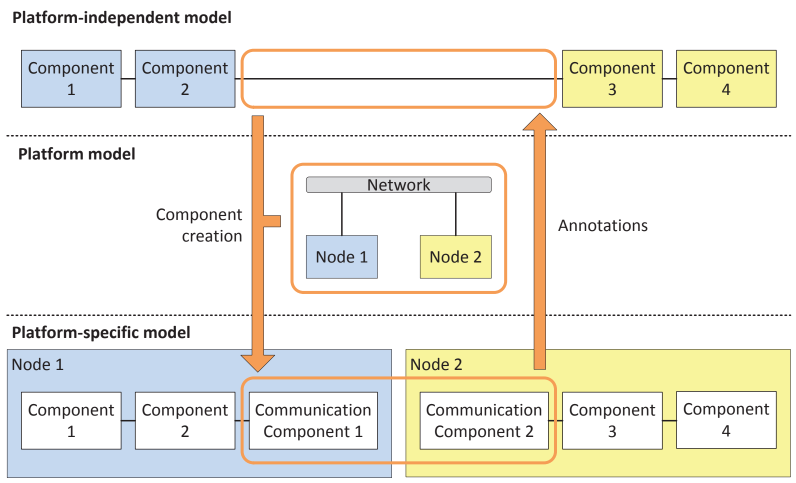


Figure 4.1: Overview of the generation process. Matching colors of software components and processing nodes denote mapping between the two. While the orange boxes specify parts of the models used in the generation, the orange arrows depict flow of information between these parts.

Since the generation process uses an intermediate model to describe the communication needs of an application, in the following sections we first introduce the intermediate communication model, and then describe the details of the generation. After that we give a description of the implementation of the framework for the IEC 61499 standard, including a prototype tool, and exemplify the approach. The contributions of this chapter are based on Paper C [36] of the main contributing publications.

4.1 The communication model

We capture the communication between components located on different nodes by creating a communication model. The Ecore metamodel of the communication model can be seen in Figure 4.2. The main elements of the model are *Channels*, which represent data or events produced together on the same source node, and transferred together as messages to one or more destination nodes.

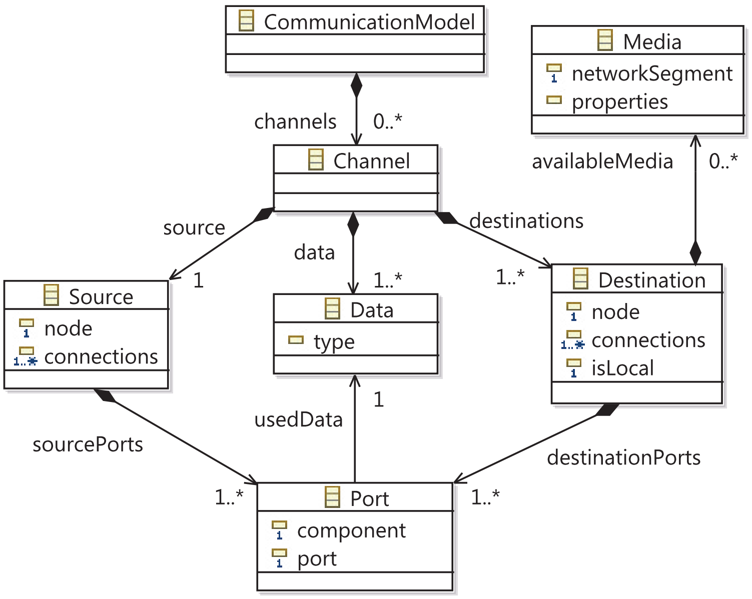


Figure 4.2: Ecore metamodel that defines the communication model.

The content of a message sent through a channel is defined by a set of *Data* elements. Each data element defines the type of data it represents using the *type* attribute. As events are not distinguished by types, and carry no semantics besides the occurrence of an event, we represent them by *Data elements* with *type* set to *none*.

In addition to a set of data elements, a channel also contains one *Source* and one or more *Destination* elements. Both of these element types have a reference to the platform node which is the endpoint of the communication. The *Destination* element has an additional *isLocal* flag, which indicates if the source and destination nodes reside on the same physical node, or if they are distributed on separate physical nodes. As an example, in the IEC 61499 implementation of the generation method, communication between two resources belonging to the same device is marked as local. The inter-node connections for which the *Source* and *Destination* elements are generated are referenced using the *connections* attributes.

The *Source* and *Destination* elements also describe how the messages are created and consumed by components. This is done by a set of *Port* elements which are added either to the *sourcePorts* set of a source or the *destinationPorts* set of a destination. Each of these elements define from which port of which component in the application model the message data is read (in case it is added to a source) or to which port of which component the data needs to be delivered (in case it is added to a destination). This is done using the *component* and *port* attributes. The message element that is generated by or delivered to the referenced port is denoted by a reference to a *Data* element of the *Channel*.

Each *Destination* element can also contain a number of *Media* elements. These elements are used to describe which communication media can be used for communication between the destination and the source of the channel. Besides the *networkSegment* attribute which references the network segment of the platform model, a *Media* element also contains a *properties* attribute which can contain information about how the destination node is connected to the network segment, for example an IP address of a device on an Ethernet network.

4.2 Generation process

The generation of platform-specific model elements is separated into four activities, each with clearly defined inputs and outputs, and the framework also distinguishes between the generic generation mechanism and generation of protocol-specific elements. These phases are (i) communication model extraction, (ii) detection of available media, (iii) protocol selection and (iv) component creation. An overview of the generation process can be seen in Figure 4.3.

Separation of the generation process in multiple phases introduces a level of flexibility which results in multiple benefits. First, the method can easily be extended, for example to add new optimization algorithms during protocol selection, or to update the generation with new communication components or protocols. The level of automation of the generation process can be varied, which provides a possibility for manual input of a developer. Also, the separation allows easier adaptation of the method to different component frameworks.

The following sections give a detailed description of the four generation phases.

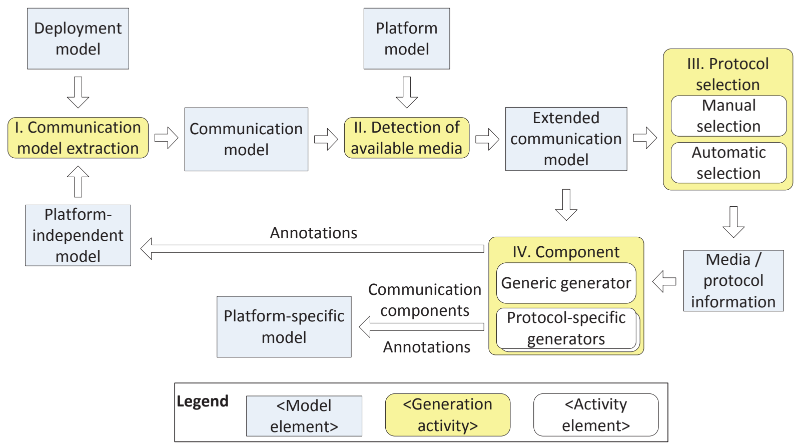


Figure 4.3: Overview of the communication component generation process.

4.2.1 Communication model extraction

The communication model of an application is extracted from the platform-independent model and the deployment model. The process starts with detecting connections which connect components deployed to different platform nodes. A *Channel* element is generated for each group of connections for which source values are generated together and on the same node. Then, the *Data* element set of each channel is generated based on the information about the sources of the connections. The channels' single *Source* element is created and initialized with the information about the source node and the represented connection, and *Port* elements of the *sourcePorts* set are created based on the output ports from which the connections start.

The represented connections are then grouped by the nodes that their destination components are mapped to. For each such group, a *Destination* element is added to the channel. A *Port* element for each connection is then added to the *destinationPorts* set, and initialized to point to the target port of the connection.

4.2.2 Communication media detection

Once the communication model has been derived, it can be used to determine which media alternatives are available to implement the inter-node communication. This is done in combination with the model of the platform.

As channels can have multiple destinations, each on a distinct platform node, media detection has to be performed separately for each channel destination. Available media for a destination is determined by finding all communication networks in the platform model to which both the node of the destination and the node of the channel source are connected. For each available media, a *Media* element is added to the *Destination* and the value of the *properties* attribute for the new element is set based on how the node is connected to the network.

The results of the media detection are added to the existing communication model and the new extended model is stored.

4.2.3 Protocol selection

After the media detection is done, it must be decided which of the available media will be used for communication, and how (i.e. using which protocol). The protocols which can be used to implement communication using each communication media are defined by the available protocol-specific generators (described in more details in the following section). Each protocol-specific generator provides a list of media it can be used for, and for each supported media a list of available protocols.

Protocol selection is a combination of manual and automated process. Manual protocol selection allows developers to use expert knowledge to satisfy possible communication constrains that can not be expressed by models, or to obtain a desired system behavior. For communication that does not have any specific requirements to be considered, media and protocols can be selected automatically. In addition to just providing a viable communication solution, automated selection can also include various optimizations of communication on the system level. Thus, automated protocol selection reduces the amount of activities the developer has to perform to implement the system, and thus allows faster development.

4.2.4 Communication component creation

The actual creation of the communication components is based on the extended communication model and the information about selected communication protocols. It is done using two mechanisms: a *generic generator* and a set of *protocol specific generators*.

The task of the *generic generator* is to traverse the communication model and initiate creation of communication components for sources and destinations. For each destination one component is generated on the destination node. Creation of components on the source side is more complex. Messages from one source can be delivered to more than one destination, and the destinations can require different communication media or protocol. Because of this, in some cases there is a need to generate more than one communication component for a single communication channel source, each for a specific media or protocol.

The component creation initiated by the generic generator is performed by *protocol-specific generators*. A protocol-specific generator first determines the type of the component that needs to be generated. This is done based on the selected media and protocol, and information about data that needs to be transferred. Once the type of the component is determined, it is added to the platform-specific model. After adding the component, a protocol-specific generator also configures it for communication. As already mentioned in the previous section, each protocol-specific generator provides a list of media and protocols it supports. This information is also used by the generic generator to determine which specific generator is to be used for each communication channel.

After the communication components are generated, the *generic generator* creates connections between application components and the generated communication components based on the information stored in the *Port* elements of the communication model. In the end, the information about the connections represented by sources and destinations is used to annotate these connections with information about the generated components.

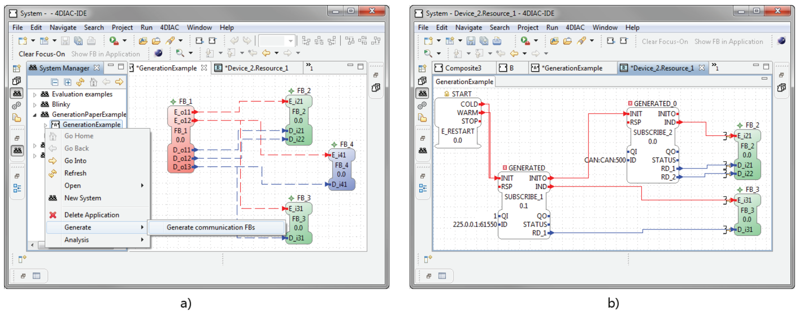


Figure 4.4: Screenshot of a) example application and the automatic generation menu in 4DIAC-IDE and b) a resource-specific software model containing components created by the prototype generation tool.

4.3 IEC 61499 implementation of the framework

To demonstrate applicability of the communication generation framework it was applied to the IEC 61499 standard, and implemented as a prototype tool. The tool was developed as a plug-in for the 4DIAC-IDE, an open-source IEC 61499 development environment [56]. Integration with 4DIAC allows us to execute the tool using the graphical model editors, and perform generation using existing system models. Generation results in systems which are fully executable without any need for manual editing of resource-specific software models or code.

A screenshot depicting the 4DIAC-IDE is shown in Figure 4.4. The figure contains a) the application model from the previous example and the generation menu added by our plug-in, and b) the resource-specific software model for *Resource 2* containing the generated communication blocks. The prototype tool is freely available for download[2].

The section continues by describing the details of how the generation process was applied to the specifics of IEC 61499, and how the prototype tool implements the the framework.

[2]http://www.idt.mdh.se/~jcn01/research/4DIAC-plugins/

Communication model extraction

The IEC 61499 *application model* provides both a platform independent view of software and information about deployment decisions. Because of that, the IEC 61499 implementation of the generation performs communication channel extraction using only this model.

The requirements for distributed communication are determined by detecting event or data connections which link function blocks deployed to different resources. Such connections are grouped into channels using the The WITH operators specified for the output ports of function blocks, as each WITH defines one event and an arbitrary number of data outputs generated at the same time.

As already mentioned, destinations which target IEC 61499 resources residing on the same device as the source resource are marked as local destinations.

Communication media detection

Detection of media available for satisfying communication requirements is performed based on the IEC 61499 *system model*, which describes which devices exist in the system and which network segments, and how, connect the devices.

Protocol selection

As the aim of the contribution presented in this chapter was to provide a definition of a framework for generation of distributed communication, rather than actual optimization of communication, or a fully working tool, the implementation of protocol selection includes only basic functionality. The implemented automated selection is able to determine a valid set of communication protocols which can be used for communication, is such set exists. We have however omitted providing an implementation of manual protocol selection, as this would require a fairly complicated user interface.

To provide an example of optimization during automated selection, we have implemented an algorithm which uses heuristics to find common media for multiple destination of a single communication channel. The goal of this algorithm is to reduce the number of messages that need to sent over the network.

Communication component creation

When applied to IEC 61499, the component creation generates communication function blocks in *resource models*, which are platform-specific software models of the standard, while annotations about generated components are created in both *application model* and *resource models*. The prototype tool implements two protocol-specific generators: a generator for UDP communication over an Ethernet network, and a generator for communication using CAN bus.

Communication using UDP over Ethernet is part of an interoperability provisions defined by Holobloc [24]. In the platform-specific resource software model, the communication is implemented using two standardized function block types: the PUBLISH function blocks are used to send multicast UDP messages to the network, while SUBSCRIBE function blocks receive such multicast messages. There exist multiple versions of both function block types, each with a different number of data values they send or receive. To enable sending messages from one PUBLISH function block to one or more SUBSCRIBE function blocks, they must be configured to send and receive messages on the same UDP port.

The protocol-specific generator for UDP over Ethernet generates PUBLISH function blocks for channel sources, and SUBSCRIBE blocks for destinations, choosing the correct versions of the function blocks based on the number of data values transferred by the channel. Because of the specifics of the implementation of message decoding in SUBSCRIBE function blocks, which does not allow data output ports of these function blocks to be left disconnected, in some cases the generator creates multiple PUBLISH blocks for a single source. In this case each destination is treated as a separate communication channel. After creation of communication function blocks, each channel is assigned a unique UDP port. The selected ports are then used to configure all the generated blocks.

In case of communication destinations that are marked as local, the generator creates special version of function blocks which provide implementation optimized for communication between resources residing on same physical device.

The generator for CAN communication functions in a manner similar to the UDP one, generating PUBLISH and SUBSCRIBE function blocks configured for communication using CAN identifier parameter. However, current implementation of the IEC 61499 run-time that we used to build and test a prototype of the generation framework (described in the following section) does not support CAN communication. Because of this, systems which include function blocks created using this generator can not be deployed for execution.

4.4 Example and case-study

To validate the applicability of the framework and the tool, it will first be demonstrated on an extended example. After that, a description of how the generation framework was applied to two case-study system is given.

4.4.1 Example

The proposed communication component generation method will be demonstrated on an example system. The distributed application of the example is shown in Figure 4.5 a), and the platform model is given in Figure 4.5 b). The application contains three function blocks: *FB_1*, *FB_2* and *FB_3*. The example platform consists of three devices, each with a single resource: *Device 1* containing *Resource 1*, *Device 2* containing *Resource 2*, and *Device 3* containing *Resource 3*. Function block *FB_1* is mapped to run on *Resource 1*, *FB_2* and *FB_3* to *Resource 2*, and *FB_4* to *Resource 3*.

Figure 4.5 c) shows the communication model extracted from the application and deployment models. The extraction results in two communication channels, one for each WITH qualifier of *FB_1*. The first channel is used to transmit one event and two data values from *Resource 1* to a single destination on *Resource 2*. The model also shows at which port the message data originates, and which destination ports receive the data. The second channel transmits one event and one data value originating on *Resource 1*, and has two destinations, one on *Resource 2* and one on *Resource 3*.

Results from communication media detection are also depicted in Figure 4.5 c). Both *Destination 1* of *Channel 1* and *Destination 1* of *Channel 2* can be reached from their sources by either the CAN bus or the Ethernet connection. Ethernet is however the only media that can be used to communicate between the source of *Channel 2* and *Destination 2* of the same channel.

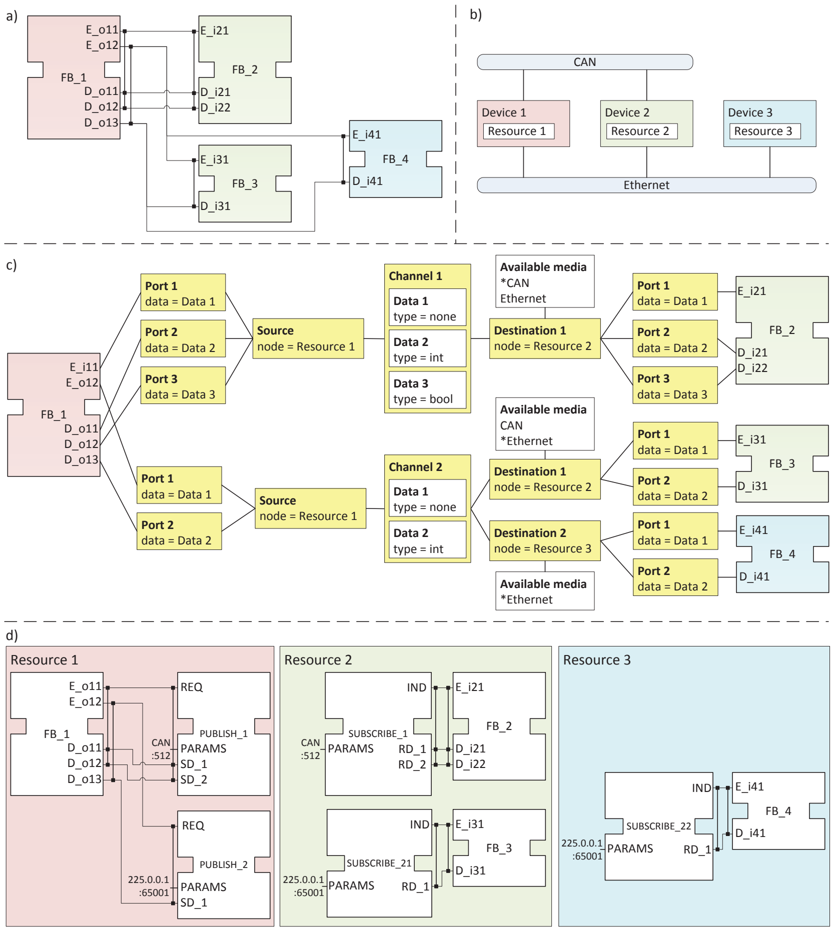


Figure 4.5: a) Platform-independent application model. b) Platform model. c) Communication model derived from the example application and platform model. d) Platform-specific resource software model containing generated communication components. The colors of the resource models matches the color of devices that they belong to, while the colors of function blocks denote to which device and resource they are deployed to.

We assume that the developer has manually selected the CAN bus as the media to be used for *Destination 1* of *Channel 1*, and that media selection for *Channel 2* is left to the tool. As Ethernet is a common available media to all destinations of *Channel 2*, it is selected to implement communication for both destinations by the automated selection process. The selected media are marked with an asterisk in the model figure.

The result of the component creation can be seen in Figure 4.5 d). The figure shows resource-specific software models of the system resources, which, in addition to the deployed function blocks, contain the inter-node communication function blocks. For the sake of simplicity, the models do not show all ports of PUBLISH and SUBSCRIBE function blocks, such as the ports used to trigger function block initialization. Communication between *FB_1* and *FB_2*, captured by *Destination 1* of *Channel 1* is implemented by *PUBLISH_1* and *SUBSCRIBE_1*. Communication from *FB_1* to *FB_3* and *FB_4*, described by *Channel 2* of the communication model, is implemented by a single publish function block,*PUBLISH_2*. Messages sent by this publisher are received by two subscribers: *SUBSCRIBE_21*, which implements *Destination 1* of the channel and delivers messages to *FB_3*, and *SUBSCRIBE_21*, implementing *Destination 2* of the channel and delivering messages to *FB_4*.

4.4.2 Case-study evaluation

In order to validate applicability of the framework in a realistic scenario, we have applied it to two example systems provided by the 4DIAC group[3]. The first system is an implementation of a traffic light system for pedestrian crossing, while the second one is simulation of manual operation mode of a mechanical press.

The first step of testing the generation was a thorough inspection of the original system functionality. After that, the two systems were recreated without communication components. During this process, we also needed to create connections between distributed devices and an Ethernet network segment, as these connections were left out of the original platform model. We then applied the automatic communication generation to the recreated models.

[3]Publicly available as a part of the 4DIAC project at http://sourceforge.net/projects/fordiac/.

The resulting platform-specific models were first visually compared to the same models of the original systems. As a result of the comparison, we concluded that the automatically generated models were in-line with the original one, although the original models contained some optimizations which were not present in the generated ones (e.g. reuse of same communication blocks for multiple communication channels). However, lack of such optimization in the generated models was to be expected, as providing optimization was not the primary goal of the contribution presented in this chapter.

After the visual inspection of the generated models, the systems containing them were deployed, and the functionality of the generated systems was compared to the one previously examined in the original models. In both cases, the generated systems provided the same functionality as the original ones.

4.5 Summary

In this chapter we have presented a framework which allows automatic generation distributed communication in component-based systems. The framework defines separate phases of generation, each with well defined inputs and outputs: (i) extracting communication requirements from platform-independent model and model of deployment, (ii) determining which communication media can be used to satisfy these requirements, (iii) a combination of automatic and manual protocol selection, and (iv) creation of communication components in platform-specific models. Clear separation between these phases allows framework to be easily extensible and adaptable to different component models. The defined framework has been applied to the IEC 61499 standard and implemented by a prototype tool. Applicability of the framework has been evaluated using an extended example, and two case-study systems, showing the applicability of the approach.

Chapter 5

Model-level timing and utilization analysis

Many embedded software systems are safety-critical and have a requirement to operate in real-time. For such systems the ability to provide functionality on time is as vital as the functionality itself. One possibility to ensure their correct behavior is by analysis of system properties. However, analysis is typically not performed until late in system development, leading to late detection of problems. Providing a way to perform analysis early in development can reduce the risk of redesign in late stages of development, thereby reducing development time and cost.

One approach that allows for early investigation of system properties is analysis on the model level. This type, which relies on determining system properties using abstract system models, provides multiple benefits. One of such benefits is that a system does not have to be deployed or even fully implemented to perform the analysis. The unimplemented parts of the system can be represented by abstract model elements, which can provide estimates of timing properties. During the development, these elements can gradually be replaced by the actual implementation, providing more accurate analysis results. Another benefit of model-level analysis is that it can also result in very efficient analysis methods, since the analysis is performed on an abstract view of a system, and doesn't need to process all the details of the implementation. Model-level approach to analysis can also include some disadvantages. For example, the fact that it is performed on an abstract view of a system can reduce

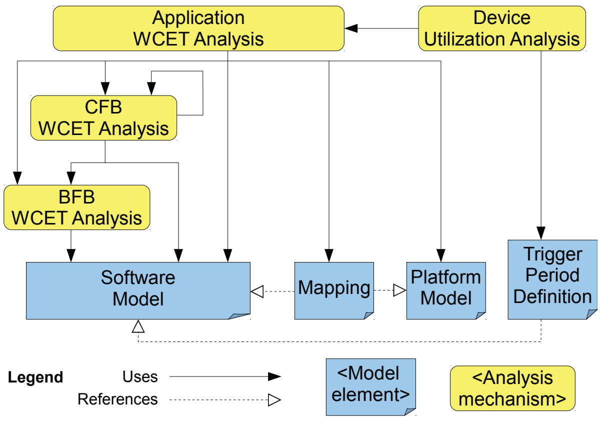


Figure 5.1: Overview of the analysis approach.

the accuracy of analysis results. The fact that in component-based systems are already developed using models makes this approach to analysis highly suitable to component-based systems.

An important property that can be analyzed in real-time embedded software systems is the Worst Case Execution Time (WCET) of tasks or components. The WCET value corresponds to the maximum time that a processing resource will be used to finish execution of a task or a component's functionality [48]. Knowing the WCET for the components of a real-time system is essential for ensuring it's timing requirements are met.

This chapter describes a model-level analysis approach for determining the WCET of components and applications in IEC 61499 systems, and a way to derive the utilization of processing resources in distributed applications. An overview of the complete method is shown in Figure 5.1. It consists of four analysis mechanisms, targeting different levels in the IEC 61499 software model. The first two analysis mechanisms, basic and composite function block WCET analysis, are performed using only platform-independent software models. Analysis of applications extends the function block analysis to take into account platform models and models of deployment. Finally, device utilization analysis combines the application analysis and information about periods of execution triggers.

The WCET data for each model element is calculated using the data of its subcomponents, based on the models of subcomponent interaction. For basic function blocks, this means that the results are attained by examining their internal execution control charts. Analysis of composite function blocks is performed in a compositional manner. The results for a composite are calculated by composing the existing WCET data attached to the function blocks implementing the composite. These results are then stored together with the composite, and used for analysis whenever a composite is instantiated on a higher level of hierarchy.

The rest of the chapter first gives a formal definition of IEC 61499 model elements. It then continues by first describing the WCET analysis of basic and composite function blocks. The approach is then extended with the ability to handle cyclic execution paths, and methods for analysis of platform-specific application WCET and device utilization is presented. After that, an overview of the implemented prototype analysis tool is given, together with information about evaluation of the approach. This chapter is based on the main contributing publications Paper D [40], Paper E [39] and Paper F [37].

5.1 Formal definition of IEC 61499

The description of the WCET analysis requires exact definitions of the involved IEC 61499 elements. This section provides these definitions, on a level more formal than is given by the standard [28]. Parts of the definition provided in this paper are based on work from Čengić and Åkesson [62]. Definitions that differ from the ones defined in the mentioned paper were modeled according to the IEC 61499 standard definition [28]. In this definitions we will disregard elements that are not relevant to the analysis described in this paper, such as data ports and connections between them. The following definitions constitute the representation of IEC 61499 systems used in this chapter

Definition 1. *A* ***function block interface*** *is represented by the following construct:*

Function block interface: $fbi = \langle E_i, E_o \rangle$ *where*

E_i *is a set of event inputs;*
E_o *is a set of event outputs.*

Definition 2. *A* ***basic function block*** *and it's subcomponents are represented by the following constructs:*

Basic function block: $bfb = \langle fbi, ecc, A \rangle$ *where*

fbi	*is a function block interface;*
ecc	*is an execution control chart (ECC);*
A	*is a set of algorithm functions.*

Execution control chart (ECC): $ecc = \langle Q, T \rangle$ *where*

Q	*is a set* $\{q_0, \ldots, q_{\lvert Q \rvert}\}$ *of ECC states;*
T	*is a set* $\{t_0, \ldots, t_{\lvert T \rvert}\}$ *of ECC transitions.*

ECC state: $q = \langle k_1, \ldots, k_{\lvert q \rvert} \rangle$ *where*

k_i	*is a ECC state action;*

ECC state action: $k = \langle a, e_o \rangle$ *where*

a	*is the algorithm to be executed,* $a \in A$*;*
e_o	*is the output event to be generated,* $e_o \in E_o$*.*

ECC transition: $t = \langle q_s, e_g, q_d \rangle$ *where*

q_s	*is the source state,* $q_s \in Q$*;*
e_g	*is the input event guarding the transition, or 1 if the transition is not guarded by an event,* $e_g \in E_i \cup \{1\}$*;*
q_d	*is the destination state,* $q_d \in Q$*.*

Definition 3. *A* ***service interface function block*** *is represented by the following construct:*

Service interface function block: $sifb = \langle fbi \rangle$ *where*

fbi	*is a function block interface.*

Definition 4. *A* ***composite function block*** *and it's constituting elements are represented by the following construct:*

Composite function block: $cfb = \langle fbi, fbn \rangle$ *where*

fbi	*is the function block interface;*
fbn	*is the internal function block network.*

Function block network: $fbn = \langle F, C \rangle$ *where*

F	*is a set of function blocks, each of which is either a bfb, sifb or cfb;*
C	*is a set* $\{c_0, \ldots, c_{\lvert C \rvert}\}$ *of connections between event ports.*

Connection: $c = \langle e_s, e_d \rangle$ *where*

e_s	*is the event port used as the source of the connection;*
e_d	*is the event port used as the connection target.*

For the purpose of the presented analysis **applications** being just function block networks. Therefore we do not need to introduce a separate formal definition for applications.

Definition 5. ***Deployment*** *of function blocks to platform devices is modeled by function* $\textsc{deployedTo}(fb)$*, returning the device* d *to which the function block* fb *is deployed to:*

Deployment: $\textsc{deployedTo}(fb) = d$

5.2 WCET analysis of function blocks

As already mentioned, the analysis of function blocks is performed by composing the WCET data of its subcomponents. It is applied to a single level of hierarchy at a time, starting from the bottom. In case of basic function blocks this is done by analysis of the ECC implementing the block, using the WCET of individual algorithms. For composite function blocks the WCET data is obtained by composing the WCET data of the constituent function blocks, according to the event connections in the internal function block network.

The compositional analysis approach is enabled by providing context-independent WCET data on the level of function block interface. In this way, WCET for function blocks, regardless of their type, can be described without the need of knowing details about their implementation.

The analysis presented in this section is performed only using platform-independent models, ignores the fact that WCET depends on the type of hardware that a function block is deployed to. How the analysis is extended to include platform-specific models and hardware-specific WCET values is described in Section 5.4.

At the bottom of the IEC 61499 model hierarchy are algorithms implemented by code rather than further defined by models. Determining the WCET values for algorithms is outside the scope of this thesis, and we assume that this information has been already established, either as expert estimates, measurements, of from code analysis (some existing methods and analysis tools are described in [68]). Similarly, service interface function blocks are also not further elaborated at model level, and thus not handled by the analysis. We assume the WCET data for these blocks have been defined manually, e.g. using code analysis.

In the following sections we first five an overview of how WCET data for function blocks is defined and normalized, and then describe details of basic and composite function blocks analysis.

5.2.1 WCET data definition

This section describes how WCET data for algorithms used for implementation of basic function blocks, and for all types of function blocks on the level of their interface. As already mentioned, the WCET values represent the maximum amount of time that an algorithm or a function block will require a processing resource for its execution. This time does not include waiting due to preemption or blocking.

The WCET for algorithms used to implement basic function blocks are defined as functions returning a single positive integer value. How this value is attained is out of scope of the presented analysis.

Definition 6. *The* ***WCET data*** *for an algorithm $a \in A$, represented by the function* $\textsc{wcet}(a)$*:*

$$\textsc{wcet}(a) \in \mathbb{N}$$

Defining WCET data for function blocks is, however, a more complex task. To allow compositional WCET analysis, this data has to be context independent, and describe possibly complex execution inside a function block on the interface level.

Each IEC 61499 function block can have multiple execution alternatives, depending on how the execution was triggered and the internal state of the function block. To be able to correctly describe these alternatives, the WCET data assigned a function block can contain more than one WCET entry, each described by a WCET value

The need for the WCET data to be context-independent means that the data entries do not only have to specify time needed for execution of function block functionality, but also the effects that this execution can have on the rest of the system. Since function blocks can transfer execution flow only through event ports, such effects can be described by the output events this execution generates. Because of this, besides a WCET value, each data entry also contains information about generated events.

The alternative WCET entries are organized in two data sets, based on how their execution is initiated: (a) *event WCET data* and (b) *internal trigger WCET data.*

The event data set consists of entries with information about execution initiated by the arrival of event inputs to the function block. Each event data entry is associated with an input event of the function block, which can trigger the execution alternative described by the entry.

Similarly, the internal trigger WCET data contains information for executions initiated by the internal activities in the function block. In this case the WCET data entry sets are associated with a trigger ID instead of an input event.

An event input or an internal trigger can result in multiple internal execution paths, each with a different WCET value and a different set of generated outputs. When considering a function block in isolation, in some cases it can not be decided which of these internal paths will lead to the worst execution time on the system level. Because of this, each event input and internal trigger can gave more than one WCET data entry describing it.

Once we calculate the WCET data of a function block it can be reused in any context, and needs to be recalculated only if the internals of the function block change.

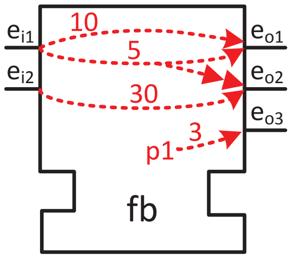


Figure 5.2: Example of WCET data.

Definition 7. *The WCET data for a function block f is represented by the function* $\textsc{wcet}(f)$*:*

Function block WCET data: $\textsc{wcet}(f) = \langle W_e, W_i \rangle$ *where*

W_e	*is a set of elements on the form $\langle e_i, W \rangle$, representing WCET information for input events.*
W_i	*is a multiset of elements on the form $\langle t, W \rangle$, representing WCET information for execution started by internal triggers;*
e_i	*is an input event, $e_i \in E_i$;*
t	*is the ID of an internal trigger;*
W	*is a set of WCET data entries.*

WCET data entry: $w = \langle v, o \rangle$ *where*

v	*is the WCET value, $v \in \mathbb{N}$;*
o	*is a function mapping output ports to the maximum number of events generated at them, $o \subseteq E_o \times \mathbb{N}$.*

Example

We can illustrate the WCET data for a function block by the following example:

$$\begin{aligned}\textsc{wcet}(fb) = \langle\{\ &\langle e_{i1}, \{\ \langle 10,\{e_{o1}{=}1\}\rangle,\\ &\qquad\qquad \langle\ 5,\{e_{o1}{=}2, e_{o2}{=}1\}\rangle\ \}\rangle,\\ &\langle e_{i2}, \{\ \langle 30,\{e_{o2}{=}1\}\rangle\ \}\rangle\ \},\\ &\{\ \langle p_1, \{\ \langle\ 3,\{e_{o3}{=}1\}\rangle\ \}\rangle\ \}\rangle\end{aligned}$$

The same data is also represented with a graphical notation in Figure 5.2. Each dashed arrow represents a WCET data entry. The start of the arrows denotes either the input event or the internal trigger that the entry is associated with, while the arrow end depicts the generated outputs. Output of multiple events is shown by a split in the arrow. Multiple outputs to a single event are represented by multiple arrow heads. The numbers next to the arrows give the WCET value of the alternative.

In the example, the o functions are shown as sets of equalities between ports and the number of generated events at that port, and for all output events that do not appear in the set the value of the function is 0. We can see that for input event e_{i1} we have two WCET data entries. One has a value of 10 and generates one event at the e_{o1} output port. The other one has a value of 5 but generates two events at e_{o1} and one event at the output port e_{o2}. Input event e_{i2} has only one data entry with a value of 30 and one event generated at the output port e_{o2}. The WCET data also contains one internal trigger with the ID p_1 and a single WCET data entry with the value 3 and one output at the e_{o3} port.

Operations on WCET data

For the purpose of our analysis we also need to define the following operations on the data elements:

Definition 8. *For the functions o, and $n \in \mathbb{N}$, we define the operations $+$, $*$ and inc as follows:*

$$\begin{aligned}
inc(o, e', n)\ (e) &= \begin{cases} o(e) + n & \text{if } e = e' \\ o(e) & \text{otherwise} \end{cases} \\
(o_1 + o_2)\ (e) &= o_1(e) + o_2(e) \\
(o * n)(e) &= n * o(e)
\end{aligned}$$

Definition 9. *For the sets of WCET data entries, W in the definition above, and for $n \in \mathbb{N}$, we define the following operations:*

$$\begin{aligned}
W * n &= \{\langle v * n, o * n\rangle \mid \langle v, o\rangle \in W\} \\
W_1 \otimes W_2 &= \begin{cases} W_1 & \text{if } W_2 = \emptyset \\ W_2 & \text{if } W_1 = \emptyset \\ \{\ \langle v_1 + v_2, o_1 + o_2\rangle \mid & \\ \qquad \langle v_1, o_1\rangle \in W_1 \ \wedge & \text{otherwise} \\ \qquad \langle v_2, o_2\rangle \in W_2 \quad \} & \end{cases}
\end{aligned}$$

5.2.2 Data normalization

As we have shown in Section 5.2.1, the WCET data that we define can contain more than one data entry for the same execution source (i.e. input event or internal trigger). When using such data sets in compositional analysis the amount of resulting data can grow rapidly, when all combinations of alternatives for all subcomponents must be considered. To address this problem we will use data normalization to remove redundant data entries and optimize large data sets by introducing over-approximations.

First, we introduce a comparison relation capturing when one WCET data entry is completely covered by another one.

Definition 10. *We define the following comparison relation between* $w_1 = \langle v_1, o_1 \rangle$ *and* $w_2 = \langle v_2, o_2 \rangle$*:*

$$\begin{aligned} w_1 \preceq w_2 &= v_1 \leq v_2 \wedge \forall e : o_1(e) \leq o_2(e) \\ w_1 \prec w_2 &= w_1 \preceq w_2 \wedge w_1 \neq w_2 \end{aligned}$$

In our current work we have defined two methods of data normalization, the *maximal elements method* and the *supremum method*, which we will now describe.

The maximal elements method

Using the maximal elements method we only remove redundant data from our WCET data sets. By this method we normalize a data set by keeping only entries which are maximal elements. Since all other elements are guaranteed to produce lower or equal WCET values in any context, and thus can be removed without loss of precision, this method allows reducing the number of data entries without introducing any overestimation in the normalization process.

Definition 11. *The maximal elements normalization function is defined as follows:*

$$\textsc{normalize}^{\max}(W) = \{w \mid w \in W \wedge \neg\exists w' \in W : w \prec w'\}$$

The Supremum Method

The second normalization method deals with incomparable WCET data values by replacing them by the supremum (the least upper bound), i.e.

the smallest value which is greater than both of them. The result of such normalization can be a drastic reduction of data and complexity of analysis, as all alternative execution paths for the input event or internally triggered activity are represented by a single WCET entry. However, this method also introduces an overestimation of the function block execution time and generated outputs. This overestimation can increase with each use of the normalization when applied to the whole function block hierarchy, and thus decrease the precision of the analysis results.

Definition 12. *The supremum normalization function is defined as follows:*

$$\textsc{normalize}^{\text{sup}}(W) = \{\sup(W)\}$$

Example

As an example of the two normalization methods, consider the following set of WCET data entries:

$$\begin{aligned} W = \{&\langle 10,\{e_{o1}{=}2\}\rangle,\\ &\langle\ 8,\{e_{o1}{=}1, e_{o2}{=}1\}\rangle,\\ &\langle\ 3,\{e_{o1}{=}2\}\rangle\} \end{aligned}$$

For this set, the two normalization methods give the following results:

$$\begin{aligned} \textsc{normalize}^{\text{max}}(W) = \{&\langle 10,\{e_{o1}{=}2\}\rangle,\\ &\langle\ 8,\{e_{o1}{=}1, e_{o2}{=}1\}\rangle\} \end{aligned}$$

$$\textsc{normalize}^{\text{sup}}(W) = \{\langle 10,\{e_{o1}{=}2, e_{o2}{=}1\}\rangle\}$$

The maximal elements normalization removes the third element since it is smaller than the first element, while the supremum method returns a single element that safely approximates all three.

5.2.3 Basic Function Block Analysis

Now that we have described the WCET data that we will use in the analysis and the operations for manipulating this data, we can present the actual analysis method. This section will describe the first part of our analysis method, the analysis of basic function blocks. Analysis of composite function blocks will be given in the Section 5.2.4.

The WCET of a basic function block is determined by analysis of its ECC. To gather the data about execution based on input events, we

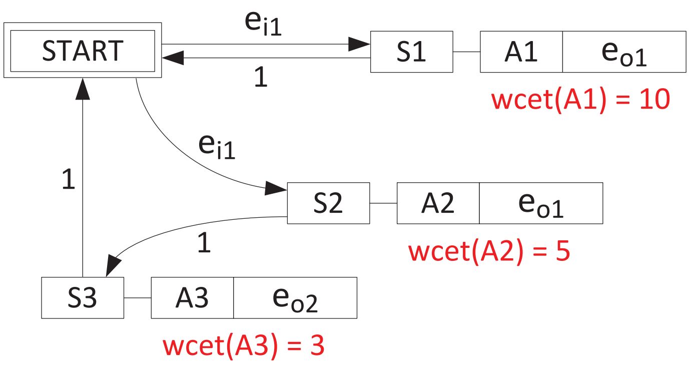


Figure 5.3: ECC analysis example.

must analyze all possible ECC runs that can be executed by the event input ports in the function block interface. As the execution of a basic function block can only start by receiving an input event, their WCET data only contains information in the event data set W_e, while the W_p set of information for internal triggers is always empty.

We start the basic function block analysis by going through the interface, and for each input event port we look for all transitions in the ECC guarded by the given event. For each such transition we go through all possible ECC runs, and for each run add together the WCET values of the algorithms to be executed and collect information about produced output events.

Example

We can illustrate how the ECC analysis is performed on the example shown in Figure 5.3. We will assume that the ECC in the figure is part of a basic function block bfb_1 containing only one input event port, e_{i1}, and two output event ports, e_{o1} and e_{o2}. Here we can see that the event e_{i1} can result in two different runs. The first run visits only state S1, executes algorithm A1 and produces an event on the output port e_{o1}. The second run visits states S2 and S3, executes algorithms A2 and A3, and produces events at both outputs e_{o1} and e_{o2}.

As the result of the analysis of bfb_1 we would get the following data:

$$
\begin{aligned}
WCET(bfb_1) = \langle \{ \langle e_{i1}, \{ & \langle 10, \{e_{o1}{=}1\} \rangle, \\
& \langle\ 8, \{e_{o1}{=}1, e_{o2}{=}1\} \rangle \} \rangle \}, \\
\emptyset \rangle &
\end{aligned}
$$

The BFB Analysis Algorithm

Algorithm 1 defines the analysis of a basic function block. The *for* loop starting on line 3 is used to iterate over all event inputs. The combination of the *for* loop on line 5 and the *if* statement on line 6 is used to find all transitions that are guarded with the given input event. Then, on line 7, we call the ECC analysis function (described in Algorithm 2) to collect analysis results for all ECC runs starting from the destination state of a transition. The results from the ECC analysis are added to a set W, which will in the end be associated with the currently analyzed input event and added to the result set W_e on line 11.

The ECC Analysis Algorithm

The ECC run analysis is described by Algorithm 2. In lines 1 to 6 we collect the WCET value v for all algorithms and output information o of actions defined for current ECC state. We use the combination of a

Algorithm 1 bfbAnalysis(bfb)

```
 1: $W_e \leftarrow \emptyset$
 2: $\langle\langle E_i, E_o\rangle, \langle Q, T\rangle, A\rangle \leftarrow bfb$
 3: for each $e_i \in E_i$ do
 4:    $W \leftarrow \emptyset$
 5:    for each $\langle q_s, e_g, q_d\rangle \in T$ do
 6:       if $e_g = e_i$ then
 7:          $W \leftarrow W \cup$ eccAnalysis($q_d$, $T$)
 8:       end if
 9:    end for
10:    $W \leftarrow$ normalize($W$)
11:    $W_e \leftarrow W_e \cup \{\langle e_i, W\rangle\}$
12: end for
13: return $\langle W_e, \emptyset\rangle$
```

Algorithm 2 $\textsc{eccAnalysis}(q,\ T)$

```
1: $v \leftarrow 0$
2: $o \leftarrow \emptyset$
3: for each $\langle a, e_o \rangle \in q$ do
4:     $v \leftarrow v + WCET(a)$
5:     $o \leftarrow \mathrm{inc}(o, e_o, 1)$
6: end for
7: $W' \leftarrow \emptyset$
8: for each $\langle q_s, e_g, q_d \rangle \in T$ do
9:     if $q_s = q$ and $e_g = 1$ then
10:        $W' \leftarrow W' \cup \textsc{eccAnalysis}(q_d,\ T)$
11:    end if
12: end for
13: if $W' = \emptyset$ then
14:    $W \leftarrow \{\langle v, o \rangle\}$
15: else
16:    $W \leftarrow \emptyset$
17:    for each $\langle v', o' \rangle \in W'$ do
18:        $W \leftarrow W \cup \{\langle v + v', o + o' \rangle\}$
19:    end for
20:    $W \leftarrow \textsc{normalize}(W)$
21: end if
22: return $W$
```

for loop and an *if* statement on lines 8 and 9 to find all ECC transitions starting with current state that do not have any event inputs as guards. We recursively start ECC analysis for destination states of such transitions. The results of recursive analysis are stored in a temporary WCET data entry set, W'. If there were no such results (i.e. no possible transitions), we assign the WCET value and output information for the current state as the final ECC analysis results on lines 13 and 14. Otherwise, we add the data for the current state to all WCET data entries collected by the recursive analysis of ECC in lines 16 to 20. Before the result data set W is returned, it is normalized on line 20.

5.2.4 Composite Function Block Analysis

Analysis of a composite function block consists of two separate parts: (a) analysis of execution based on input events and (b) analysis of the internal execution sources. Both of these parts are based on the analysis of the function block network contained in the composite function block. The algorithms presented in this section assume that the function block networks do not contain any cyclic execution paths. Section 5.3 describes how this restriction is lifted.

Input event execution analysis starts from the interface of the composite function block. Similar to the analysis of basic function blocks, for each input event we find all possible execution paths in the internal function block network. The analysis is performed on only one hierarchical level at a time. The execution paths are determined by traversing event connections of the network and by using event output information included in the existing WCET data for the function blocks in the network. For each execution path we accumulate the WCET values defined in the function block WCET data and gather information about produced output events if a path ends at one output event ports of the composite.

Analysis of the internal event sources is performed by iterating over all function blocks contained in the network and finding the ones which have at least one entry in their internal trigger WCET data set. For each such entry we start a network analysis based on the event output information of the entry.

Example

We will illustrate the analysis of composite function blocks by a simple example depicted in Figure 5.4. The figure shows a composite, *cfb*, containing three function blocks, and we assume the following WCET data for them:

$$\begin{aligned}\textsc{wcet}(\mathit{fb}_1) = \langle\{\ &\langle e_{i11}, \{\ \langle 1, \{e_{o11}{=}1, e_{o12}{=}2\}\rangle\ \}\rangle\ \},\\ &\emptyset\rangle\end{aligned}$$

$$\begin{aligned}\textsc{wcet}(\mathit{fb}_2) = \langle\{\ &\langle e_{i21}, \{\ \langle 10, \{e_{o21}{=}2\}\rangle,\\ &\qquad\qquad \langle 30, \{e_{o21}{=}1\}\rangle\ \}\rangle\ \},\\ &\emptyset\rangle\end{aligned}$$

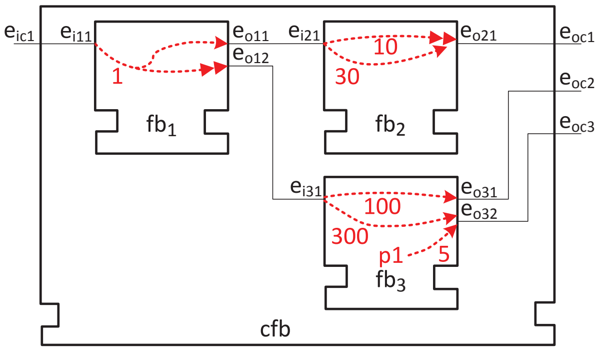


Figure 5.4: CFB analysis example.

$$\begin{aligned}\text{WCET}(\mathit{fb}_3) = \langle\{ \langle e_{i31}, \{ &\langle 100,\{e_{o31}{=}1\}\rangle, \\ &\langle 300,\{e_{o32}{=}1\}\rangle \}\rangle \}, \\ \{ \langle \ \ p_1, \{ &\langle \ \ \ 5,\{e_{o32}{=}1\}\rangle \}\rangle \}\rangle\end{aligned}$$

The analysis of composite *cfb* begins by determining WCET data for its single input event port e_{ic1}. Starting from this port, four different execution paths can be traced. The first one takes the single execution alternative with two outputs defined for fb_2, the first alternative for fb_2, and the first alternative for fb_3. As the execution of fb_1 generates two output at e_{o12}, all execution started by this output needs to be multiplied by two. The same is valid for the first execution alternative of fb_2. In total, the first execution path of *cfb* has WCET value 211 (1+10+2∗100), and generates two outputs to both e_{oc1} and e_{oc2}.

The second path in *cfb* takes the single alternative of fb_1, the second execution alternative of fb_2 and the first alternative of fb_3, resulting in WCET value 231, one output to e_{oc1} and two outputs to e_{oc2}.

Analogously to the first two execution paths, the second two execution paths are traced by taking the second alternative of $mathitfb_3$.

The internally triggered execution data for the *cfb* is calculated using the information about internal triggers of its subcomponents, in this case only fb_3. Because the e_{o32} event output which the internal trigger of fb_3 generates is connected directly to the output of the composite, the result will include just the WCET value of the internal trigger data entry (10) with its trigger ID p_1, and an output to e_{oc3}.

Assuming that we use the maximal element normalization method, which in this case does not remove any entries, the final result of the analysis is:

$$\begin{aligned}\textsc{wcet}(\mathit{cfb}) = \langle\{\langle e_{ic1}, \{\, &\langle 211, \{e_{oc1}{=}2, e_{oc2}{=}2\}\rangle, \\ &\langle 231, \{e_{oc1}{=}1, e_{oc2}{=}2\}\rangle, \\ &\langle 611, \{e_{oc1}{=}2, e_{oc3}{=}2\}\rangle, \\ &\langle 631, \{e_{oc1}{=}1, e_{oc3}{=}2\}\rangle\,\}\rangle\}, \\ \{\langle\ p_1, \{\langle\ \ 5, &\{e_{oc3}{=}1\}\rangle\,\}\rangle\,\}\rangle\end{aligned}$$

The supremum normalization would instead give:

$$\begin{aligned}\textsc{wcet}(\mathit{cfb}) = \langle&\{\langle e_{ic1}, \{\langle 631, \{e_{oc1}{=}2, e_{oc2}{=}2, e_{oc3}{=}2\}\rangle\,\}\rangle\}, \\ &\{\langle\ p_1, \{\langle\ \ 5, \{e_{oc3}{=}1\}\rangle\,\}\rangle\,\}\rangle\end{aligned}$$

The CFB Analysis Algorithm

The start of the composite analysis is given in Algorithm 3, where the separate execution of event based and internally triggered execution analysis is performed. The results attained by the two is then combination into a final WCET data set.

Algorithm 3 $\textsc{cfbAnalysis}(\mathit{cfb})$

```
1: $\langle \mathit{fbi}, \mathit{fbn}\rangle \leftarrow \mathit{cfb}$
2: $W_e \leftarrow \textsc{eventAnalysis}(\mathit{cfb})$
3: $W_i \leftarrow \textsc{triggerAnalysis}(\mathit{fbn})$
4: **return** $\langle W_e, W_p\rangle$
```

The CFB Event Analysis Algorithm

Algorithm 4 shows how we start the event analysis for a composite function block. In the *for* loop starting on line 3, we iterate through all event inputs of the composite. We start the network analysis algorithm (given in Algorithm 5) for each input and store the results of the analysis to the event WCET data set, linking it with the starting event input port.

The FBN Analysis Algorithm

As the algorithm for internally triggered execution analysis uses some concepts from the function block network analysis algorithm, the latter

Algorithm 4 eventAnalysis(cfb)

```
1: $W_e \leftarrow \emptyset$
2: $\langle\langle E_i, E_o\rangle, fbn\rangle \leftarrow cfb$
3: for each $e_i \in E_i$ do
4:     $W \leftarrow$ fbnAnalysis($e_i$)
5:     $W_e \leftarrow W_e \cup \langle e_i, W\rangle$
6: end for
7: return $W_e$
```

one is introduced first.

The function block network analysis described in Algorithm 5 starts from an input event port e of the composite or an output port of a function block instance and gathers WCET data for all execution paths that can be taken from that event port. The algorithm starts with initialization of the WCET data entry set W which will hold the results associated with the port e.

On line 3 we test if there is any connection leading out from the selected event port. If not, the resulting WCET data for this event is empty. Otherwise, the analysis continues at the destination port of the connection.

On line 7, we test if the destination port is an output port of the composite. If it is, we return a WCET data entry with WCET value 0 and a single output to the destination port as the only WCET data entry for the currently analyzed event.

If the connection's destination port is an input event port of a function block, we continue by first retrieving the WCET data for that function block, as shown on line 13.

As the function block WCET data can have multiple data entries (for multiple internal execution paths), we continue the analysis for each data entry separately by a *for* loop on line 15. On line 16 we initialize a temporary set W' which we will use to collect the intermediate results. The intermediate results will be added to the final result set on line 23.

With a recursive call of network analysis for each output event in the output information of the current WCET entry we gather information for execution paths started by these events, as shown on lines 17 and 18. The results of a single recursive call are stored in the W_r set, which is multiplied by the number of occurrences of the output event on line

19. On line 20 we construct all possible combinations of WCET data for execution paths started by the current output event (W_r) with the ones already gathered in the temporary set W', and use these combinations as our new temporary set. By this we have created all possible execution paths that can be taken by generating all output events of the currently examined WCET data entry.

Once we have collected the data for all possible execution paths we add the WCET value of the current data entry to all the data in the

Algorithm 5 FBNANALYSIS(e, fbn)

```
1:  $W \leftarrow \emptyset$
2:  $\langle F_i, C\rangle \leftarrow fbn$
3:  **if** $\neg\exists\langle e_s, e_d\rangle \in C : e_s = e$ **then**
4:      **return** $\emptyset$
5:  **else**
6:      Let $\langle e_s, e_d\rangle \in C$ be the connection for which $e_s = e$
7:      **if** $e_d$ is an output port **then**
8:          $o \leftarrow inc(\emptyset, e_d, 1)$
9:          $W \leftarrow \langle 0, o\rangle$
10:         **return** $W$
11:     **end if**
12:     Let $f$ be the FB to which $e_d$ belongs
13:     $\langle W_e, W_i\rangle \leftarrow$ WCET$(f)$
14:     Let $\langle e_i, W_t\rangle$ be the element in $W_e$ for which $e_i = e_d$
15:     **for each** $\langle v, o\rangle \in W_t$ **do**
16:         $W' \leftarrow \emptyset$
17:         **for each** $e_o \in E_0 : o(e_o) > 0$ **do**
18:             $W_r \leftarrow$ FBNANALYSIS$(e_0,\ fbn)$
19:             $W_r \leftarrow W_r * o(e_o)$
20:             $W' \leftarrow W' \otimes W_r$
21:         **end for**
22:         $W' \leftarrow \{\langle v' + v, o'\rangle : \langle v', o'\rangle \in W'\}$
23:         $W \leftarrow W \cup W'$
24:     **end for**
25:     $W \leftarrow$ NORMALIZE$(W)$
26:     **return** $W$
27: **end if**
```

temporary set W' in line 22. We can now add this temporary data set to our final data set W, as can be seen in line 23. The final analysis results are normalized in line 25.

The internal trigger execution analysis algorithm

Analysis of WCET for internally triggered execution inside a function block network is described by Algorithm 6. The algorithm starts with preparing an empty set W'_p which will hold our result.

The *for* loops on lines 3 and 5 iterates through all function blocks in the network, and their internal trigger WCET data if it is defined. For each such data entry a temporary empty data set W' is created.

Lines 7 to 16 contain the same recursive analysis of all possible execution paths that can be started using the currently analyzed internal trigger WCET entry, as already described in the function block network

Algorithm 6 TRIGGERANALYSIS(*fbn*)

```
 1: $W'_t \leftarrow \emptyset$
 2: $\langle F, C\rangle \leftarrow fbn$
 3: for each $f \in F$ do
 4:    $\langle W_e, W_i\rangle \leftarrow$ WCET($f$)
 5:    for each $\langle t, W\rangle \in W_i$ do
 6:       $W' \leftarrow \emptyset$
 7:       for each $\langle v, o\rangle \in W$ do
 8:          $W'' \leftarrow \emptyset$
 9:          for each $e_o \in E_o : o(e_o) > 0$ do
10:             $W_r \leftarrow$ FBNANALYSIS($e_o$, $fbn$)
11:             $W_r \leftarrow W_r * o(e_o)$
12:             $W'' \leftarrow W'' \times W_r$
13:          end for
14:          $W'' \leftarrow \{\langle v'' + v, o''\rangle : \langle v'', o''\rangle \in W''\}$
15:          $W' \leftarrow W' \cup W''$
16:       end for
17:       $W' \leftarrow$ NORMALIZE($W'$)
18:       $W'_t \leftarrow W'_t \cup \langle t, W'\rangle$
19:    end for
20: end for
21: return $W'_t$
```

analysis. After the results collected in the temporary data set W' are normalized on line 17, they are added to the final result data set on line 18. They are linked to the trigger ID defined for the internal trigger WCET data entry that was the origin of execution paths collected in the temporary data set.

5.3 Handling cyclic execution paths

The timing analysis presented in the previous section was restricted to function block networks which do not contain cyclic execution paths. Such cyclic paths can, for example, be used to implement functionalities such as aggregation of sensor data, or iterative algorithms. This section presents an extension to the previously defined function block network analysis, which allows this analysis to be performed on networks containing cyclic execution paths.

The support for analysis of cyclic execution paths consists of three parts. First, it is specified how to define limits to cycle iterations using cycle bounds. Having defined such bounds, the function block network analysis algorithm is extended to allow analysis of bounded cycles. To still support the compositional analysis approach and treatment of function blocks as black boxes, a method for propagation of cycle bounds through function block hierarchy is provided. The following sections give a detailed description of these three parts.

5.3.1 Cycle bound definition

Performing WCET analysis on code containing program loops is possible if the number of loop iterations has an upper limit – a loop bound. To allow for analysis of cyclic paths in component-based models we will use a similar approach and introduce cycle bounds to the model. Cycle bounds are defined by annotating elements of the component model. These annotations will be used by the WCET analysis to determine the maximum number of cycle iterations when the annotated element is part of a cycle. Cycle bound annotations defined in this work are only used by the WCET analysis, and have no impact on system execution. When developing composite components, cycle bounds can also be defined on elements which are not contained by a cycle. Although in this case the cycle bound will not be used when analyzing the composite in isolation,

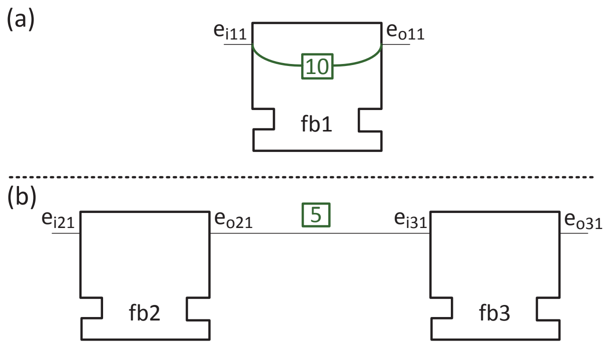


Figure 5.5: Example of graphical representation of (a) a component cycle bound and (b) a connection cycle bound.

it may be used when the cycle is formed on a higher level of hierarchy.

The approach defines two different types of cycle bounds: *component cycle bounds* and *connection cycle bounds.* The two are described in detail in the following sections.

Component cycle bounds

Component cycle bounds are defined by the component developer as annotations to the component interface. They are used to describe internal mechanisms that a component implements to limit the number of cyclic iterations. Because the iteration limit is a result of the component's internals, it is independent of the context that the component is used in, and can be reused together with the component.

Each component cycle bound is defined between one input and one output event port. The value of the bound represents the maximum number of times an execution started at the input port will result in an event at the output port if the two are used in a cycle.

The graphical notation used for component cycle bound can be seen in Figure 5.5 (a). The bound for component *fb1* between ports e_{i11} and e_{o11} with value 10 is represented by a green connection between the ports, containing the value of the bound inside a square.

When coupled with WCET data definition for components, component cycle bounds can give special semantics to WCET execution alternatives of the data. These semantics are applied to the alternatives which start with an input port contained by a cycle bound. If an execution alternative starts with the input of the bound, and produces an event at the output port of the bound, during the analysis it will be treated as a cycle-forming alternative. If an execution alternative contains only the input port of the bound, but does not produce an event at the output port of the bound, it will be treated as an exit alternative.

Definition 13. *To be able to store component cycle bounds together with WCET data for components, the component WCET data function defined in Section 5.2.1 is extended with cycle bound information:*

Function block WCET data: $WCET(f) = \langle W_e, W_p, CB_{fb} \rangle$ *where*

W_e, W_p	*are previously defined sets of WCET elements.*
CB_{fb}	*is a set of elements in the form* $\langle e_{icb}, e_{ocb}, b \rangle$*, representing component's cycle bounds.*
e_{icb}	*is an input event,* $e_i \in E_i$*.*
e_{ocb}	*is an output event,* $e_o \in E_o$*.*
b	*is the cycle bound value,* $b \in \mathbb{Z}^+$*.*

Connection cycle bounds

Connection cycle bounds are defined by component integrators as annotations on connections between components. These bounds denote that the number of cyclic iterations is limited by the interaction of multiple components. Opposed to the component bounds, connection bounds are specific to the component network, and are not aimed to be reused. They can however be propagated to component bounds of a composite and reused in this form, as we will describe in Section 5.3.3.

Connection cycle bounds can only be defined for event connections, as the data connections do not transfer control flow in IEC 61499. The values of a connection cycle bound represents the maximum number of times the connection will be traversed if it is part of a cycle.

Figure 5.5 (b) shows how connection cycle bounds are represented in the model, where a connection bound with value 5 is defined as an annotation for the connection between ports e_{o21} and e_{i31}.

Definition 14. *As connection cycle bounds are a property of each individual connection, the definition of a connection presented in Section 5.1 is extended with bound information. The new connection definition allows each connection to also represent a cycle bound:*

Connection: $c = \langle e_s, e_d, b \rangle$ *where*

e_s	*is the event port used as the source of the connection;*
e_d	*is the event port used as the connection target;*
b	*is the cycle bound value, $b \in \mathbb{N} \cup 0$, where 0 represents connections with no cycle bound.*

5.3.2 Cycle analysis

With the ability to define cycle bound annotations on model elements, and thus defining a limit to the number of cycle iterations, we can extend the standard WCET analysis algorithms to allow analysis of systems containing cycles. Cycle analysis consists of three separate stages: (i) cycle discovery, (ii) isolated cycle analysis, and (iii) merging of cycle analysis results with the results of the standard WCET analysis. The following sections give details of these three stages, while also describing them on the example depicted in Figure 5.6.

Cycle discovery

Because cycle bounds can be defined on two different types of model elements, components and connections, the cycle discovery is also implemented in two parts of the network analysis algorithm – whenever an event connection between two components is considered, and on each usage of component WCET data. When a cycle bound definition is found during network analysis, the network is traversed in search of an analyzable cycle which contains the bound. If no such cycle is found, the the bound definition is disregarded, and the cycle analysis and merging stages omitted. Such bound definitions can however still be used as a candidate for hierarchical cycle bound propagation, described in Section 5.3.3. If the bound is included in more than one event cycle, cycle analysis will give an error, as the bound can be applied to only one of them, leaving the other unbound. In case exactly one event cycle for the bound is found, we can proceed with the isolated cycle analysis.

Cycle discovery is depicted in Figure 5.6 (i). When applying analysis to the application, a component cycle bound is detected between the two

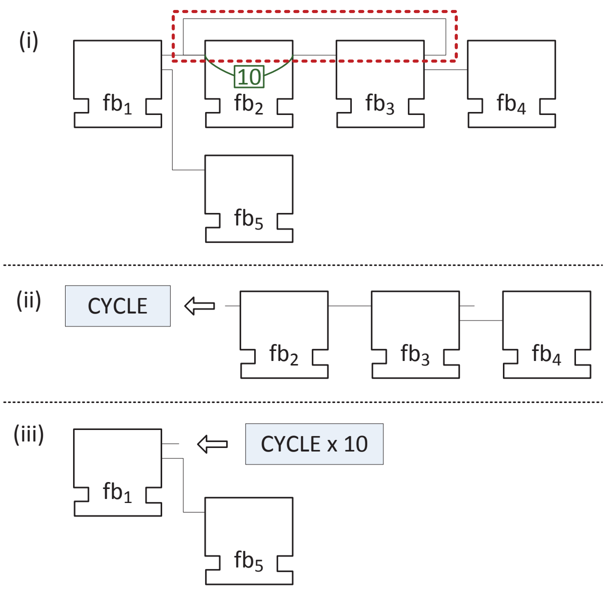


Figure 5.6: The cycle analysis approach depicted on an example system. A component cycle bound with value 10 is shown between two ports of component fb_2.

ports of function block fb_2. As there also exists a cyclic execution path which crosses the bound, spanning function blocks fb_2 and fb_3, isolated cyclic analysis is applied.

Isolated cycle analysis

The algorithm that implements the actual WCET analysis of a cycle uses a modified version of the standard network WCET analysis algorithm described in Section 5.2.4. The network analysis is extended with a stack of currently analyzed cycles. For each cycle detected by cycle discovery, a new instance of network analysis is started form the beginning of the cycle, while adding the cycle definition to the top of the stack. By starting a new instance of analysis for each cycle, the cycle paths are isolated from the analysis performed for the rest of the component network. The cycle definition stack is used to break the connections that lead from the end of a cycle back to its beginning. Using a stack to store information about currently analyzed cycles allows starting the cycle analysis recursively, thus providing the ability to analyze multiple nested cycles. When the analysis reaches the end of the cycle that is currently at the top of the stack, the analysis for the current top-most cycle is stopped, the cycle definition is remove from the stack, and the obtained results for the isolated cycle are temporarily stored. In case that during cycle analysis the cycle discovery detects a cycle which is already in the analysis stack, and is not the top-most cycle, the analysis will report an error. In this way we detect combinations of cycles which would result in infinite recursions.

Figure 5.6 (ii) shows isolated cyclic analysis. As the cycle bound is defined for fb_2, the connection leading from fb_3 to fb_2 is broken. The isolated analysis is performed starting from the input port of fb_2, and the results are stored temporarily. It should be noted that function block fb_4 is also included in isolated cycle analysis, although it is not part of the path forming the cycle. This is because fb_4 is triggered in each iteration of the cycle, and therefore is a part of the cyclic execution.

Merging cycle analysis results

Once the isolated analysis of a cycle is finished, the obtained results can be merged with the standard network analysis. The cycle results are first multiplied by the value of the cycle bound. The multiplied results, together with possible results for exit alternatives, are then added to the cumulative results of the network.

The standard analysis of the execution paths covered by the cycle bound is skipped. If the analyzed cycle bound was defined for a component, the standard network analysis is continued using cycle exit alternatives of the WCET data of the component, i.e. alternatives that do not produce outputs to the output port of the component cycle bound.

Merging of cycle analysis results is shown in Figure 5.6 (iii). The results of isolated cycle analysis are multiplied by the bound value, 10, and returned to the standard analysis. The standard analysis then continues by analyzing execution of function block fb_5.

Example

We demonstrate the cycle analysis method on an example composite function block which is used to filter sensor input noise by providing a

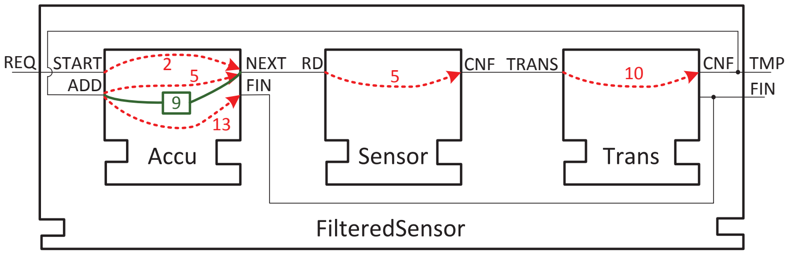


Figure 5.7: Composite component used in the example of cycle analysis.

mean value out of ten sensor readings. The composite and its internal component network is shown in Figure 5.7. The *Accu* component is used to accumulate ten readings using an internal counter which is reset when an event is received at the *START* port. Resetting the counter also results in an event at the *NEXT* port, signaling that a new value should be read. The *Sensor* component reads the actual sensor value, and the *Trans* component transforms and normalizes the raw sensor input. The transformed (non-accumulated) sensor data is sent outside of the composite by an event at the *TMP* output port of the composite, and also back to the *Accu* component. An event at the *ADD* port of *Accu* adds the current value to the accumulator and increments the internal counter. Depending on the state of the counter an event is generated at either the *NEXT* port again (starting another iteration of the cycle) or at the *FIN* port (exiting the cycle).

While explaining how the example composite is analyzed we will refer to the intermediate and final results of the analysis shown in Table 5.1.

The analysis starts from the *REQ* port of the composite, and following the execution path collects the temporary WCET value of 17 and one output to *TMP* for the initial execution of the three components. This temporary WCET result is shown in Table 5.1 as Step 1.

When the standard analysis algorithm arrives to the *ADD* port the cycle discovery algorithm detects the cycle bound with value 9 between this port and the port *NEXT*. The discovery algorithm then performs a test to determine if the bound is a part of an analyzable cycle. Since it is, the isolated cycle analysis is triggered.

The isolated cycle analysis starts by adding the cycle bound to the cycle analysis stack, and proceeds with analysis of the network starting

Table 5.1: Intermediate and final results of the analysis example.

Step	Analysis/result type	WCET	Outputs
1	Standard	17	$TMP = 1$
2	Isolated cycle	20	$TMP = 1$
3	Multiplied cycle	180	$TMP = 9$
4	Cycle exit	13	$FIN = 1$
5	Final (1+3+4)	210	$TMP = 10$, $FIN = 1$

from *ADD* port, using the cycle-forming execution alternative in *Accu* with WCET value 5. As the cycle execution path is traced through all three components, collecting the cycle WCET value of 20 and one output to *TMP*. At this point the analysis reaches the *ADD* port again, and because the cycle bound for that port is on top of cycle analysis stack, the isolated cycle analysis is stopped. The results of this analysis are shown in Table 5.1 as Step 2. After the isolated analysis of the cycle is finished, the results are multiplied with the value of the cycle bound, resulting in the values shown as Step 3 in Table 5.1.

The analysis continues with the cycle-exit alternative of the *AND* port, which has WCET value 13 and generates an output to the *FIN* output port. The cycle-exit results are shown as Step 4 in Table 5.1. The multiplied cycle results are added to the cycle-exit results and combined with the temporary WCET result. The final results for the composite are shown in Table 5.1 as Step 5.

The cycle analysis algorithm

The cycle analysis algorithm is defined by two separate parts. First, a function for starting isolated cycle analysis is defined. After that, the extended function block network analysis, including the cycle detection, invocation of isolated cycle analysis, and merging of isolated cycle analysis results, is given.

The starting of isolated cycle analysis is presented in Algorithm 7. As arguments, the cycleAnalysis function takes the cycle bound *cb* for which the analysis should be started, and the function block network *fbn* which is the context of the analysis. The *if* statement on line 2 checks if the current cycle bound is on top of the cycle analysis stack. If this is true, the isolated analysis has reached the end of the cycle, and

Algorithm 7 $\textsc{cycleAnalysis}(cb,\ fbn)$

```
1: $\langle e_{scb}, e_{ecb}, b\rangle \leftarrow cb$
2: if $cb$ is on top of the cycle stack then
3:     return $\emptyset$
4: else
5:     push $cb$ to the cycle stack
6:     $W_c \leftarrow \textsc{fbnAnalysis}(e_{ecb},\ fbn)$
7:     pop $cb$ from the cycle stack
8:     return $W_c$
9: end if
```

an empty result is returned on line 3. If the cycle bound definition is not at the top of the cycle analysis stack, it is first put there by the *push* operation on line 5. On line 6, a new instance of function block network analysis is started from the cycle bound end port, giving the results for the cyclic path. Next, on line 7 the currently analyzed cycle bound is removed from the cycle analysis stack In the end, line 8 returns the isolated cycle analysis result.

The new function block network analysis, extended with analysis of cyclic execution paths, is given in Algorithm 8. The changes to the original algorithm, presented in Algorithm 5, are highlighted by red text color. Detection of connection cycle bounds is located on line 12. In case a cycle bound for the current connection exists, isolated cycle analysis for that bound is performed on line 13, multiplied by the bound value on line 14, and returned as the result of analysis of current sub-network on line 15.

Detection and handling of component cycle bounds is performed in two separate parts of the algorithm. In this way it is way, most of the

Algorithm 8 $\textsc{fbnAnalysis}(e,\ fbn)$

```
1: $W \leftarrow \emptyset$
2: $\langle F_i, C\rangle \leftarrow fbn$
3: if $\neg\exists\langle e_s, e_d, b\rangle \in C : e_s = e$ then
4:     return $\emptyset$
5: else
6:     Let $\langle e_s, e_d, b\rangle \in C$ be the connection for which $e_s = e$
```

Algorithm 8 FBNANALYSIS(e, fbn) – continued

```
 7:     if $e_d$ is an output port then
 8:         $o \leftarrow inc(\emptyset, e_d, 1)$
 9:         $W \leftarrow \langle 0, o \rangle$
10:         return $W$
11:     end if
12:     if $b > 0$ then
13:         $W \leftarrow$ CYCLEANALYSIS($\langle e_s, e_d, b \rangle$, $fbn$)
14:         $W \leftarrow W * b$
15:         return $W$
16:     end if
17:     Let $f$ be the FB to which $e_d$ belongs
18:     $\langle W_e, W_p, CB_{fb} \rangle \leftarrow$ WCET($f$)
19:     Let $\langle e_i, W_t \rangle$ be the element in $W_e$ for which $e_i = e_d$
20:     for each $\langle v, o \rangle \in W_t$ do
21:         $W' \leftarrow \emptyset$
22:         for each $e_o \in E_0 : o(e_o) > 0$ do
23:             if $\exists cb = \langle e_{icb}, e_{ocb}, b \rangle \in CB_{fb} : e_{icb} = e_i \wedge e_{ocb} = e_o$ then
24:                 $W' \leftarrow W' \cup$ CYCLEANALYSIS($cb$, $fbn$)
25:             else
26:                 $W_r \leftarrow$ FBNANALYSIS($e_0$)
27:                 $W_r \leftarrow W_r * o(e_o)$
28:                 $W' \leftarrow W' \otimes W_r$
29:             end if
30:         end for
31:         $W' \leftarrow \{\langle v' + v, o' \rangle : \langle v', o' \rangle \in W'\}$
32:         if $\exists cb = \langle e_{icb}, e_{ocb}, b \rangle \in CB_{fb} : e_{icb} = e_i \wedge o(e_{ocb}) > 0$ then
33:             $W' \leftarrow W' * b$
34:         end if
35:         $W \leftarrow W \cup W'$
36:     end for
37:     $W \leftarrow$ NORMALIZE($W$)
38:     return $W$
39: end if
```

code for analysis of normal and cycle-forming execution alternatives is reused. On lines 23 and 24, the standard network analysis is extended to detect component cycle bounds, and start isolated cycle analysis for output the event of a cycle-forming alternative which actually forms the cyclic path. The task of the statements on lines 26 to 28 is twofold. In case that the current execution alternative is cycle-forming, these lines perform analysis of output events which are generated as a part of a cycle, but do not form the cyclic path. If the current execution alternative is not a cycle-forming one, these lines perform standard network analysis.

The second part of component cycle bound analysis starts on line 32, where it is tested if the current execution alternative is a cycle-forming one. In this case the temporary result W' contains results of cycle analysis, and it is multiplied by the bound value on line 33. Because the temporary result W' is used to store both standard and cycle analysis results, the statement on line 35, which collects standard analysis results, also merges the results of cycle analysis.

5.3.3 Hierarchical propagation of cycle bounds

The WCET analysis method we extend in this work takes advantage of the component-based development approach, and performs the analysis in a compositional manner. Analysis of each component is performed in isolation, and only on one hierarchical level. Analysis results for a component are stored with the component, and reused together with it. However, cyclic execution paths can span over multiple levels of hierarchy. As a result, there can be a situation in which the mechanism that limits the cycle is inside a composite component, while the actual cycle is formed outside the composite, on a higher level of hierarchy. In this case the cycle bound will be defined on a model element that is inside of the composite, and will not be visible on the composites component's interface, which causes a problem for the compositional WCET analysis. To still support the compositional approach to analysis, we have to be able to represent cycle bounds defined inside composites as cycle bounds on the level of the composites' interfaces. We do this by propagation of cycle bounds.

Both component and connection cycle bounds can be propagated to higher levels of hierarchy. However, as by propagation we define representing composite's internal bounds on the level of it's interface, results are always component bounds.

Cycle bound propagation method

The cycle bound propagation starts by searching the internal component network of a composite and finding bounds which are candidates for propagation. For a bound to be a candidate for propagation, it must not be a part of an event cycle inside the composite. If it is a part of an event cycle already, the bound is treated as consumed, since in this case there is no guarantee that the mechanism that implements the bound will still work if contained by a new cycle outside of the composite. However, if the bound is not a part of a cyclic path in the composite's network, the bound mechanism can be utilized when the cycle is formed on a higher level of hierarchy.

Once the propagation candidate bounds have been found, they are tested for propagation to each pair of one input and one output port of the composite. To propagate a candidate bound, the port pair has to satisfy two requirements. First, there must be at least one path between the two ports in the internal network of the composite. Second, all paths between the port pair must traverse the candidate bound. If a combination of a port pair and a candidate bound that satisfies the two requirements is found, a new component cycle bound definition for the composite can be defined between the input and output port pair, with the same value as the candidate bound. This bound definition can then be used in any component network which contains the composite, without the need to reanalyze it.

Example

We will demonstrate cycle bound propagation on the example shown in Figure 5.8. The composite in this example is a modified version of the one used to demonstrate cycle analysis (shown in Figure 5.7). The new composite, shown in Figure 5.8 (a), encapsulates the filtering functionality of the previous composite, and exposes it as a reusable component. This is done by removing the sensor component from the function block network, and allowing it to be connected on the next level of hierarchy to the *S_CNF* and *S_RD* ports on the composite interface. The usage of the filter composite is depicted in Figure 5.8 (b). To be able to still perform compositional analysis on the new system, the cycle bound defined in *Accu* component needs to be propagated to the *Filter* composite.

Examining the *Filter* composite for bound propagation candidates identifies the cycle bound defined in the *Accu* component, as it is not

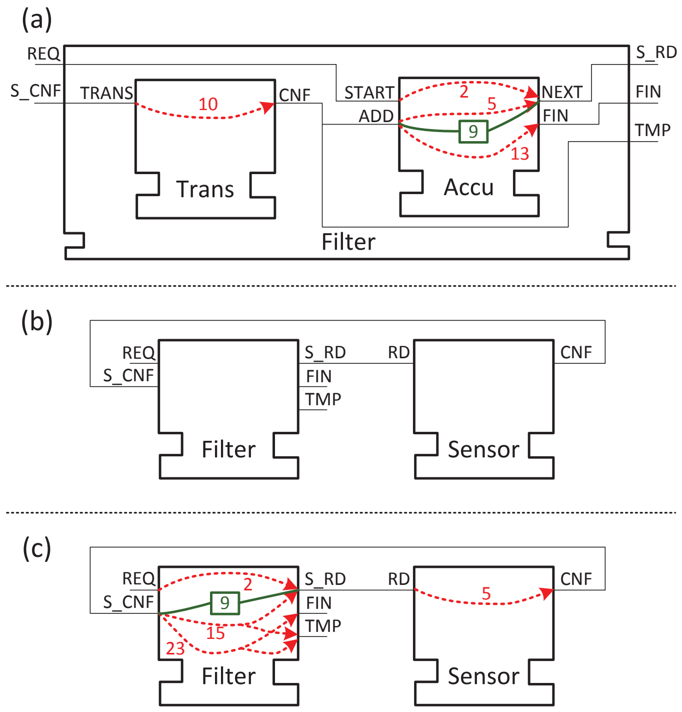


Figure 5.8: (a) Filter composite component used for exemplifying cycle bound propagation. (b) An instance of the composite used to filter sensor data. (c) Results of WCET analysis and cycle bound propagation for the composite and the sensor component.

contained by a cycle in the internal composite network. As the candidate bound is traversed by the only path (and thus also all paths) between ports *S_CNF* and *S_RD*, it can be propagated from the *Accu* component to the *Filter* composite. A new bound will be defined between ports *S_CNF* and *S_RD*, and the value of the bound will be 9. The combined results of WCET analysis and bound propagation for *Filter* are shown as annotations to its interface in Figure 5.8 (c).

The cycle bound propagation algorithm

Definition of the algorithm for propagation of cycle bounds requires means to find all execution paths between a pair of event ports. The first step to this is to define a function which can be used to detect if two event ports are connected by a single connection or a single function block.

Definition 15. *The function* CONNECTED *returns true if an event signal can be transferred from event port e_s to event port e_e, through either a single connection or a single function block, in function block network fbn:*

$$\begin{aligned}\textsc{connected}(e_s,\ e_e,\ \mathit{fbn}) \quad = \quad & \langle e_s, e_d, b\rangle \in C \vee \\ & \exists f \in F : \langle W_e, W_i\rangle = \textsc{wcet}(f) \wedge \\ & \exists \langle e_s, W\rangle \in W_e : \\ & \exists \langle v, o\rangle \in W : o(e_e) > 0\end{aligned}$$

where

fbn	*is a function block network in form $\langle F, C\rangle$.*

Using the previously defined function, a function for retrieving execution paths from a function block network can be defined.

Definition 16. *The function* GETPATHS *returns a set of execution paths $\{p_0, \ldots, p_n\}$ execution paths between a starting event port e_s and an ending event port e_e, from a function block network fbn:*

$$\begin{aligned}\textsc{getPaths}(e_s, e_e, \mathit{fbn}) \quad = \quad & \{p \mid p = \langle e_0, \ldots, e_n\rangle \wedge e_0 = e_s \wedge e_n = e_e \wedge \\ & \forall 0 \leq i < n : \textsc{connected}(e_i,\ e_{i+1}, \mathit{fbn}) \wedge \\ & \forall 0 \leq i, i < m : e_i \neq e_j \vee i = j\}\end{aligned}$$

The final prerequisite for defining the cycle bound propagation algorithm is a function which can be used to test if an execution path traverses a cycle bound.

Definition 17. *The function* TRAVERSES *returns* true *if an event path p contains a cycle bound cb:*

$$\textsc{traverses}(p, cb) = \begin{cases} \mathit{true} & \text{if } \exists e_i, e_{i+1} \in p : e_i = e_s \wedge e_{i+1} = e_d \\ \mathit{false} & \text{otherwise} \end{cases}$$

where

p	*is a path in form $\langle e_0..e_n\rangle$;*
cb	*is a cycle bound definition in form $\langle e_s, e_d, b\rangle$.*

Algorithm 9 PROPAGATEBOUNDS(cfb)

```
1: $\langle fbi, fbn \rangle \leftarrow cfb$
2: $\langle E_i, E_o \rangle \leftarrow fbi$
3: Let $CB$ be a set of all cycle bounds defined for connections and
   function blocks of $fbn$
4: $CB_p \leftarrow \emptyset$
5: for each $e_i \in E_i, e_o \in E_o$ do
6:    $P = $ GETPATHS($e_i, e_o, fbn$)
7:    if $\exists cb \in CB : \forall p \in P :$ TRAVERSES($p, cb$) then
8:       $\langle e_{cbs}, e_{cbd}, b \rangle \leftarrow cb$
9:       $CB_p \leftarrow CB_p \cup \langle e_i, e_o, b \rangle$
10:   end if
11: end for
12: return $CB_p$
```

Having the definition the functions to retrieve execution paths from function block networks, and test if execution paths contain cycle bounds, the formal definition of cycle bound propagation for composite function blocks can be defined. This definition is given by Algorithm 9.

The algorithm initiates a set CB_p, which will contain the cycle bounds to be propagated to the composite, on line 4. On line 5, the algorithm iterates through all combinations of input event ports e_i and output event ports e_o of the composite's interface. All execution paths between the current combination of input and output events are retrieved on line 6. The *if* statement on line 7 checks if there exists a cycle bound cb, which all retrieved execution paths traverse. If such a cycle bound exists, a new cycle bound is created and added to the list of cycle bounds to be propagated, on line 9. The new bound contains the input and output event ports of the composite, and the bound value of the cycle bound cb. As a result, the algorithm returns the list of cycle bound that can be defined for the composite on line 12.

5.4 Analysis using hardware-specific models

The WCET analysis described in previous sections was performed only using platform-independent models, and did not take into account that WCET information for algorithms and function blocks depends on the type of the device they are deployed to. In this section previously defined analysis is extended to utilize the model of function block deployment, and two new methods are introduced: *device-specific application WCET analysis* and *device utilization analysis*.

The section continues by first extending the previously defined analysis with device-specific WCET information, and then describing the two new analysis methods.

5.4.1 Device-specific WCET analysis

As algorithms can have different execution time when running on different types of hardware devices, providing timing analysis that takes into account deployment requires the function block analysis presented in Section 5.2 to take into account to which device a function block is deployed. In this section we present how the previously defined WCET data and analysis algorithms are extended to take into account deployment information.

To allow describing device-specific WCET for algorithms and function blocks, the two WCET functions presented in Section 5.2.1 are extended to allow returning different values for an algorithm or a function block for different devices. This is done by adding an argument representing a specific device. The rest of the function definition is kept the same. The definition of the new functions follow:

Definition 18. *The WCET data for an algorithm $a \in A$ or a function block fb, specific for a device d, are represented by the function* $\textsc{wcet}(a,\ d)$ *and* $\textsc{wcet}(fb,\ d)$*, respectively:*

Algorithm WCET data: $\textsc{wcet}(a,\ d) \in \mathbb{N}$

Function block WCET data: $\textsc{wcet}(f,\ d) = \langle W_e, W_i \rangle$

The same principle used for WCET data is also used to extend function block analysis algorithms. Their list of arguments is in the same way extended with device information, which is used when retrieving WCET data values. Because the changes in the algorithms are minor, they will not be explicitly presented.

One of the consequences of using device-specific WCET information for function blocks is that results of analysis can be reused only in the context of the device for which the analysis was performed. This means that analysis for a single function block has to be performed multiple times, once for each device.

5.4.2 Application Analysis

The method presented in this section uses the previously defined analysis, and applies it to the application software model and the model of deployment to calculate device-specific WCET for applications. The rest of the section first defines how device-specific WCET is represented, and then describes the details of the analysis method.

Device-specific application WCET data

Applications are the top-most elements in the hierarchy of IEC 61499 software models which usually provide functionality which is highly specific to a concrete system. They are not intended for reuse, and therefore do not have an explicit interface. Because of this, compared to the WCET data for function blocks, the WCET data describing applications does not have to be context-independent. Lack of an interface, and therefore input event ports, means that the WCET data for applications can consist of only a set of entries describing execution initiated by internal triggers. Also, as there are no generated output event ports to consider, each WCET data entry for an application can be described by just a single WCET value. Since there will always be a maximum of these values, a clear worst case, each internal trigger can be described by only one WCET data entry. The following definition provides formal description of device-specific WCET data.

Definition 19. *The WCET data for an application app, specific for a device d, is represented by the function* $\textsc{wcet}(app,\ d)$*:*

Application WCET data: $\textsc{wcet}(app,\ d) = S \quad \textit{where}$

S	*is a set* $\{s_0, \ldots, s_{\lvert S\rvert}\}$ *of application WCET data entries;*
s_i	*is a WCET data entry in form of* $\langle t, v \rangle$*;*
t	*is the ID of an internal trigger;*
v	*is the WCET value,* $v \in \mathbb{N}$*.*

Application analysis method

The device-specific WCET application analysis is performed using same principles as the internal trigger analysis for composite function blocks. The function block network is scanned for function blocks containing internal execution triggers. For each such trigger, an analysis of function block network is performed.

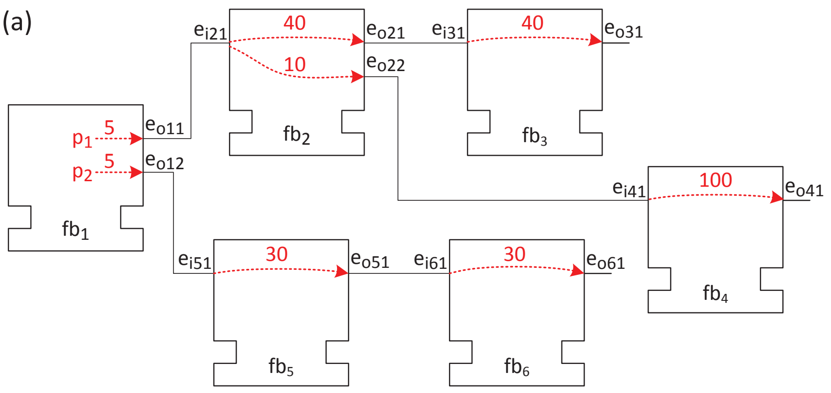
Compared to function block network implementing composites, where all function blocks have to execute on the same device, each function block on the application level can be deployed separately. To analyze WCET of an application for a specific device, the analysis method needs to consider only execution time of function blocks deployed to that device. Compared to the function block network analysis for composites, the network analysis for applications ignores WCET values for function blocks not deployed to the currently analyzed device. However, the part of function block WCET data containing information about generated outputs is still used. In this way, all possible execution paths are still explored, but the results for each device contain only execution paths that produce maximal WCET value for that device, rather than ones that lead to maximal overall WCET in the whole system.

Example

We will show an example of application analysis using the application depicted in Figure 5.9 a). The example platform consists of two devices, d_1 and d_2, and for the purpose of simplicity, we will assume that both devices are of the same type. In the example we will explore two different deployment alternatives specified in Figure 5.9 b).

In case of deployment alternative A, for device d_1 there will be two WCET entries. The first one is for execution trigger p_{11}. Starting from this trigger, two execution paths can be traced. The first one takes the first execution alternative of fb_2 and the only alternative of fb_3. The second one takes the second alternative of fb_2, and the only execution alternative of fb_4. As the latter execution path has greater WCET value,

(a)



(b)

Deployment	Device	Function blocks
A	d_1	fb_1, fb_2, fb_3, fb_4
	d_2	fb_5, fb_6
B	d_1	fb_1, fb_2, fb_3, fb_5
	d_2	fb_4, fb_6

Figure 5.9: a) Application example. b) Function block deployment information.

and there is no need to consider generated outputs, only the WCET value of this path is assigned to p_{11}.

The second entry for device d_1 only takes into account the alternative of fb_1 which starts by the trigger p_{12}, as fb_5 and fb_6, which are triggered by that alternative, are not deployed to this device.

The same deployment alternative for device d_2 results in WCET value 0 for execution trigger p_{11}, because none of the function blocks that are executed as a consequence of this trigger are deployed to d_2. The entry for this device and trigger p_{12} takes into account execution of function blocks fb_5 and fb_6. The results for the example application using mapping alternative A are the following:

$$\text{WCET}(app,\ d_1) = \{\langle p_{11},\ 115\rangle,\ \langle p_{12},\ 5\rangle\}$$
$$\text{WCET}(app,\ d_2) = \{\langle p_{11},\ 0\rangle,\ \langle p_{12},\ 60\rangle\}$$

We can see how application analysis results change depending on the deployment configuration by applying the analysis to the same example, but using deployment alternative B. In this case, fb_4 is no longer deployed to d_1, so the WCET entry for p_{11} on this device now uses the first alternative in fb_2 and the only alternative in fb_3. The entry for p_{12} on device d_1 now also includes the execution of fb_5. Using this deployment alternative, the WCET result for device d_2 now has two entries, one for each trigger in fb_1. This is because now fb_4, which is triggered by p_{11}, is now mapped to this device. In case of using deployment alternative B the result of application WCET analysis would be the following:

$$\begin{aligned}\textsc{wcet}(app,\ d_1) &= \{\langle p_{11},\ 85\rangle,\ \langle p_{12},\ 35\rangle\}\\ \textsc{wcet}(app,\ d_2) &= \{\langle p_{11},\ 100\rangle,\ \langle p_{12},\ 30\rangle\}\end{aligned}$$

The Application analysis algorithm

The algorithm for device-specific WCET analysis of applications is separated into two functions. The first function, appFBNAnalysis, is used to recursively analyze the function block network implementing the application. The second function, appAnalysis, starts the network analysis for the application's internal triggers, and collects the results in a set representing the device-specific WCET of the application.

The function for analysis of application function block network is given in Algorithm 10. It uses the same principles as the function block network analysis for the composites, but does not collect information about generated outputs, and takes into account deployment of function blocks. The arguments for the function are event port e from which to start analysis, function block network *fbn* which is the context of the analysis, and device d for which the application analysis is performed. As a result, the function returns an integer value, corresponding to the WCET of the part of *fbn* starting with event port e.

The function starts with finding a connection leading from the selected event port on line 2, and returns 0 as a result on line 3 if no such connection exists. As applications have no interface, destination ports of connections can only be input event ports of function blocks. On line 7, the device-specific WCET data for the function block f, to which the destination port of the connection belongs to, is retrieved. The device argument of the wcet function is set to be the device that f is deployed to. Line 9 initializes the WCET value v, which represents the result of the application network analysis function. The analysis continues on line 10 by considering all WCET data entries defined for the function block's input event port. On line 11, the temporary WCET value for the currently analyzed data entry is initialized to 0. The for loop starting on line 12 iterates through all generated output events of the WCET entry. For each event, the WCET value of the sub-network starting with the event is analyzed to the recursive function call on line 13. The recursive result is multiplied by the number of event occurrences, and added to the temporary result v'. The combination of the *if* statement and the assignment on lines 16 and 17 add the WCET value of the data entry to the temporary result, but only if the function block f is deployed to the device d. Line 19 ensures that the function result v contains the highest

Algorithm 10 APPFBNANALYSIS(e, fbn, d)

```
 1: $\langle F_i, C\rangle \leftarrow fbn$
 2: if $\neg\exists\langle e_s, e_d\rangle \in C : e_s = e$ then
 3:     return 0
 4: else
 5:     Let $\langle e_s, e_d\rangle \in C$ be the connection for which $e_s = e$
 6:     Let $f$ be the FB to which $e_d$ belongs
 7:     $\langle W_e, W_i\rangle \leftarrow \textsc{wcet}(f,\ \textsc{deployedTo}(f))$
 8:     Let $\langle e_i, W_t\rangle$ be the element in $W_e$ for which $e_i = e_d$
 9:     $v \leftarrow 0$
10:     for each $\langle v_{fb}, o\rangle \in W_t$ do
11:         $v' \leftarrow 0$
12:         for each $e_o \in E_0 : o(e_o) > 0$ do
13:             $v_r \leftarrow \textsc{appFBNAnalysis}(e_0,\ fbn,\ d)$
14:             $v' \leftarrow v' + v_r * o(e_o)$
15:         end for
16:         if $\textsc{deployedTo}(f) = d$ then
17:             $v' \leftarrow v' + v_{fb}$
18:         end if
19:         $v \leftarrow \textsc{max}(v,\ v')$
20:     end for
21:     return $v$
22: end if
```

value computed so far. Line 21 returns the final WCET value.

The function performing device-specific application analysis, given in Algorithm 11, is similar to the internal trigger analysis defined for composite function blocks. As arguments, the function takes application function block network *fbn*, and device d, for which to perform the analysis. The function returns the previously defined application WCET data.

The application analysis function starts by initializing the result set w_{app} on line 1. The for loop on line 3 iterates through all function blocks contained by the network. For each function block, the WCET data specific for the device to which it is deployed to is retrieved on line 4. The analysis continues by considering all internal trigger WCET data entries of the retrieved data on line 5. On line 6, the WCET value v for the internal trigger is initialized to 0. The analysis then considers all

Algorithm 11 appAnalysis(*fbn*, *d*)

```
1:  $W_{app} \leftarrow \emptyset$
2:  $\langle F, C \rangle \leftarrow fbn$
3:  for each $f \in F$ do
4:      $\langle W_e, W_i \rangle \leftarrow$ wcet($f$, deployedTo($f$))
5:      for each $\langle t, W \rangle \in W_i$ do
6:          $v \leftarrow 0$
7:          for each $\langle v_{fb}, o \rangle \in W$ do
8:              $v' \leftarrow 0$
9:              for each $e_o \in E_o : o(e_o) > 0$ do
10:                 $v_r \leftarrow$ appFBNAnalysis($e_o$, $fbn$, $d$)
11:                 $v' \leftarrow v' + v_r * o(e_o)$
12:             end for
13:             if deployedTo($f$) $= d$ then
14:                 $v' \leftarrow v' + v_{fb}$
15:             end if
16:             $v \leftarrow$ max($v$, $v'$)
17:         end for
18:         $W_{app} \leftarrow W_{app} \cup \langle t, v \rangle$
19:     end for
20: end for
21: return $W_{app}$
```

data entries associated with that internal trigger, in order to find the one with the highest WCET value. The temporary value v' for each entry is initialized on line 8. The loop starting on line 9 iterates through all generated output events. The WCET value for each event is calculated by a call to the appFBNAnalysis function on line 10, multiplied by the number of event occurrences, and added to the temporary value v' on line 11. Lines 13 and 14 add the value of the function block WCET data to v', but only if the function block is deployed to the device for which the analysis is performed. On line 16 the WCET value for the internal trigger v is updated to the highest value calculated so far. The assignment on line 18 combines the internal trigger id with the final WCET value for the trigger, and adds it to the result set w_{app}.

5.4.3 Utilization analysis

A valuable property that can be calculated for processing resources, such as IEC 61499 devices, is their utilization. Knowing the utilization of processing resources early in the development process can help dimensioning the hardware platform to satisfy the needs of real-time execution even in the worst-case scenarios.

We calculate device utilization by combining the device-specific WCET of an application with the period of each execution trigger. The periods of execution triggers are defined on the system level, by assigning values to internal trigger IDs defined in function block WCET data. In case of sporadic triggers, the period denotes the minimal interarrival time between two triggering events.

Utilization is calculated for each device separately, as a sum of utilization of all execution triggers. For each execution trigger we calculate its utilization contribution by dividing the WCET value by the period of the trigger.

Definition 20. *The utilization U of device d in application app can thus be defined as:*

$$U(app,\ d) = \sum_{\langle p,w\rangle \in S} \frac{w}{period(p)},$$

$$where\ S = \textsc{wcet}(app,\ d).$$

The result of the utilization analysis is a positive real number, with

Table 5.2: Results of the utilization analysis.

Deployment alternative	Device	Trigger	WCET	Trigger Util.	Device Util.
A	d_1	p_{11} (300)	115	0.383	0.483
		p_{12} (50)	5	0.1	
	d_2	p_{11} (300)	0	0	1.2
		p_{12} (50)	60	1.2	
B	d_1	p_{11} (300)	85	0.283	0.983
		p_{12} (50)	35	0.7	
	d_2	p_{11} (300)	100	0.333	0.933
		p_{12} (50)	30	0.6	

the value of 1 representing full utilization of a device. A value over 1 corresponds to overutilization.

Example

To demonstrate the utilization analysis we will again use the application from Figure 5.9 a). To show how we can detect overutilization of a device, and verify that the problem is solved after redistributing function blocks between the two devices, we will use the two function block deployment alternatives given in Figure 5.9 b). As period values for p_{11} and p_{12} we will use 300 and 500, respectively. Table 5.2 shows the utilization analysis results. The first column gives the deployment alternative for which utilization is calculated. The second column states for which device the utilization is calculated. While the ID of the internal triggers and the period associated with it is given in the third column, the fourth column contains the WCET for each trigger. The fifth column describes how each trigger contributes to device utilization, and the sixth column gives the resulting device utilization.

From the results we can see that when using deployment alternative A the device d_2 is overutilized, having the utilization value 1.2. Changing the mapping of the function blocks to option B resolves the problem of overutilization, as in this case utilization is lower than 1 for of both devices.

It should be noted that the sum of WCET for all devices and execution triggers is not the same in deployment alternatives A and B. This is because the analysis considers the worst case for each device in isolation. For deployment alternative A, only one of the two alternative paths in fb_2 contributes to the worst case. For deployment alternative B, however, both of them result in worst cases, for device d_1 and d_2 respectively.

5.5 Implementation and evaluation

In addition to from formally defining the analysis algorithm, a part of the contribution of this thesis is a prototype analysis tool. In this section we first give an overview of the tool, and then describe how the tool was used to validate different parts of the analysis approach.

5.5.1 Analysis tool

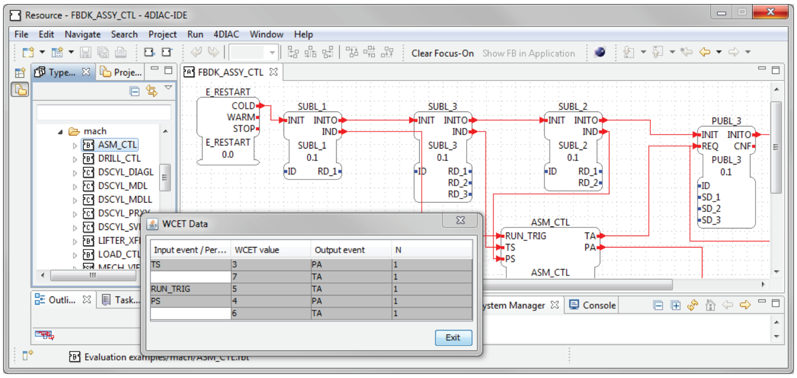
The prototype analysis tool implements all algorithms defined in this chapter, including compositional WCET analysis, two alternatives for normalizing WCET data, analysis of bounded cycles and propagation of cycle bounds, and analysis of processing resource utilization. The analysis tool is built as a plug-in for 4DIAC-IDE [56]. The implementation is completely integrated with 4DIAC-IDE model editors and allows analysis to be performed on standard 4DIAC models extended with WCET values for basic function block algorithms. The analysis results are also stored as a part of 4DIAC model elements, allowing reuse during compositional analysis. The plug-in also defines GUI elements for presentation and editing of WCET data stored together with function blocks. A screenshot of the 4DIAC tool containing a dialog window for presentation of WCET data can be seen in Figure 5.10.

The tool, packaged as an Eclipse plug-in, is freely available for download[1].

5.5.2 Evaluation of the WCET analysis

The analysis tool has been used to evaluate three separate aspects of the compositional analysis method: (i) demonstrate that the analysis can be applied to non-trivial systems, (ii) test the performance (running time) of the analysis method and how it scales with the size of the system, and

[1] http://www.idt.mdh.se/~jcn01/research/4DIAC-plugins/


Resource - FBDK_ASSY_CTL - 4DIAC-IDE
File Edit Navigate Search Project Run 4DIAC Window Help
Clear Focus-On Show FB in Application
FBDK_ASSY_CTL
mach
ASM_CTL
DRILL_CTL
DSCYL_DIAGL
DSCYL_MDL
DSCYL_MDLL
E_RESTART
SUBL_1
SUBL_3
SUBL_2
PUBL_3
ASM_CTL
Console
Exit

WCET Data

Input event / Per...	WCET value	Output event	N
TS	3	PA	1
	7	TA	1
RUN_TRIG	5	TA	1
PS	4	PA	1
	6	TA	1

Figure 5.10: A screenshot of the 4DIAC tool showing results from the analysis plug-in.

(iii) compare the performance and output of the two suggested WCET data normalization methods.

The tests were carried out on four different IEC 61499 models taken from the example systems provided by 4DIAC-IDE [56] and FBDK [24] tools. The four models were selected because they were most complex when taking into account the number of function blocks and their instances, and the number of levels of hierarchy. The analysis was executed on a computer containing a 4-core Intel I7 processor and 4GB of RAM.

Because true WCET values for algorithms and internal execution of service interface function blocks were not attainable, the tests were conducted using random WCET values for algorithms, uniformly distributed between 1 and 100. To account for the randomness of the data the test were repeated 1000 times for each system, each time with a new random set of WCET values. For each analysis invocation the existing WCET data stored in the models was cleared, forcing re-analysis of all function blocks.

Information about the number of function block types, instances, hierarchical levels and average analysis running time for the test systems is given in Table 5.3. To increase the precision of measuring short time intervals, the actual analysis running time values were measure for ten consecutive invocations and then divided to estimate running time of a single analysis invocation.

Table 5.3: Experiment setup and runtime results

System	FB types	FB instances	Hierarchy levels	Average max. el. running time (ms)	Average supremum running time (ms)
Boiler	60	158	6	11.81	11.86
DSCY_MDLL	20	41	4	4.64	4.47
ASSY_CTL	7	7	2	1.46	1.45
XFER_MDL	8	17	2	6.96	6.87

The time needed to perform the analysis using the maximal elements and the supremum normalization methods can be seen in the fifth and sixth column of Table 5.3, respectively. The results show that the choice of the normalization method did not significantly impact analysis running time. The time needed for performing the analysis was in the order of magnitude of 10ms, also for the fairly complex Boiler system.

During evaluation we have also compared results given by analysis using the two proposed normalization methods. The comparison was done for each individual execution origin (i.e. event source) within the systems, using the same sets of random algorithm WCET values. The results of this investigation are given in Table 5.4. They are aggregated by the systems and the execution origins, as shown in the first and the second column. The third column gives the number of different execution paths that can be taken from each origin. The fourth column contains the overestimation produced by the supremum method compared to the maximal elements method, on average over all random WCET input sets. The fifth column shows the maximum of all the supremum overestimation.

The results contain many execution origins with only one execution path, for which the supremum method naturally did not produce any overestimation. The average WCET increase when using the supremum method was between 3% and 75%. Although the maximal overestimation was only 12% for one origin, it went up to 258% in the worst case. The data shows that there is no direct correlation between the number of components, instances, hierarchy levels and execution paths, and the overestimation of the supremum method. This indicates that the effects of the normalization mostly depend on specific combinations of

Table 5.4: Experiment results per execution origin

System	Execution origin	Execution paths	Average WCET overestimation (%)	Maximal WCET overestimation (%)
Boiler	1 – 8	1	0	0
	9	36	10	21
DSCY_MDLL	1 – 7	1	0	0
	8	6	3	12
ASSY_CTL	1 – 4	1	0	0
	5	3	24	80
	6	2	26	80
XFER_MDL	1, 2, 3	1	0	0
	4, 5, 6	4	75	244
	7, 8, 9	4	75	258
	10	4	75	238

components rather than architectural complexity.

The two normalization methods have not shown any significant difference in terms of analysis running time. Lack of performance increase when using the supremum normalization can be explained by a combination of multiple factors. The target applications were not complex enough for the supremum method to have much effect on running time. This is also indicated by a large amount of execution paths having the same WCET value for both normalization methods. Also, in many cases the maximal elements method greatly reduced the number of examined execution paths, and thus also shortened the analysis running time. Lastly, the implementation of the supremum normalization results in a more complex algorithm than the maximal elements one, which can lead to a decline in performance for systems with a small number of hierarchical levels and execution paths. Still, we believe that the supremum method could be useful for dealing with combinatorial complexity in very large systems.

Table 5.5: Description of validation tests for cyclic execution path analysis.

Test	Tested functionality
1a	Detection of unbounded cycle.
1b	Application-level cycle analysis using a connection bound.
1c	Detection of unsupported cycle pattern.
2a	Analysis of nested cycles.
2b	Dependency of analysis results on bound position.
3a	Application-level cycle analysis using a component bound.
3b	Cycle analysis inside a composite component, with execution starting at an internal trigger.
3c	Cycle analysis inside a composite component, with execution starting at an input port of a composite.
4	Analysis of exit alternatives for component cycle bounds.
5	Detection of multiple cycle bounds for a single cycle.
6a	Propagation of a component cycle bound.
6b	Propagation of a connection cycle bound.
6c	Omitting propagation of consumed cycle bounds.
7a	Does not test a functionality which is not covered by previous test, but the results are used in combination with Test 7b and Test 7c.
7b	Correct handling of cycle exit alternatives in cases when cycle bound is propagated. Also used together with Test 7a and Test 7c.
7c	Validation of analysis using hierarchical composition of analysis results and bound propagation. Used together with Test 7a and Test 7b.

5.5.3 Validation of cyclic path analysis

The validity of the cycle analysis method and the functionality of the prototype implementation has been evaluated using 16 test scenarios. Each scenario was designed to cover a part of the desirable analysis behavior using a simple example. This approach allowed for parts of analysis method to be validated in isolation. The scenarios consisted of a test model and results which are expected as output of the analysis. Testing was done by re-creating the test models in the 4DIAC develop-

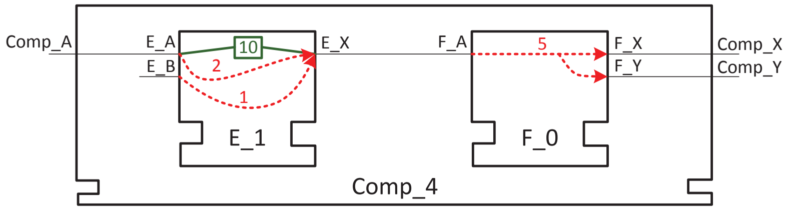


Figure 5.11: Model used for the example test scenario.

ment environment, and applying the prototype analysis tool to them. A detailed description of the test scenarios is publicly available as a technical report [38], while an overview is given in Table 5.5. The first column of the table provides the test ID, assigned based on the general model used in the test, and the second column gives a short description of the tested functionality. For all test scenarios, the analysis results obtained by the prototype tool matched the expected analysis results.

Test scenario example

To provide an overview of how testing was performed, this section describes one of the test scenarios in details. For this purpose, test *6a* from the technical report [38] was chosen, because it is the first one covering both WCET analysis and propagation of cycle bounds.

The purpose of the scenario is to test if a component cycle bound defined for a function block inside a composite is properly propagated to the interface of the composite. The model used for the test is given in Figure 5.11.

Because the cycle bound defined for function block *E_1* is traversed by the only path from input port *Comp_A* to input port *Comp_X*, and it is not consumed by a cycle inside the composite, it should be propagated to these two ports. In the same way, the same bound should be also propagated to ports *Comp_A* and *Comp_Y*. In addition, the timing analysis should return one execution alternative, starting from port *Comp_A*, having WCET value 7, and resulting in one output to both *Comp_X* and *Comp_Y*. The expected results can also be seen in Table 5.6.

It should be noted that although the propagation should create two new bounds, only one of them can be used on a higher level of hierarchy. From the point of view of composite's internal function block

Table 5.6: Expected results of WCET analysis and cycle bound propagation for the example test scenario.

WCET analysis results for *Comp_4*		
Input event	WCET	Generated outputs
Comp_A	7	*Comp_x* = 1; *Comp_Y* = 1;

Cycle bounds created for *Comp_4*		
Input port	Output port	Bound value
Comp_A	*Comp_X*	10
Comp_A	*Comp_Y*	10

network, this is because a single internal bound can not be applied to two cyclic paths. From the point of view of composite's interface, using both bounds would result with an unsupported cyclic pattern.

With the test scenario defined, the evaluation continued by creating the test model inside the 4DIAC development tool, and applying the analysis tool. The results obtained by the tool matched the expected results.

5.6 Summary

In this chapter we have presented a method for analysis of WCET and utilization of processing nodes for systems developed using the IEC 61499 standard. The analysis is performed in a compositional manner, using models of software, hardware and deployment. The compositional model-level approach results in efficient analysis methods, and allows the analysis to be performed in early stages of system development.

The WCET analysis is performed for each component in isolation, and results in context-independent WCET data that can be stored together with the component. This data can be reused when applying analysis to compositions which contain the component. The analysis method can also be applied to models containing cyclic execution paths through definition of cycle bounds, analysis of bounded cycles, and cycle bound propagation. By taking into account the model of the platform, and how components are deployed to the platform's processing nodes, the method can also be used to calculate a specific WCET of an application for each processing node, allowing calculation of node utilization.

The proposed analysis method is implemented as a prototype tool. The approach is validated through evaluation on a set of models from the example libraries of two IEC 61499 development tools, and a set of test scenarios.

Chapter 6

Related work

This section presents research related to the three topics covered by the work presented in this thesis: design and synthesis of hardware-specific code, communication in distributed embedded systems and analysis of extra-functional properties.

6.1 Design and synthesis of hardware-specific code

Although there exist many component models which specifically target the domain of embedded systems [19] only a few of them provide any support for modeling sensors and actuators, or the hardware platform in general.

Support for platform modeling is one of the core elements of the IEC 61499 standard [28], described in detail in Section 2.2. Although the standard allows modeling processing nodes and computer networks between them, it does not provide a method for modeling IO devices such as sensors and actuators. Instead, communication with IO devices is hard-coded inside software components.

A similar approach for modeling the hardware platform is provided by the ProCom component model [14,53], described in Section 2.1. Originally, ProCom did not provide any support for integrating sensors and actuators in the modeling or synthesis process. This support was added through contributions RC1 and RC2 of this thesis.

One of the most extensive supports for modeling hardware platform in component-based approaches is given by the AUTOSAR standard [2, 3, 23]. The standard is able to describe details of Electronic Control Units (ECUs), such as available ports and pins, how an ECU is connected to sensors and actuators, or how it is connected to a network. The software components can be mapped to ECUs, resulting in configuration descriptions for each ECU. In the software model, sensors and actuators are modeled using specialized components. Connections between these software components and actual devices are made during ECU configuration. Although AUTOSAR supports the notion of hierarchical component composition, the dependencies on sensors and actuators are not propagated from specialized components to the interface of composite components. Compared to our approach, where such dependencies inside composite components are exposed on the level of composite interface, the AUTOSAR composites hide the dependencies of their sub-components. Compared to ProCom which supports code synthesis, AUTOSAR relies on a run-time environment for execution of software components. Thus, the standard also leaves the communication with sensors and actuators to be handled by the run-time environment, rather than by a code synthesis approach which we propose.

Development of embedded systems using software and platform models has also been explored in the context of model-driven development. In general, automatic generation of code from models can be done using many languages, and examples of this can be seen in code generation for AADL [25,31], extended UML [12] and MARTE profile for UML [35,49]. Although the approaches using AADL and MARTE in some extent support modeling of sensors and actuators, all of these approaches either explicitly exclude or do not describe synthesis of code for such devices. Also, compared to the component-based development approach, model-driven approaches do not try to use reusable components as building blocks for systems, but only provide the ability to develop systems on a higher level of abstraction, hiding some of the details of the executable code.

Code synthesis for communication with IO devices is provided by methods for automatic generation of device drivers. For example, Zhang et al. [69], Chen et al. [16] and Ryzhyk [50] present methods where various domain-specific languages are used to describe hardware, behavior and interface between a device and operating system. These descriptions are used to generate C code for interaction with the device. These approaches, compared to ours, do not connect device models to models of applications, concentrating more on providing the correct code to interface with a device, rather then integrating a device into a system. The code generated using these methods could be compared with the IO device input code used as a part of our synthesis. Also, methods for automatic driver generation often target devices more complex than most sensors and actuators used in embedded systems, like network or graphical cards.

6.2 Communication in distributed embedded systems

Automatic generation of inter-node communication for the IEC 61499 standard has already been implemented as a part of some development tools, for example ISaGRAF [26] and nxtStudio [47]. Both of these tools generate the communication on the level of executable code. Generation of distributed systems from models is also investigated by several model-driven development approaches. Balasubramanian et al. [4] describe how models of software, platform and the mapping between the two can be used to generate parts of code that implement access to the communication media. Gokhale et al. [21] describe how distributed embedded systems can be generated using system models. The authors propose using platform-independent application models together with models of the platform to configure pre-existing middleware components. Hugues et al. [25] also provide means for generation of distributed applications from models, relying on predefined middleware to implement communications. Compared to all these approaches, which generate inter-node communication on the level of code, the approach described in research contribution RC3 of this thesis generates this communication on the model level, making it more visible to the system developers and analysis tools. Our approach also introduces separate generation phases, all of which have well-defined inputs and outputs, making the generation easily adaptable to new communication protocols, or transferable to other component-based frameworks.

A method for automatic generation of communication on an abstract level is described by Keznikl et al. [29]. The authors propose a method in which distributed applications can be modeled using components and connectors. In this approach connectors are first-class entities encapsulating communication and coordination between components, and can describe communication style and required extra-functional properties. On the application level, connectors are independent of component deployment. Base on the application model, deployment model and information contained by connectors, the method automatically generates deployment-specific connector instance configurations. Compared to the approach presented in Chapter 4, because the information needed to generate deployment-specific configurations is contained by the connector, generation does not need to extract a communication model or take into account the model of the platform. Also, the method concentrates more on providing configurations that satisfy requirements defined by connector, as opposed to our method which aims to define a generic generation framework which would synchronize platform-independent and platform-specific models.

Modeling of distributed software is also supported by the AUTOSAR standard [2,23]. Applications can be developed in a platform-independent manner, by connecting software components via a virtual function bus. Similarly to communication with sensors and actuators, communication between distributed nodes ia also provided during ECU configuration on the level of executable code, rather than on the model level.

Doukas and Thramboulidis [18] present a real-time framework that is able to run systems created using function block models, which also includes automatic generation of inter-device communication. The communication is implemented using entities called event-connection managers, providing an implementation which is more flexible than directly generating executable code code. Compared to our approach, the automatically generated communication is not visible in function block models, and the generation takes into account only the deployment model as opposed to the complete platform model. Also, the approach also only targets one protocol, and does not provide means for extensions like the approach presented in this thesis.

Brisolara et al. [9] provide generation of communication on the level of models as a part of a method which uses high-level UML models to generate executable and synthesizable Simulink models. Providing extensive support for automatic generation of communication was not the main aim of this work. Compared to work presented in this paper, the communication generation does not take into account the model of platform nodes and network connections between them, and therefore can not generate communication for different communication media and protocols. Also, in this approach the information about generated elements is not propagated back to the UML model.

6.3 Analysis of extra-functional properties

One of the main contributions of this thesis is a method for analysis of Worst-Case Execution Time (WCET). An introduction to WCET analysis and an overview of existing methods can be found in work by Wilhelm et al. [68]. Although analysis of WCET on the level of executable code is an already well established area, analysis of code can be performed only in late stages of development, when the complete implementation is available. As an alternative, in this thesis we describe a method for WCET analysis which is performed on the level of models, and can be applied in early stages of system development. The potential and some of the problems of such an approach have been previously described by Lisper [45].

The need for early analysis is also stressed in the work by Gustafsson et al. [22] where the authors propose a method for early WCET analysis on the code level by first creating a timing model of the code. Compared to the work presented in this thesis, this method does not allow performing the analysis on a system level (taking into consideration the software and the platform model at the same time), and does not produce safe estimates.

An approach for WCET analysis on an architectural level using AADL is presented by Gilles and Hugues [20]. The method first performs a model-to-code transformation, and then applies the analysis on the code level. The results of the analysis are in the end propagated back to the model. A similar method for Matlab/Simulink models has been presented by Kirner et al. [32]. Compared to these approaches, our contribution performs the analysis using only models, without a need for generation of executable code. The model-level analysis we propose can potentially be executed much faster than the analysis on the level of code, with the down side of producing greater overestimation.

Klobedanz et al. [33] present timing analysis for AUTOSAR. In their approach standard AUTOSAR models are first transformed to a model specialized for timing analysis, TIMMO, and described by event chains. These chains can then be used to determine CPU load and reaction time of parts of the AUTOSAR system. Compared to this, the analysis we propose does not require model transformations, and calculates timing properties based on component composition rather than event chains.

A method for worst-case reaction time analysis in IEC 61499 systems is described by Kuo et al. [34]. The authors first compile the system to gather timing information for its code. The analysis continues by applying a model-checker to the system code to incrementally predict reaction time, while visiting all possible states of the system. Besides the difference of analyzing execution time rather than reaction time, our analysis method does not imply generation of code and does not consider all possible execution paths in the system, as the compositional approach removes some paths that will never contribute to a worst-case result.

The timing analysis method presented as part of contribution RC4 is partly based on timing analysis for the ProCom component model presented by Carlson [13]. Although we applied some of the ideas of this approach, it deals with many ProCom-specific constructs and could not be directly applied to the IEC 61499 standard. Compared to the analysis proposed in this thesis, the analysis for ProCom does not allow capturing alternative execution alternatives, but always describes a single safe over-approximation of execution, similar to the supremum normalization described in our analysis. Also, it does not utilize platform and mapping information during the analysis.

Schedulability analysis [46] can be used in real-time applications to determine that all timing requirements of a system are satisfied, and can be compared to the utilization analysis presented in this thesis. One of the methods for performing schedulability analysis using system models is presented by Khalgui et al. [30]. The analysis method relies on transforming models to OS tasks. Each task is characterized by a WCET value and its predecessor and successor. Arranging the tasks in chains allows one to check if end to end deadlines are violated. Zoitl et al. [71] define the concept of event chains for IEC 61499 models as sets of function block execution, starting from an event source inside a function block and ending with a function block that will not cause any further execution. The authors aim to allow application of real-time scheduling theories by assigning real-time constraints to these event chains. Compared to these two approaches, the utilization analysis method presented in this thesis performs the analysis directly using models of component composition, rather than extracting execution tasks or event chains from such models. While these two approaches perform analysis only on the level of whole systems, our method applies analysis for each component in isolation and reuses results by performing the analysis in a compositional manner. Also, our approach computes only utilization, and not response times or absence of deadline violation.

Another part of the contribution RC4 is support for analysis of software models containing cyclic execution paths. How to apply model level analysis to such systems has rarely been explored by the research community None of the methods previously described in this section, with the exception of [32], addresses this problem.

Vulgarakis et al. present a method [65] for analysis of resource consumption of component-based systems by combining a component model with models of component behavior. Although this approach cannot be used to describe cyclic execution paths in multiple components, cyclic behavior contained by a single component can be modeled using history variables and specialized interface ports. Compared to this, we allow analysis of cyclic paths in compositions of components, and provide a method for propagating cycle bounds from a network inside a composite component to the interface of the composite.

In some cases a modeling language has the ability to explicitly describe iterative execution. An example of this is *for* loop element in Simulink. Support for analysis of such constructs for Simulink can be seen in work by Kirner et al. [32]. Similar support for loop analysis is included in work by Becker et al. [6] for the Palladio Component Model. As such cyclic execution is explicit and contained in one level of hierarchy, it relates to analysis of loops on code level. Contrary to this, our approach provides an analysis method that can be used on implicit cycles that occur as a result of component composition and can span over multiple levels of the model hierarchy.

Chapter 7

Conclusion

In this thesis we have presented multiple contributions which provide advancements in development of component-based embedded systems through use of software and hardware models. These contributions include modeling of sensors and actuators and automatic synthesis of code for communication with such devices, automatic generation of communication between distributed system nodes and analysis of extra-functional properties of systems. In this chapter we first summarize the contributions and discuss how they relate to the research questions, and then describe some of the possibilities for future research based on the work presented in this thesis.

7.1 Summary and discussion

Research Question 1: *How can we improve the support for integration of sensors and actuators in component-based development for embedded systems, so that dependencies to these devices are more easily manageable?*

This question was addressed by two separate contributions presented in Chapter 3. The first contribution allows modeling of sensors and actuators, how they are connected to processing nodes, and how they relate to software components. The modeling approach also provides means for software components to explicitly state their dependencies on sensors and actuators on the level of component interface, and permits definitions of these dependencies to propagate through the component hierarchy.

This contribution provides multiple benefits when developing embedded systems using a component-based approach: Firstly, it provides developers with a better overview of a system, as the same set of models that describes the software and execution platform now also provides information about sensors and actuators. Secondly, by removing sensor- and actuator specifics from software components, replacing them with explicitly stated dependencies, the approach both promotes reuse of components and reinforces the black-box view of a component. Lastly, the models allow easier re-configuration of systems, as, for example, changes in the hardware platform do not impose changes in the software model.

The second contribution addressing Research Question 1 utilizes the previously defined model to provide automatic synthesis of code for communication with sensors and actuators. The synthesis method relies on defining reusable code elements for model entities that are not system-specific, and automatic generation of code which combines such code elements in a way that provides executable code for a specific configuration of IO devices, processing nodes and software components.

The ability to automatically generate code for communication with IO devices provides a potential to alleviate development of component-based embedded systems by removing the time consuming and error prone task of implementing this functionality by hand. Also, this approach simplifies the synchronization between the system model and code, as the synthesized code always reflects the model it has been derived from.

In order to test the applicability the two contributions and evaluate them, we have created a prototype tool which implements both the modeling and the synthesis method. The applicability of the new modeling and code synthesis method has been tested by applying them to a realistic example, showing that the models can accurately describe the system and that the synthesized code implements the expected functionality. We have also evaluated the overhead of the synthesized code compared to a hand-written implementation.

Research Question 2: *How can we enhance development of distributed component-based systems in order to reduce the effort of synchronizing platform-independent and platform-specific models?*

This research question was answered by proposing a framework for automatic generation of communication between distributed platform nodes on the level of software models. The framework analyzes platform-independent software models in order to capture the distributed communication requirements of a system, detects how these requirements can be satisfied based on models of the platform, and then generates communication components in platform-specific models in order to implement the required communication. The framework was defined in a way which (i) allows it to easily be extended with implementation of communication using different media and protocols, (ii) provides means of implementing automatic optimization of generated communication, and (iii) allows it to be adapted to different component models. Synchronization between the platform-specific and platform-independent models is facilitated through annotations attached to both generated communication components and the original connections represented by these components.

As a part of the contribution, we have applied the proposed framework to the IEC 61499 standard and implemented a prototype tool. By exemplifying the framework on a simple example, and applying it to a case-study of two systems taken from an external library, we have evaluated the applicability of the approach.

Research Question 3: *How can we utilize software and platform models to efficiently analyze extra-functional properties of component-based systems in early stages of development?*

As a part of this thesis we have presented an approach that allows analysis of worst-case execution time and processing node utilization by using models of software, hardware and deployment defined by the IEC 61499 standard. The analysis is performed in a compositional manner. Each component is analyzed in isolation, by composing the data of its subcomponents, based on a model of component's implementation. The results of a component's analysis are stored together with the component, and reused when an instance of that component is used in a higher level of hierarchy. As a part of the contribution, we have also provided means to analyze models containing cyclic execution paths.

The resulting analysis method is applicable in early stages of system development, because it can be applied to system models even before the full implementation of a system is available. Also, the combination of a model-level and compositional analysis makes the analysis method efficient, and thus allows analysis to be performed early and often, as opposed to the traditional approach, where analysis is performed as a separate stage late in the system development.

The analysis method was also implemented as a prototype tool. The evaluation of the applicability and the performance of the analysis was performed using two approaches. The overall timing analysis was evaluated by applying the tool to multiple models taken from an external library, while a part of the analysis concerning cyclic execution path was evaluated using a set of internally defined test scenarios.

7.2 Future work

This section discusses some of the possibilities of future research based on the work described in this thesis.

7.2.1 Support for sensors and actuators

Although the approach for modeling sensors and actuators presented in this thesis allows attaching extra-functional properties to sensors and actuators, the presented work does not describe in detail how these properties should be propagated to the software model, or how to handle them in analysis. As a part of the future work, it should be investigated how to derive properties of IO software components from the properties of the IO devices, IOs, and platforms, based on the defined mappings and allocations.

The current method for defining synthesis input code, and generation of the adequate output code, results in some function calls and structure member dereferencing which add overhead to both the execution time and memory footprint of the synthesized code. It would be worth investigating how the presented method could be updated to produce more efficient code. This could, for example, be done by changing the method for defining input and output code for the synthesis. This could, however, reduce the readability of the input code and make definition of input code more complex, and result in more complex output code generation.

7.2.2 Automatic generation of distributed communication

The presented framework for automatic generation of distributed communication in component-based models allows extension of its automatic protocol selection phase. As has been demonstrated, in addition to the selection of viable communication protocols, this phase can be used to optimize the communication that will be generated. The optimization presented in this thesis was mainly intended for exemplifying the approach, and was therefore very simple. The current work would benefit by extending it to provide various optimization methods that could be used during the automatic protocol selection phase. Such work could also include defining extra-functional properties for communication media elements and connections between communication media and processing nodes. These properties could then be utilized by optimization methods.

The current version of the generation framework supports only direct connections between distributed nodes. In order to generate communication between components on two different nodes, a common communication media between the two nodes is required. Considering this limitation, the framework could be extended with the ability to generate relayed communication. This would enable communication between nodes that are not directly linked by a network, but share a common node to which both are connected. The principle could also be applied to chains of relaying nodes, instead of just one. Providing such a possibility would require changes to multiple parts of the framework: the communication model should be extended to allow describing relay communication, the media detection phase would have to be updated with the ability to detect possible relays, the protocol selection phase should be extended to properly treat the relay connections, and the component generation should include creation of relay components.

7.2.3 Analysis of extra-functional properties

The model-level worst-case execution time analysis presented in this thesis assumes that WCET values for algorithms implementing basic function blocks already exist, and the implemented prototype tool requires these values to be set by hand. As a part of future work, it would be worth exploring how the presented analysis method could be extended with an ability to acquire such values automatically. This could be done, for example, by static code analysis, or by performing measurements while executing the code.

The proposed analysis method can also be applied to analysis of properties other than worst-case execution time, for example average execution time or memory usage. This would however require more than simple inclusion of new properties as part of component description. Since different properties involve different methods of property composition, the part of the analysis which calculates results for networks of components would have to be adapted for each property. Still, the principles of compositional analysis and definition of context-independent properties could be reused.

The presented analysis method includes two methods for reducing the amount of context-independent WCET data stored with a component – the maximal elements and the supremum normalization methods. Although the tests conducted as part of the evaluation did not show any significant difference in performance of the analysis when using the two methods, in some cases of very complex systems the choice between the two methods could impact the analysis performance. It would be worth investigating on which level of complexity, or for which compositional patters, the performance of the two methods would differ. If the performance difference would justify the use of the less accurate method, it would also be interesting to explore how hybrids between the two methods could be used to provide a trade-off between the performance and the accuracy of the analysis. An example of such hybrid normalization method would be one which creates supremums only for groups of WCET alternatives that cross a specific threshold of similarity.

The analysis method could also be extended to take into account data flow in component compositions. In addition to being able to detect data dependencies between components, the method could utilize values of the exchanged data to provide more accurate analysis results. This could be achieved by parameterizing WCET values, or defining data conditions under which some execution alternatives are valid.

Bibliography

[1] Colin Atkinson, Christian Bunse, Christian Peper, and Hans-Gerhard Gross. Component-based software development for embedded systems – an introduction. In *Component-Based Software Development for Embedded Systems*, pages 1–7. Springer, 2005.

[2] AUTOSAR Development Partnership. Software component template version 4.2.1. Technical report, 2014.

[3] AUTOSAR Development Partnership. Specification for the ECU resource template version 4.2.1. Technical report, 2014.

[4] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, Feb 2006.

[5] Victor R. Basili. The experimental paradigm in software engineering. In H.Dieter Rombach, VictorR. Basili, and RichardW. Selby, editors, *Experimental Software Engineering Issues: Critical Assessment and Future Directions*, volume 706 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 1993.

[6] Steffen Becker, Heiko Koziolek, and Ralf Reussner. Model-Based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th International Workshop on Software and Performance*, WOSP '07, pages 54–65, New York, NY, USA, 2007. ACM.

[7] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.

[8] Etienne Borde, Jan Carlson, Juraj Feljan, Luka Lednicki, Thomas Leveque, Josip Maras, Ana Petricic, and Séverine Sentilles. PRIDE – an Environment for Component-based Development of Distributed Real-time Embedded Systems. In *9th Working IEEE/IFIP Conference on Software Architecture*. IEEE, June 2011.

[9] Lisane B. Brisolara, Marcio F. S. Oliveira, Ricardo Redin, Luis C. Lamb, Luigi Carro, and Flavio Wagner. Using UML as Front-end for Heterogeneous Software Code Generation Strategies. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 504–509, New York, NY, USA, 2008. ACM.

[10] Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, and Aneta Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

[11] Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A component model family for vehicular embedded systems. In *Proceedings of ICSEA*. IEEE, 2008.

[12] Sven Burmester, Holger Giese, and Wilhelm Schäfer. Model-driven architecture for hard real-time systems: From platform independent models to code. In *Model Driven Architecture–Foundations and Applications*, pages 25–40. Springer, 2005.

[13] Jan Carlson. Timing analysis of component-based embedded systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering*. ACM, June 2012.

[14] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment modelling and synthesis in a component model for distributed embedded systems. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 74–82. IEEE Computer Society, 2010.

[15] Jan Carlson and Luka Lednicki. Feasibility of migrating analysis and synthesis mechanisms from procom to iec 61499. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-268/2012-1-SE, June 2012.

[16] Hui Chen, G. Godet-Bar, F. Rousseau, and F. Petrot. Me3d: A model-driven methodology expediting embedded device driver development. In *Rapid System Prototyping (RSP), 2011 22nd IEEE International Symposium on*, pages 171–177, May 2011.

[17] Ivica Crnkovic and Magnus Larsson. *Building reliable component-based software systems.* Artech House Publishers, 2002.

[18] G. Doukas and K. Thramboulidis. A Real-Time-Linux-Based Framework for Model-Driven Engineering in Control and Automation. *Industrial Electronics, IEEE Transactions on*, 58(3):914–924, March 2011.

[19] Juraj Feljan, Luka Lednicki, Josip Maras, Ana Petričić, and Ivica Crnković. Classification and survey of component models. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-242/2009-1-SE, December 2009.

[20] Olivier Gilles and Jérôme Hugues. Applying WCET analysis at architectural level. In *8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[21] Aniruddha Gokhale, Krishnakumar Balasubramanian, Arvind S. Krishna, Jaiganesh Balasubramanian, George Edwards, Gan Deng, Emre Turkay, Jeffrey Parsons, and Douglas C. Schmidt. Model driven middleware: A new paradigm for developing distributed real-time and embedded systems. *Science of Computer Programming*, 73(1):39 – 58, 2008. Special Issue on Foundations and Applications of Model Driven Architecture (MDA).

[22] Jan Gustafsson, Peter Altenbernd, Andreas Ermedahl, and Björn Lisper. Approximate worst-case execution time analysis for early stage embedded systems development. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 308–319. Springer, 2009.

[23] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. Sangiovanni-Vincentelli, and M. Di Natale. Software components for reliable automotive systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 549–554, New York, NY, USA, 2008. ACM.

[24] Holobloc Inc. Function block development kit (FBDK), May 2012. http://www.holobloc.org/.

[25] Jerome Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):42:1–42:25, August 2008.

[26] ICS Triplex ISaGRAF. ISaGRAF, April 2014. http://www.isagraf.com/.

[27] IEC 61131-3: Programmable Controllers–Part 3: Programming Languages. International Electrotechnical Commission, Geneva, 1993.

[28] IEC 61499-1: Function Blocks-Part 1 Architecture. International Electrotechnical Commission, Geneva, 2005.

[29] Jaroslav Keznikl, Tomáš Bureš, František Plášil, and Petr Hnětynka. Automated resolution of connector architectures using constraint solving (ARCAS method). *Software & Systems Modeling*, 13(2):843–872, 2014.

[30] M Khalgui, X Rebeuf, and F Simonot-Lion. A tolerant temporal validation of components based applications. In *12th IFAC International Conference on Information Control Problems in Manufacturing (INCOM 06)*, 2006.

[31] BaekGyu Kim, L.T.X. Phan, O. Sokolsky, and Insup Lee. Platform-dependent code generation for embedded real-time software. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10, Sept 2013.

[32] R. Kirner, R. Lang, G. Freiberger, and P. Puschner. Fully automatic worst-case execution time analysis for matlab/simulink models. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 31–40, 2002.

[33] Kay Klobedanz, Christoph Kuznik, Andreas Thuy, and Wolfgang Mueller. Timing Modeling and Analysis for AUTOSAR-based Software Development: A Case Study. In *Proceedings of the Conference*

on Design, Automation and Test in Europe, DATE ’10, pages 642–645, 3001 Leuven, Belgium, Belgium, 2010. European Design and Automation Association.

[34] M.M.Y. Kuo, Li Hsien Yoong, S. Andalam, and P.S. Roop. Determining the worst-case reaction time of IEC 61499 function blocks. In *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, pages 1104 –1109, july 2010.

[35] Stéphane Lecomte, Samuel Guillouard, Christophe Moy, Pierre Leray, and Philippe Soulard. A co-design methodology based on model driven architecture for real time embedded systems. *Mathematical and Computer Modelling*, 53(3–4):471 – 484, 2011. Telecommunications Software Engineering: Emerging Methods, Models and Tools.

[36] Luka Lednicki and Jan Carlson. A framework for generation of inter-node communication in component-based distributed embedded systems. In *19th IEEE International Conference on Emerging Technologies and Factory Automation.* IEEE, September 2014.

[37] Luka Lednicki and Jan Carlson. Handling cyclic execution paths in timing analysis of component-based software. In *The 40th Euromicro Conference on Software Engineering and Advanced Applications.* IEEE, August 2014.

[38] Luka Lednicki and Jan Carlson. Specification of tests for validation of worst-case execution time analysis for cyclic execution paths in IEC 61499. Technical report, February 2014.

[39] Luka Lednicki, Jan Carlson, and Kristian Sandström. Device utilization analysis for IEC 61499 systems in early stages of development. In *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8, 2013.

[40] Luka Lednicki, Jan Carlson, and Kristian Sandström. Model Level Worst-case Execution Time Analysis for IEC 61499. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE ’13, pages 169–178, New York, NY, USA, 2013. ACM.

[41] Luka Lednicki, Ivica Crnković, and Mario Zagar. Automatic synthesis of hardware-specific code in component-based embedded systems. In *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pages 563–570, 2012.

[42] Luka Lednicki, Ivica Crnkovic, and Mario Zagar. Towards automatic synthesis of hardware-specific code in component-based embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, pages 71–74. IEEE, 2012.

[43] Luka Lednicki, Juraj Feljan, Jan Carlson, and Mario Žagar. Adding support for hardware devices to component models for embedded systems. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 149–154, 2011.

[44] Thomas Leveque, Etienne Borde, Amine Marref, and Jan Carlson. Hierarchical composition of parametric WCET in a component based approach. In *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, March 2011.

[45] Björn Lisper. Trends in timing analysis. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems, IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, volume 225, pages 85–94. Springer, 2006.

[46] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[47] nxtControl. nxtStudio, April 2014. http://www.nxtcontrol.com/.

[48] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

[49] A. Wendell O. Rodrigues, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from UML/MARTE to OpenCL. *Computing in Science & Engineering*, 15(1):46–55, 2013.

[50] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with termite. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, New York, NY, USA, 2009. ACM.

[51] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A resource model for embedded systems. In *In Proc. of the 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009)*. IEEE CS, June 2009.

[52] Séverine Sentilles, Petr Štěpán, Jan Carlson, and Ivica Crnković. Integration of Extra-Functional Properties in Component Models. In *12th International Symposium on Component Based Software Engineering*. Springer, 2009.

[53] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A component model for control-intensive distributed embedded systems. In *11th Int. Symposium on Component Based Software Engineering*, pages 310–317. Springer, 2008.

[54] Mary Shaw. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer*, 4(1):1–7, 2002.

[55] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[56] T. Strasser, M. Rooker, G. Ebenhofer, A. Zoitl, C. Sünder, A. Valentini, and A. Martel. Framework for Distributed Industrial Automation and Control (4DIAC). In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 283 –288, july 2008.

[57] Thomas Strasser, Alois Zoitl, James H Christensen, and C Sunder. Design and execution issues in IEC 61499 distributed automation and control systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 41(1):41–51, 2011.

[58] C. Sunder, A. Zoitl, J.H. Christensen, M. Colla, and T. Strasser. Execution Models for the IEC 61499 elements Composite Function Block and Subapplication. In *Industrial Informatics, 2007 5th IEEE International Conference on*, volume 2, pages 1169 –1175, june 2007.

[59] Jagadish Suryadevara, Aneta Vulgarakis, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Procom: Formal semantics. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-234/2009-1-SE, March 2009.

[60] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[61] Kleanthis Thramboulidis. IEC 61499 in factory automation. In *Advances in Computer, Information, and Systems Sciences, and Engineering*, pages 115–124. Springer, 2006.

[62] G. Čengić and K. Åkesson. On Formal Analysis of IEC 61499 Applications, Part A: Modeling. *Industrial Informatics, IEEE Transactions on*, 6(2):136 –144, may 2010.

[63] G. Čengić and K. Åkesson. On Formal Analysis of IEC 61499 Applications, Part B: Execution Semantics. *Industrial Informatics, IEEE Transactions on*, 6(2):145 –154, may 2010.

[64] K. Venkatesh Prasad, M. Broy, and I. Krueger. Scanning advances in aerospace & automobile software technology. *Proceedings of the IEEE*, 98(4):510–514, April 2010.

[65] A. Vulgarakis, S. Sentilles, J. Carlson, and C. Seceleanu. Integrating behavioral descriptions into a component model for embedded systems. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on*, pages 113–118, Sept 2010.

[66] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the ProCom realtime component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications*, 2009.

[67] V. Vyatkin. IEC 61499 as Enabler of Distributed and Intelligent Automation: State-of-the-Art Review. *Industrial Informatics, IEEE Transactions on*, 7(4):768 –781, nov. 2011.

[68] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – Overview of methods and survey

of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[69] Qing-Li Zhang, Ming-Yuan Zhu, and Shuo-Ying Chen. Automatic generation of device drivers. *SIGPLAN Not.*, 38(6):60–69, June 2003.

[70] A. Zoitl, T. Strasser, K. Hall, R. Staron, C. Sünder, and B. Favre-Bulle. The past, present, and future of IEC 61499. *Holonic and Multi-Agent Systems for Manufacturing*, pages 1–14, 2007.

[71] Alois Zoitl, Rene Smodic, C Sunder, and Gunnar Grabmair. Enhanced real-time execution of modular control software based on IEC 61499. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 327–332. IEEE, 2006.