Design, Verification and Deployment of Software Intensive Systems

A Multi-Paradigm Modelling Approach

Ontwerp, Verificatie en Ontplooiing van Software-Intensieve Systemen
Een Multi-Paradigma Modelleer Aanpak

Joachim Denil

Universiteit Antwerpen

Promotor:
prof. dr. Serge Demeyer
Co-Promotors:
prof. dr. Hans Vangheluwe
dr. Paul De Meulenaere

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica

Acknowledgments

As all dissertations, this work would not have been accomplished without the help of other people. First and foremost I would like to thank my supervisors. I could express my gratitude by saying how much I admire them academically, as supervisors, as mentors and as friends. But I am taking a different approach.

Back in 2008, I was working on an industrial technology transfer project about a new automotive communication network in the applied engineering department. While visiting an Austrian Technical University I got a phone call from *Paul De Meulenaere* to ask me if I would be interested in pursuing a PhD. After some initial fears, I agreed. Paul, thank you for giving me that push and believing in me. You also took care of all the financial aspects by getting me a grant from the university that allowed me to accomplish this. Thanks for bringing me into contact with a lot of industrial partners, the fruitful discussions and giving me the opportunity to aid in the creation of the automotive engineering courses at the applied engineering department.

At the start of my PhD, Paul introduced me to *Serge Demeyer*, my supervisor. Serge took me in as one of his own. Serge, you warned me about the permanent identity crisis, something I did not believe existed. After some initial successes, in the form of the publication and presentation of my first workshop paper, the first crisis arrived. I was lost. It is at these times that a supervisor is a beacon of light in the darkness. In the most difficult period of my PhD, you kept me motivated. Serge, you steered me through the whole process and encouraged me to broaden my horizon. You planted the initial seeds that a researcher needs and nurtured these. You also gave me the freedom to have fun while doing my job!

One year later, my last supervisor came into the picture. When *Hans Vangheluwe* became a professor at the UA, Serge urged me to have a good talk with Hans. Our first contact was very short. I explained very concisely what I was working on. Hans was already late for a meeting but commented: ”You should model this with DEVS, we will talk soon!” I had never heard about DEVS but with renewed courage, I started learning DEVS and created my simulation model. The next time I saw Hans my model was finished. The enthusiasm with which he welcomed my work was the turning point for me. The crisis years were over and I embraced the MPM-approach. Hans, thank you for introducing me to simulation, language engineering and model transformation. You gave a purpose to

my research and inspired me to make this work.

I am grateful to all the other people who have given me help and encouragement with everything related to my research. A special thanks goes to my jury for their interest in my work. Their comments greatly improved the quality of this dissertation. *Dirk Janssens*, *Pieter Mosterman* and *Martin Törngren*, I am proud you are in my jury! I hope we will meet again in the future to discuss interesting topics but also to enjoy the other good things of life. I would also like to thank the other researchers in the simulation and modelling community. They commented on this work and made my conference visits interesting in both an academic and non-academic way!

I would also like to thank the people from the RIOFI-fund for providing me with the means to accomplish this work. I also acknowledge the people from industry that listened to this work: LMS international, Dana, Melexis, Bombardier, Flanders' Drive, FMTC, DSP-Valley, Punch Powertrain, TriPhase, Luperco, Verhaert, NXP, and many others.

A special thanks goes to my past and current research friends at the Cosys-Lab in Antwerp, at the Ansymo group in Antwerp and the MSDL-group at McGill University in Montreal. They provided me with nice insights, a pleasurable research environment and some memorable after-work events. Thanks (in chronological order) Bart, Hugo, Marijn, Pieter, Tim, Joris, Sylvain, Margarete, Anne, Hans, Jan, Peter, Bart, Ahmed, Quinten, Tim, Peter, Peter, Levi, Sadaf, Eugene, Raphael, Maris, Frederic, Bart and Filip. A special thanks to Sadaf who proofread and commented on an early draft of this work. I will bring the chocolates!

My friends deserve a commendation for their support and for making the process bearable. Bondt, Fre and Katrien, To and Katrijn, Pietie and Ellen, Nico, BK and Sofie, Moeyer and Tinne, Knakke and Els, Danny and Heidi, Femke and Fille, and all others: "maybe a last one?"

Thanks to my parents, my sister Fran, my grandfather and my parents in law. They had to put up with me for years. Thanks for giving me a chance to find my interests and supporting me in all the choices I made during my life. I hope I do them proud.

Last but not least, I would be a professional garbage collector instead of a researcher without Natalie. Thanks for understanding this pursuit. Without your love and dedication over the last 12 years, I would not have gotten here. This work is dedicated to you.

Thank you all!

Abstract

Software has become a key component of a rapidly growing range of applications, products and services from all sectors of economic activity. This can be observed in large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services etc. Such systems are commonly called software-intensive systems. Software-intensive Systems (SiS) are characterized by their reactive nature, real-time requirements, mix of continuous and discrete (hybrid) behavior, the embedded character of some components of the system, the required dependability and the distribution of its elements.

Multi-Paradigm Modeling has been proposed by Vangheluwe and Mosterman as a method for the development and verification of software-intensive systems. In multi-paradigm modeling every aspect of a system is modeled explicitly, at the most appropriate level(s) of abstraction, using the most appropriate formalism(s). In addition, the development processes (workflows) are modeled explicitly, This thesis addesses some of the remaining research challenges in the design, verification and deployment of software-intensive systems.

Since multi-paradigm modeling promotes the explicit modeling of all aspects of a system, the process to design, verify and deploy systems has to be explicitly modeled as well. We propose the Formalism Transformation Graph and Process Model (FTG+PM) for this purpose. The FTG+PM contains, on the one hand, the specification of all formalisms and transformations involved in creating software-intensive systems, as well as their relationships. On the other hand, these formalisms and transformations are used within an explicitly modeld process, going from intial, domain-specific requirements to detailed design and deployment.

The FTG+PM is applied to the creation of an automotive power window system. Several languages and transformations are identified to create this exemplar. The process starts with the modeling of requirements. Domain-specific languages are used to model different aspects of the system (namely: the environment, plant and control model). These models are transformed to Petri nets for verification and to a hybrid simulation model to evaluate the dynamic behavior of the system. Finally, the system is deployed to a network of electronic control units.

During deployment it is necessary to evaluate the behavior and other extra-functional

properties of the solution. It is however not feasible to build the system and check the behavior at the implementation level. Current simulation models for system-level performance focus only on the computational parts of the system to evaluate the worst-case execution paths of the application. The system-level performance however, does not solely depend on the computational part of the system. The effects of the plant and environment should also be taken into account when evaluating the behavior of the system. For this, the Discrete Event System Specification (DEVS) formalism is used to describe and subsequently simulate deployed software-intensive systems. This includes the development of DEVS models of processors, memory, and buses (such as CAN and FlexRay). It is shown that DEVS is the most appropriate formalism to evaluate the effects of deployment of software-intensive systems by comparing the characteristics of SiS to the provided features of the DEVS formalism. Furthermore, a general simulation model for the automotive industry is constructed. Rule-based model transformations are used to create the simulation model from the different design models.

System-level performance models, though very useful, require calibration of the involved parameters to correctly reflect the actual system behavior. Since all aspects of the system are modeled explicitly, those models can be used to synthesize a calibration infrastructure. From the execution of the calibration infrastructure, a performance model can be built for calibration, i.e., to estimate the parameters of the simulation models of deployed software-intensive systems.

Finally, we evaluate the feasibility of transformations to optimise the deployment of software-intensive systems. A number of methods are used to leverage the explosion of the design-space. Firstly, multiple levels of abstraction/approximation are used where non-feasible solutions (or solutions that will never be optimal) are pruned. The created DEVS model together with other high-level performance models are used to evaluate the behavior at these different levels of approximation. Secondly, general search techniques are introduced in transformation models to search the design space. Finally, if an appropriate optimization formalism exists, for example mixed integer linear programs, the deployment model is transformed to this form. The FTG+PM is used for this purpose.

Nederlandstalige samenvatting

Software is uitgegroeid tot een belangrijk onderdeel van zeer veel toepassingen, producten en diensten uit alle economische sectoren. Dit kan worden waargenomen in grootschalige heterogene systemen, ingebedde systemen voor toepassingen in de automobielsector, telecommunicatie, draadloze ad hoc systemen, zakelijke toepassingen met een nadruk op web services, enz. Dergelijke systemen noemen we software-intensieve systemen; ze worden gekenmerkt door hun reactieve aard, real-time vereisten, mix van continu en discreet gedrag, een vereiste van betrouwbaarheid en de gedistribueerde aard van de componenten van het systeem.

Multi-Paradigma Modelleren (MPM) is reeds voorgesteld als een werkwijze voor de ontwikkeling en verificatie van software-intensieve systemen. Tijdens multi-paradigma modelleren wordt elk aspect van een systeem expliciet gemodelleerd, op het (de) meest geschikte niveau(s) van abstractie, met behulp van de meest geschikte formalisme(n). Dit proefschrift tracht een aantal van de resterende lacunes voor het ontwerp, de verificatie en de ontplooiing van software-intensieve systemen te overbruggen.

Omdat multi-paradigma modelleren het expliciet modelleren van alle aspecten van een systeem bevordert, moet ook het proces om software-intensieve systemen te creëren, worden gemodelleerd. Wij stellen de Formalism Transformation Graph and Process Model(FTG + PM) voor als een geschikt formalisme om MPM processen te modelleren. De FTG + PM bevat enerzijds, de definities van alle formalismen en transformaties die betrokken zijn bij het creëren van software intensieve systemen. Anderzijds worden deze formalismen en transformaties gebruikt in een proces om een systeem te creëren.

De FTG + PM wordt toegepast op de ontwikkeling van een autotechnologisch voorbeeld, een automatische ruitbedieningssysteem. Het proces begint met het modelleren van vereisten van het systeem. Domein-specifieke talen worden gebruikt om verschillende aspecten van het systeem te modelleren (namelijk: het omgevings-, plant en controle model). Deze modellen worden getransformeerd tot Petri netten voor verificatie enerzijds en tot een simulatiemodel om het dynamisch gedrag van het systeem te evalueren anderzijds. Ten slotte wordt het systeem ontplooid op een netwerk van elektronische regeleenheden.

Tijdens de ontplooiing van het systeem moeten het gedrag en andere extra-functionele eigenschappen van de oplossing worden geëvalueerd. Het is echter niet haalbaar om

het systeem te bouwen en daarna het gedrag te controleren tijdens de uitvoering ervan. Het Discrete Event System Specification formalisme wordt geëvalueerd als een geschikt formalisme voor de simulatie van gedistribueerde software-intensieve systemen.

Systeemniveau-modellen vereisen echter kalibratie van de betrokken parameters om de werkelijkheid te weerspiegelen. Aangezien alle aspecten van het systeem zijn gemodelleerd, kunnen deze modellen gebruikt worden om een infrastructuur te genereren die voor de ijking instaat. Door de uitvoering van de kalibratie-infrastructuur kan een performantie model worden opgebouwd dat zorgt voor de kalibratie namelijk, de parameters van de simulatie modellen inschatten.

Ten slotte hebben we de haalbaarheid onderzocht van transformaties voor het optimaliseren van de ontplooiing van software-intensieve systemen. Een aantal werkwijzen worden gebruikt om de explosie van de ontwerp-ruimte onder controle te houden. Ten eerst zijn verschillende abstractie-/approximatieniveaus geïntroduceerd waar niet-haalbare oplossingen (of oplossingen die nooit optimaal zullen zijn) worden verwijderd. Ten tweede zijn algemene zoektechnieken in de transformatie modellen opgenomen om de ontwerpruimte te doorzoeken. Tot slot, als een bestaand optimalisatie model ter beschikking is, wordt het ontplooiingsmodel getransformeerd naar deze voorstelling.

Publications

The following peer-reviewed papers and technical reports where I was a co-author served as a basis for this dissertation:

— Joachim Denil, Hans Vangheluwe, Pieter Ramaekers, Paul De Meulenaere and Serge Demeyer. DEVS for AUTOSAR platform modelling *Proceedings of the SpringSim Multiconference (DEVS/TMS)*, 2011. (Chapter 6): Hans proposed the idea, Joachim refined the idea, implemented the simulation model and wrote the paper. Hans, Serge and Paul commented on the work and reviewed the draft of the paper.

— Joachim Denil, Antonio Cicchetti, Matthias Biehl, Paul De Meulenaere, Romina Eramo, Hans Vangheluwe and Serge Demeyer. Automatic Deployment Space Exploration Using Refinement Transformations *Electronic Communications of the EASST: Recent Advances in Multi-paradigm Modelling, vol.50*, 2012. (Chapter 8): Joachim proposed the idea. Joachim, Antonio and Matthias refined this idea. Joachim implemented the case-study, Antonio and Joachim worked on the refinement transformations, Matthias and Joachim wrote the model2text transformation, Joachim wrote the majority of the paper.

— Joachim Denil, Hans Vangheluwe, Paul De Meulenaere and Serge Demeyer. Calibration of deployment simulation models: a multi-paradigm modelling approach *Proceedings of the SpringSim Multiconference (DEVS/TMS)*, 2012. (Chapter 7): Joachim proposed the idea, did the implementation and wrote the paper. Hans, Serge and Paul reviewed the paper.

— Sadaf Mustafiz, Joachim Denil, Levi Lucio, and Hans Vangheluwe. The FTG+PM Framework for Multi-Paradigm Modelling: An Automotive Case Study *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, 2012. (Chapter 4): Joachim, Levi and Sadaf created the models, Joachim and Sadaf created the FTG+PM, Sadaf wrote the majority of the paper, Joachim wrote the rest of the paper. Hans and Levi reviewed the paper.

— Levi Lucio, Sadaf Mustafiz, Joachim Denil, Bart Meyers, Hans Vangheluwe. The Formalism Transformation Graph as a Guide to Model Driven Engineering *School of Computer Science, McGill University, Technical Report, SOCS-TR2012.1*, 2012 (Chapter 3): Hans, Joachim and Levi proposed the idea. Hans, Joachim, Sadaf, Levi and Bart refined the ideas. Joachim created the transformations to the AToMPM transformation language. Sadaf created the back-end operations in AToMPM. Joachim, Sadaf and Levi wrote the report.

— Levi Lucio, Joachim Denil, Hans Vangheluwe. An Overview of Model Transformations for a Simple Automotive Power Window *School of Computer Science, McGill University, Technical report, SOCS-TR-2012.2*, 2012. (Chapter 4): Levi and Joachim created the models and wrote the report. Hans reviewed the report.

— Joachim Denil, Han Gang, Magnus Persson, Xue Liu, Haibo Zeng, Hans Vangheluwe, Transformation-based approaches to Design-Space Exploration, *Technical Report*, 2012. (Chapter 8): Joachim proposed the ideas, implemented the prototypes (except for the MILP model by Han), and wrote the report. Magnus, Haibo, Xue and Hans helped in refining the ideas and reviewed the report.

The following peer-reviewed papers and technical reports where I was a co-author are not incorporated within this dissertation:

— Tim Hermans, Joachim Denil, Paul De Meulenaere, Jan Anthonis. Decoding of data on a CAN powertrain network *proceedings of the 16th annual symposium on communication and vehicular technology in the benelux*, 2009

— Joachim Denil, Serge Demeyer, Paul De Meulenaere, Kurt Vanstechelman and Kris Maudens. Wrapping a Real-time Operating System with an OSEK Compliant Interface - a Feasibility Study *Proceedings of the Seventh Workshop on Intelligent solutions in Embedded Systems*, 2009

— Tim Hermans, Pieter Ramaekers, Joachim Denil, Jan Anthonis, Paul De Meulenaere. Integration of AUTOSAR in an Embedded Systems Development Process: A Case Study, *Proceedings of the 37th EUROMICRO conference on software engineering and advanced application*, 2011

— Joachim Denil, Serge Demeyer, Paul Demeulenaere, Kurt Maudens, Kris Vanstechelmans. Migrating from a proprietary RTOS to the OSEK standard using a wrapper *in M. Conti, S. Orcioni, N. Martnez Madrid, R. Seepold; Lecture Notes in Electrical Engineering, Solutions on embedded systems, Springer*, 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"The time has come," the Walrus said, "To talk of many things: ..."
— Lewis Caroll
Through the Looking-Glass and What Alice Found There

1.1 Context

Software has become a key component of a rapidly growing range of applications, products and services from all sectors of economic activity. This can be observed in large-scale heterogeneous systems, embedded systems for automotive applications, telecommunications, wireless ad hoc systems, business applications with an emphasis on web services etc. For example, an automobile in the 1970's was an almost complete mechanical device where only the radio had some electronic components. In contrast, today's vehicles contain up to 70 electronic control units (ECU) for controlling a range of features from safety functions such as anti-lock braking systems (ABS) and electronic stability program (ESP) to comfort functions for air-conditioning [Broy, 2006]. Such systems are now commonly called software-intensive systems (SIS) and, more recently Cyber-Physical Systems. In such systems, software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole.

Giese characterises a software-intensive system as follows: *Software-intensive systems are characterized by their reactive nature, real-time requirements, mix of continuous and discrete behaviour (hybrid), the embedded character of some components of the system, the required dependability and the distribution of its elements.* [Giese, 2006]

The heterogeneity of the different domains makes it hard to develop software-intensive systems. For example: One of the most demanding challenges in automotive electronics stems from the plethora of physical and logical domains to be managed. In general, SIS contain electronics with embedded software. Moreover, mechanics, hydraulics, pneumatics, magnetics and many other domains may contribute to the system's behaviour. For all these domains, different description formats are used and a plethora of single-domain software tools such as simulators, are usually available [Pelz *et al.*, 2005].

1.2 Motivation

As implied in the previous section, software-intensive systems feature an integration of the continuous world and the computational components. When one attempts to analyse the behaviour of these systems some friction arise. When developing software-intensive systems for control applications for example, the laws of physics are, at a commonly used level of abstraction, naturally expressed using ODEs. This results in continuous behaviour on the one hand. On the other hand, the control system uses an actuating device to control the physical process (often referred to as the "plant"). Usually some of the effects of the actuator on the plant are measured and compared with the desired output of the overall system. The control part is commonly implemented using a digital computational device. These devices are commonly expressed in discrete-time.

The problem is further exacerbated by deployment effects. For performance, cost and practical reasons, the increasingly complex systems are often implemented in a distributed fashion. This means that the computational hardware components are scattered throughout the system and need to interact using a communication medium (most commonly, a bus). These effects, combined with the interaction with the physical part of the system, make it difficult to analyse the overall system.

When developing a process for the design, verification and deployment of software-intensive systems we need to take into account these different observations. Below are some of the requirements for an SIS development process:

1. Design of the components using abstractions and notations close to the domain of the engineer: Because of the rising complexity, it is key that domain knowledge can be represented accurately. It has been argued that using domain-specific languages instead of a general purpose language increases productivity, reduces errors, etc. [Mannadiar and Vangheluwe, 2011]. This also means that a need arises to model at different abstraction or approximation levels so different domain experts involved in different stages of the development process, possibly working on different components of the overall system, can work closer to their field. It is crucial to raise the levels of abstraction at which a design is done [Bouyssounouse and Sifakis, 2005].

2. Verification of the behaviour of the design at different stages in the process: The interactions of the physical part and the discrete part of the system have to be verified at different abstraction or approximation levels. According to a recent study by the Aberdeen Group, one of the key enablers for the design of complex systems is the use of system simulation tools [Boucher and Kelly-Rand, 2011]. Industrial

leaders identified the increased prediction of system behaviour (prior to testing) as a top strategy for system design. This means that the different modelling formalisms involved need to be meaningfully combined to produce a trace that can be analysed for verification purposes.

3. Optimization of the design: During the design of systems not only the feasibility of the design is important. Designers want optimality with respect to a set of goal functions such as cost, how well real-time deadlines are met, etc. Especially during the deployment of a system, a plethora of design choices needs to be made that have a large impact on the behaviour of the system and on goal functions such as cost, extensibility, etc. Automatic methods for exploration (more specifically known as design space exploration) are needed to make the optimization process less dependable on scarce know-how and error-prone hand tuning [Wirsing *et al.*, 2006].

4. Process and enactment of the process: Because of the complexity of SIS it is important to have appropriate processes to describe, prescribe and manage the design process [Boucher and Kelly-Rand, 2011]. Tool support is needed to help designers in all stages of the process, guiding the developers and allowing them to test different assumptions. This includes but is not limited to: opening modelling environments to model certain parts of the system and starting a simulation or analysis to verify that the behaviour is desired.

5. Traceability between the different requirements and model elements within the whole design process: Traceability is an important method used for building high-confidence software-intensive systems found in automotive, aerospace and medical domains. It is often a mandatory practice for these systems to obtain certification. It is also needed to trace back errors in the design to design choices at earlier stages in the design process. Traceability can be seen in two directions: (a) conformance: between the different modelling elements of the produced models and the requirement models at the different abstraction levels and, (b) temporal: between the different steps and modelling elements in the design process. Both need to be considered when designing a process for software intensive systems.

6. Documentation: Every step in the design process should be documented so no design information is lost. This information is used for new releases of the product and to evaluate the processes involved in building a software-intensive system. It is also an important enabler for collaboration, possibly concurrent development.

1.3 Exemplar: The Power Window

To show the contributions of this thesis we use a power window case study. This case study has been used by the multi-paradigm modelling community on a number of occasions.

— Mosterman and Vangheluwe proposed computer automated multi-paradigm modelling (CAMPaM) using the power window exemplar [Mosterman and Vangheluwe, 2004].

— Prabhu and Mosterman used a commercially available toolset for the modelling, simulation and code generation of the power window exemplar [Prabhu and Mosterman, 2004].

— Boulanger et al. use the power window exemplar to address semantic adaptations involved in multi-formalism modelling and simulation [Boulanger *et al.*, 2011].

Power windows are automobile windows that can be raised and lowered by pressing a button or switch, as opposed to using a hand-turned crank handle. Such devices exist in the majority of automobiles produced today. The basic controls of a power window include raising and lowering the window. An increasing set of functionalities is being added to increase the comfort, safety and security of vehicle passengers. To manage this complexity while reducing costs, automotive manufacturers use software to handle the operation and overall control of such devices. However, as a power window is a physical device that may come into direct contact with humans, it becomes imperative that sound construction and verification methods are used to build such software.

Safety requirements of the power window system are detailed by government bodies such as the Road Safety and Motor Vehicle Regulation Directorate - Transport Canada. They address the safety issues of the power window, for example, the maximum force that may be exerted on an object by a window going up. Below is an excerpt from the requirements issued by the Canadian Government [Canada Transport, 2009]:

> S5.1 While closing, the power-operated window, partition, or roof panel shall stop and reverse direction either before contacting a test rod with the properties described in S8.2 or S8.3, or before exerting a squeezing force of 100 Newtons (N) or more on a semi-rigid cylindrical test rod with the properties described in S8.1, when such test rod is placed through the window, partition, or roof panel opening at any location in the manner described in the applicable test under S7.
>
> S5.2 Upon reversal, the power-operated window, partition, or roof panel system must open to one of the following positions, at the manufacturer's option: (a) A position that is at least as open as the position at the time closing was initiated; (b) A position that is not less than 125 millimetres (mm) more open than the position at the time the window, partition, or roof panel reversed direction; or (c) A position that permits a semi-rigid cylindrical rod that is 200 mm in diameter to be placed through the opening at the same location as the rod described in S7.1 or S7.2(b).

Other requirements of the system are not only safety requirements and are thus not addressed by the governing bodies. These are requirements originating in the features that a company wants to present to its customers. Prahbu and Mosterman define some textual requirements of the power window system in [Prabhu and Mosterman, 2004]. We use an adapted version of these requirements with the safety requirements of [Canada Transport, 2009]:

1. The window has to start moving within 200 ms after a move command is issued;
2. The window has to be fully opened and closed within 4 s;
3. The force to detect when an object is present should be less than 100 N;
4. When an object is present, the window should be lowered by approximately 12.5 cm;
5. The window can only be operated when in the 'start', 'on' or 'accessory' position;
6. The driver commands have precedence over the passenger commands;

The power window exemplar has all the essential complexity typical of a software-intensive system. A physical window has to be moved within certain real-time bounds, while the information of the detection of an object is fed back to the control component for the safety requirement. It also has a distributed nature since the controller sensing the interactions of the driver is physically on the other side of the car. Hence, information has to be transmitted using a communication medium that connects the different control units.

1.4 The Case for Multi-Paradigm Modelling

Multi-Paradigm Modelling, as introduced by Mosterman and Vangheluwe in [Mosterman and Vangheluwe, 2004], is a perspective on system and software development that advocates that every aspect of a system should be modelled explicitly. These models should be built at the appropriate level(s) of abstraction using the most appropriate modelling formalisms. Model transformations can be used to pass information from one representation to another during development. It is thus desirable to consider modelling as an activity that spans different paradigms. The main advantage of such an approach is that the modeller can benefit from the multitude of existing languages, methods and associated tools for describing and automating software and system development activities. Furthermore, the meaningful combination of formalisms can be dealt with using appropriate model transformations.

To make this idea more concrete, one may think of a UML statechart model representing the abstract behaviour of a software system being converted into a Java model for execution on a given platform; or of the same statechart being transformed into a model in a formalism that is amenable to verification. Another advantage of this perspective on development is the fact that toolsets for implementing a particular system development method become flexible. This is thanks to the fact that formalisms and transformations may be plugged in and out of a development toolset thanks to their explicit representation.

The idea of Multi-Paradigm Modelling is close to Model Driven Engineering (MDE): in MPM the emphasis is more generally on the fact that several modelling paradigms are employed in modelling. This includes the study and use of relations between the elements of the various paradigms; MDE is rather focused on proposing a method where a set of model transformations are chained in order to pass from a set of requirements for a system to running software on a given platform.

In summary MPM encompasses: *Model every aspect of a system explicitly at the most appropriate level(s) of abstraction, using the most appropriate formalism(s).* The enabler to achieve this is *Modelling Language Engineering*. This includes, model transformations as well as processes which need to be explicitly modelled.

Referring back to the needs of design, verification and deployment of software-intensive systems presented in section 1.2, the following advantages of using a multi-paradigm modelling approach are apparent:

1. Design of the components close to the domain of the engineer: Multi-Paradigm Modelling promotes the use of the appropriate formalisms at the right level(s) of abstraction. Language engineering lets us build such appropriate languages at different levels of abstraction thereby reducing the cognitive gap between the engineers' mental model of a domain and the abstractions.

2. Verification of the behaviour of the design at different stages in the process: Model transformation allows us to transform models in domain specific languages to other well known formalisms where certain properties can be checked. Examples are the interactions between the discrete and continuous components and properties related to safety requirements, real-time behaviour, extensibility, etc.

3. Optimization of the design: As already stated, model transformations allow one to transform models to different formalisms where certain properties can be checked and performance measures computed. What-if analysis can be used to evaluate trade-offs in the design space manually. Transformations can also be used to explore the design space automatically based on the feasibility criteria and the set of goal functions.

4. Enactment of the process: A direct consequence of the MPM definition is that not only the design artefacts should be modelled but that also the process should be modelled. Megamodelling [Bézivin *et al.*, 2003; Favre, 2004; Hebig *et al.*, 2011] and process modelling [OMG, 2008b; Bendraou *et al.*, 2007; Diaw *et al.*, 2011] are well established research fields in the modelling community. The results of their research can be used to model and enact a process for the MPM design of software-intensive systems.

5. Traceability between the different requirements and model elements within the entire design process: Since everything has to be explicitly modelled within an MPM design process, all design and deployment information is present in the models and transformation models. The transformation models can create the needed traceability links between the different modelling elements. Traces of the process

can show the temporal relations between the different models.

6. Documentation: Models and model transformations make design information explicit and documented. MPM takes a separation of concerns approach to the modelling of complex systems by encouraging that models of complex multi-faceted systems be separated into a number of single-facet models. The design information for the different facets is separated.

Our premise is therefore that multi-paradigm modelling is a valuable approach for the development of software-intensive systems.

1.5 Challenges and Contributions

From an abstract perspective, multi-paradigm modelling has indeed all needed aspects for solid software-intensive systems development. This was already shown in [Mosterman and Vangheluwe, 2004] where the design and verification of the power window exemplar is developed using an MPM approach. A number of open problems do however remain.

As already mentioned, megamodelling and process modelling are well established research fields in the modelling community. However, in the context of multi-paradigm modelling, there has been very little research on the representation and execution of process models. The questions guiding the representation of a process model are: (a) *What constructs should inherently be represented in a process model for MPM?* and (b) *What is an appropriate formalism to represent these constructs in?*

During deployment it is necessary to evaluate the behaviour and other extra-functional properties of the solution. Since we are deploying systems to a distributed network of ECUs it is not feasible to build the system and check the behaviour at the implementation level. It is therefore imperative that an appropriate formalism is used to evaluate this behaviour. The question guiding this part is: *What is an appropriate formalism to evaluate the behaviour of a deployed system?*

A number of methods exist to evaluate the behaviour of a (partially-) deployed system besides the result of the previous question. All these techniques however, require a calibration of the parameters involved in the models to reflect actual system behaviour. In software-intensive systems it is impossible to decouple the behaviour of the computational and physical parts of the system for calibration since there are feedback loops between them. the question remains: *How can MPM leverage the calibration of simulation models of deployed software-intensive systems?*

To optimise a design or deployment, a multitude of methods exist in the literature, such as meta-heuristics, mathematical optimisation methods, etc. Design space exploration using a model transformation based approach is a relative new research branch in the modelling community. We want to answer the following questions: *How can multi-paradigm modelling be used for the optimisation of the deployment of software-intensive systems?* Because this question is too general, we focus on the feasibility of model transformation for design space exploration.

Finally, an example multi-paradigm modelling approach for the design and verification of software intensive systems is needed. This can be used as a guide for other software-intensive systems. We show this by building a process model for the exemplar, the power window system.

This thesis tries to bridge the presented gaps with the following contributions:

— Creation of methods, techniques and a tool chain supporting the MPM development. The methods, tools and techniques are used for the hybrid simulation of software-intensive systems, the simulation of deployed software-intensive systems, the calibration of system-level performance models, the automatic design space exploration of the deployment and for process enactment.

— A complete specification and design of an accepted exemplar, the power window system, available as a replication package. This includes: requirements engineering, verification, hybrid simulation at different abstraction levels, calibration of simulation models and automatic design space exploration. Different formalisms and transformations are defined that can be reused, extended and analysed for other research purposes.

— Demonstration of the feasibility of MPM and the accompanying tool chain on the accepted exemplar of the power window system. The complete specification of the accepted exemplar shows that the MPM-approach for the design, verification and deployment for software-intensive systems is feasible.

The focus of this research is the feasibility of MPM and the creation of new methods, techniques and tools for the design, verification and deployment of software-intensive systems. The feasibility for all of the developed techniques, methods and tools is validated by creating a proof-of-concept implementation using the power window exemplar. The materials in this work are published on the ansymo website: `http://ansymo.ua.ac.be/artefacts`.

1.6 Delimitations and Alternatives

— In this dissertation we focussed on the design, verification and deployment of the software part of software-intensive systems. The extension to a complete mechatronic case, including the electro-mechanical components, of the methods, techniques and tools developed in this thesis is future work.

— While the techniques, methods and tools presented in this thesis are aimed to be used for a broad range of software-intensive systems, we only applied the developed techniques, methods and tools on a single automotive exemplar. We believe this exemplar is representative for a broad class of problems. Still, the evidence provided is anecdotal. The validation on a broad range of software-intensive systems is future work.

— We did not take Architecture Description Languages (ADL) into account within this work. ADLs are common in the development of software-intensive systems, shown by the number of ADLs in existence [Cuenot *et al.*, 2007; Feiler *et al.*, 2006]. ADLs often combine different aspects of the design of a system, such as requirements, tracing, analysis information and design choices. In this work, the concerns addressed by ADLs are separated. This separation is done for reasons of clarity. The introduction of ADLs into the design, verification and deployment processes as we describe them in the FTG+PM is part of future work.

— Most of the tools considered in this work have been developed by different members of the MSDL group. This allows us researchers, to have full control over all the aspects of the formalisms and languages. It also allows for an a-priori integration of all the different formalisms presented in this work. In an industrial context this is hard however. The intent of our work is to explore fundamentally new approaches and provide demonstrators to industrial tool builders. In [Biehl, 2013], Biehl describes the challenges involved in a-posteriori integration of different tools and shows the creation of custom tool-chains based on a domain-specific language, the Tool Integration Language (TIL).

— Software product lines introduce variability at all levels of abstraction/approximation. This is not taken into account in this work and is considered future work.

1.7 Roadmap of the thesis

Figure 1.1 shows an overview of the thesis. The main contributions are shown using the exemplar, the power window system. It uses many of the concepts, formalisms and tools presented in Chapter 2. The other chapters of the thesis focus on techniques used for the verification and deployment of software-intensive systems.

Chapter 2 gives a short introduction to the background information needed to understand the concepts in this dissertation. This includes an explanation of modelling, meta-modelling and transformations together with an introduction to several commonly used formalisms. Finally, some tools are discussed that are used to build the formalisms, models and transformation models.

Chapter 3 explores process modelling for MPM. Mega- and process modelling techniques are investigated and lessons learned from these modelling perspectives are used in the creation of an appropriate formalism. The process model for MPM is enacted using techniques from transformation chaining.

Chapter 4 shows the complete process model for the design, verification and deployment of the power window system. We discuss the formalisms and models involved in

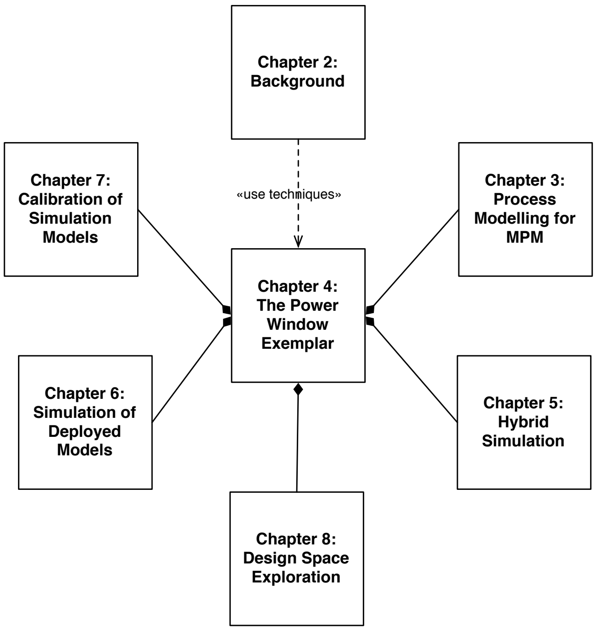

Chapter 2: Background
«use techniques»
Chapter 7: Calibration of Simulation Models
Chapter 3: Process Modelling for MPM
Chapter 4: The Power Window Exemplar
Chapter 6: Simulation of Deployed Models
Chapter 5: Hybrid Simulation
Chapter 8: Design Space Exploration

Figure 1.1: Relationships between the chapters

the process starting from the formalisation of the requirements to the code generation step.

Chapter 5 shows a multi-paradigm modelling approach to achieve hybrid simulation. A continuous time and discrete event formalism are co-simulated.

Chapter 6 evaluates the appropriateness of the DEVS formalism for the modelling and simulation of deployed software intensive systems. A prototype model is constructed and simulated for the power window system exemplar.

Chapter 7 explores the use of models for the calibration of simulation models used in the deployment process. A calibration infrastructure is automatically generated to achieve calibrated simulation models.

Chapter 8 uses model transformation techniques to explore a design space. Different techniques from the optimisation community are combined in a multi-paradigm modelling fashion.

Finally, the overall conclusions and future work are presented in Chapter 9.

Chapter 2

Background

If I have seen further it is by standing on the shoulders of giants.
— Isaac Newton

2.1 Techniques

2.1.1 Modelling and Meta-Modelling

To explicitly model domain-specific modelling languages and ultimately synthesize visual modelling environments for those, we will break down a modelling language into its basic constituents. The following is based on the *Dissecting a Modelling Language* section of the paper Domain-Specific Modelling using AToM^3 [Vangheluwe and de Lara, 2004].

The two main aspects of a model are its *syntax* (how it is represented) on the one hand and its *semantics* (what it means) on the other hand.
The syntax of modelling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet*. These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (i.e., belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language. Costagliola et. al. [Costagliola *et al.*, 2002] present a framework of visual language classes in which the analogy between textual and visual characters, words, and sentences becomes apparent. Visual languages are those languages whose concrete syntax is visual (graphical, geometrical, topological, . . .) as opposed to textual. For practical reasons, models are often stripped of irrelevant concrete syntax information during syntax checking. This results in an "abstract" representation which captures the "essence" of

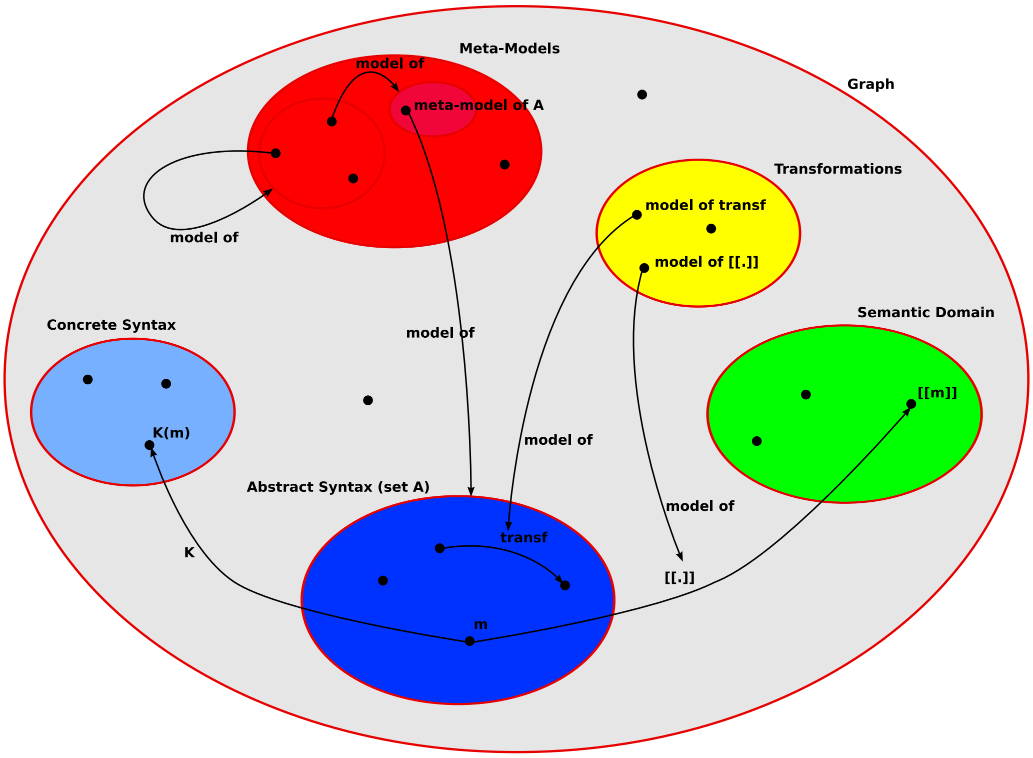


Figure 2.1: Modelling Languages as Sets, from [Vangheluwe and de Lara, 2004]

the model. This is called the *abstract syntax*. Obviously, a single abstract syntax may be represented using multiple concrete syntaxes. In programming language compilers, abstract syntax of models (because of the nature of programs) is typically represented in *Abstract Syntax Trees* (ASTs). As in the context of general modelling, models are often graph-like, this representation can be generalized to *Abstract Syntax Graphs* (ASGs). Once the syntactic correctness of a model has been established, its meaning must be specified. This meaning must be *unique* and *precise* [Harel and Rumpe, 2004] (to allow correct model exchange and code synthesis for example). Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For example, the meaning of Activity Diagrams may be given by mapping it onto Petri Nets. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own right which needs to be properly modelled (and so on, recursively). Often this domain is a set of behaviours specified in their own formalism (*e.g.*, tagged signals). In practice (in tools), the semantic mapping function maps abstract syntax onto abstract syntax.

To continue this introduction of meta-modelling and model transformation concepts, languages will explicitly be represented as (possibly infinite) sets as shown in Figure 2.1. In the figure, insideness denotes the sub-set relationship.

The dots represent models which are elements of the encompassing set(s). As one can always, at some level of abstraction, represent a model as a graph structure, all models are shown as elements of the set of all graphs Graph. Though this restriction is not necessary, it is commonly used as it allows for the elegant design, implementation and bootstrapping of (meta-)modelling environments. As such, any modelling language becomes a (possibly infinite) set of graphs. In the bottom centre of Figure 2.1 is the abstract syntax set A. It is a set of models stripped of their concrete syntax.

Meta-modelling is a heavily over-used term. Here, we will use it to denote the explicit description (in the form of a finite model in an appropriate meta-modelling language) of the abstract syntax set A of a modelling language. Often, meta-modelling also covers a model of the concrete syntax. Semantics is however not covered. In the figure, the set A is described by means of the model meta-model of A. On the one hand, a meta-model can be used to *check* whether a general model (a graph) *belongs to* the set A. On the other hand, one could, at least in principle, use a meta-model to *generate* all elements of A. This explains why the term meta-model and grammar are often used inter-changeably.

Several languages are suitable to describe meta-models. Two approaches are in common use:

1. A meta-model is a *type-graph*. Elements of the language described by the meta-model are instance graphs. There must be a *morphism* between an instance-graph (model) and a type-graph (meta-model) for the model to be in the language. Commonly used meta-modelling languages are Entity Relationship Diagrams (ERDs) and Class Diagrams (adding inheritance to ERDs). The expressive power of this approach is often not sufficient and an extra *constraint language* (such as the Object Constraint Language (OCL) in the UML) specifying constraints over instances is used to further specify the set of models in a language (adding the expressive power of first or higher order logic). This is the approach used by the OMG to specify the abstract syntax of the UML.

2. An alternative general approach specifies a meta-model as a transformation (in an appropriate formalism such as Graph Grammars [Rozenberg and Ehrig, 1999]) which, when applied to a model, verifies its membership of a formalism by *reduction*. This is similar to the syntax checking based on (context-free) grammars used in programming language compiler compilers. Note how this approach can be used to model type inferencing and other more sophisticated checks.

Both types of meta-models (type-graph or grammar) can be *interpreted* (for flexibility and dynamic modification) or *compiled* (for performance). Note that when meta-modelling is used to synthesize interactive, possibly visual modelling environments, we need to model *when* to check whether a model belongs to a language. In *free-hand* modelling, checking is only done when explicitly requested. This means that it is possible to create, during modelling, syntactically incorrect models. In *syntax-directed* modelling, syntactic constraints are enforced at all times during editing to prevent a user from creating syntactically incorrect models. Note how the latter approach, though possibly more efficient, due to its incremental nature –of construction and consequently of checking– may render certain valid models in the modelling language unreachable through incremental construction. Typically, syntax-directed modelling environments will be able to give suggestions to modellers whenever choices with a finite number of options present themselves.

The advantages of meta-modelling are numerous. First, an *explicit* model of a modelling language can serve as *documentation* and as *specification*. Such a specification can be the basis for the *analysis* of properties of models in the language. From the meta-model, a modelling environment may be *automatically* generated. The flexibility of the approach is tremendous: new, possibly domain-specific, languages can be designed by simply *modifying* parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modelling language is apparent. Above all, with an appropriate meta-modelling tool, modifying a meta-model and subsequently generating a possibly visual modelling tool is orders of magnitude *faster* than developing such a tool by hand. The tool synthesis is *repeatable*, *exhaustive* and *less error-prone* than hand-crafting. As a meta-model is a model in an appropriate modelling language in its own right, one should be able to meta-model that language's abstract syntax too. Such a model of a meta-modelling language is called a *meta-meta-model*. This is depicted in Figure 2.1. It is noted that the notion of "meta-" is relative. In principle, one could continue the meta- hierarchy ad infinitum. Fortunately, some modelling languages can be meta-modelled by means of a model in the language itself. This *meta-circularity* allows modelling tool and language compiler builders to *bootstrap* their systems.

A model m in the Abstract Syntax set (see Figure 2.1) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed. κ maps an abstract syntax graph onto a concrete syntax model. Such a model could be textual (e.g., an element of the set of all Strings), or visual (e.g., an element of the set of all the 2D vector drawings). Note that the set of concrete models can be modelled in its own right. It is noted that grammars may be used to model a visual concrete syntax [Minas, 2002]. Also, concrete syntax sets will typically be re-used for different languages. Often, multiple concrete syntaxes will be defined for a single abstract syntax, depending on the intended user. If exchange between modelling tools is intended, an XML-based textual syntax is appropriate. If in such an exchange, space and performance is an issue, a binary format may be used instead. When the formalism is graph-like as in the case of a circuit diagram, a visual (non-textual) concrete syntax is often used for human consumption. The concrete syntax of complex languages is however rarely entirely non-textual. When for example equations need to be represented, a textual concrete syntax is more appropriate.

Finally, a model m in the Abstract Syntax set (see Figure 2.1) needs a unique and precise meaning. This is achieved by providing a Semantic Domain and a semantic mapping function `[[.]]`. This mapping can be given informally in English, pragmatically with code or formally with model transformations. Natural languages are ambiguous and not very useful since they cannot be executed. Code is executable, but it is often hard to understand, analyse and maintain. It can be very hard to understand, manage and derive properties from code. This is why formalisms such as Graph Grammars are often used to specify semantic mapping functions in particular and model transformations in general.

Graph Grammars are a visual formalism for specifying transformations. Graph Grammars are formally defined and at a higher level than code. Complex behaviour can be expressed very intuitively with a few graphical rules. Since graphical rules are expressed using relations and objects, it forces the user to think in a structured way. Furthermore, Graph Grammar models can be analysed and executed. As efficient execution may be an issue, Graph Grammars can often be seen as an executable specification for manual coding. As such, they can be used to automatically generate transformation unit tests.
Not only semantic mapping, but also general model transformations can be explicitly modelled as illustrated by —transf— and its model in Figure 2.1. It is noted that models can be transformed between different formalisms.

Within the context of this thesis, we have chosen to use the following terminology.

— A *language* is the set of abstract syntax models. No meaning is given to these models.

— A *concrete language* comprises both the abstract syntax and a concrete syntax mapping function κ. Obviously, a single language may have several concrete languages associated with it.

— A *formalism* consists of a language, a semantic domain and a semantic mapping function giving meaning to a model in the language.

— A *concrete formalism* comprises a formalism together with a concrete syntax mapping function.

2.1.2 Model Transformation

As previously discussed, model transformations can explicitly be modelled, Figure 2.2 shows this. A transformation is defined at the meta-model level. From the transformation definition, a transformation is generated and executed on the model conforming to a source meta-model. The transformation outputs a model conforming to the target meta-model. It is not necessary that the source meta-model of the transformation is different from the target meta-model. This is called an endogenous transformation. An exogenous transformation has different source and target meta-models. Depending on the abstraction level of the source and target model a transformation can be vertical, when source and target are at different abstraction levels, or horizontal, when source and target model are at the same abstraction level. More information about this can be found in [Mens and Vangorp, 2006].

Most transformations approaches are based on transformation rules. Rules are composed of a Left-Hand Side (LHS), a Right-Hand Side (RHS), and optionally a set of Negative Application Conditions (NAC) patterns together with condition and action code. The LHS and NAC patterns describe the preconditions for the rule application. The RHS defines how to LHS pattern is transformed by the rule application. Extra conditions can be specified for rule application in the condition code. After a successful rule application the action code can carry out actions. The flow of execution of the transformation

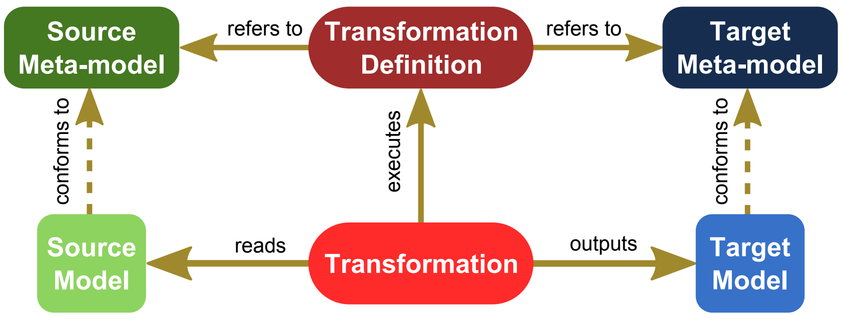


Figure 2.2: Model Transformation terminology from [Syriani, 2011]

rules is defined by the scheduler. Early graph transformation tools lacked a powerful mechanism for specifying this execution flow: any applicable rule may execute till there are no more applicable rules. However newer approaches like AToM^3 (A tool for Multi-formalism and Meta-Modelling) have extended these mechanisms with priorities or with complex control flow mechanisms. Syriani presents an overview of commonly used transformations languages in [Syriani, 2011].

Czarnecki and Helsen provide a detailed and comprehensive feature-based classification of model transformation approaches in [Czarnecki and Helsen, 2006].

2.2 Formalisms

In this section some well known formalism used in this thesis are discussed.

2.2.1 Petri nets

Petri nets, first introduced by Dr. Carl Adam Petri, is a powerful formalism used in computer science and systems engineering. Petri-nets have a well defined mathematical theory. This introduction is based on Petri-nets: Properties, Analysis and Applications [Murata, 1989]. The example is taken from [Lee and Seshia, 2011].

A Petri net is a directed, weighted bipartite graph consisting of two types of nodes: the place and the transition. The edges of the graph are either from a place to a transition or from a transition to a place. Places can be marked with a set of tokens k, by assigning a positive integer to a place. A marking is denoted by M, an m-vector with m the number of places. *M(p)* denotes the number of tokens in place p. In concrete syntax, the place is depicted with a white circle. The transition is shown using a bar. Markings are shown by drawing k number of black dots in the place p.

More formally, a Petri net is a 5-tuple, $PN = (P, T, F, W, M_0)$ where

P = $\{p_1, p_2, ..., p_m\}$ is a finite set of places

T = $\{t_1, t_2, ..., p_n\}$ is a finite set of transitions

F $\subseteq (PxT) \cup (TxP)$ is a set of arcs

W : $F \rightarrow \{1, 2, 3, ...\}$ is a weight function

M$_0$: $P \rightarrow \{0, 1, 2, 3, ...\}$ is the initial marking

$P \cap T = \emptyset$ and $P \cup T \neq \emptyset$

The behaviour of systems can be described in terms of system states and their changes. To simulate the behaviour of a system, a state or marking in a Petri net is changed according to the following firing rule:

1. A transition *t* is said to be enabled if each input place *p* of *t* is marked with at least $w(p, t)$ tokens, where $w(p, t)$ is the weight of the arc from *p* to *t*.
2. An enabled transition may or may not fire (depending on whether or not the event actually takes place).
3. A firing of an enabled transition *t* removes $w(p, t)$ tokens from each input place *p* of *t*, and adds $w(t, p)$ tokens to each output place *p* of *t*, where $w(t, p)$ is the weight of the arc from *t* to *p*.

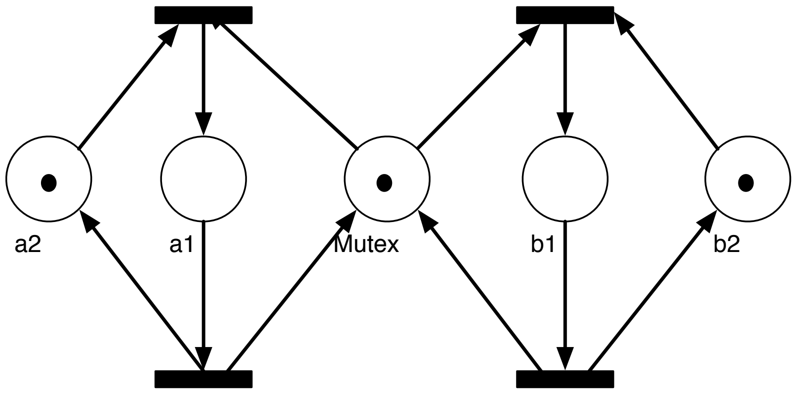


Figure 2.3: A Petri net model of two concurrent programs

An example of a Petri net is shown in Figure 2.3, that models two concurrent programs with a mutual exclusion protocol. Each of the two programs has a critical section, meaning that only one of the programs can be in its critical section at any time. The critical sections in the model are modelled with the place *a1* and *b1*. A program is in the critical section when a token is present in this place. In the figure, the two top transitions are enabled but only one can fire. Firing the left one removes the token from both the mutex place as the a2 place, and adding a token in the a1 place. Afterwards the bottom left transition becomes the only enabled transition.

Petri nets allow for a range of analysis like reachability, boundedness, etc. More information can be found in [Murata, 1989].

Sometimes we will use a particular kind of Petri nets which we call *modular Petri nets*. Modular Petri nets are an extension of the Petri Net formalism where regular Petri nets are encapsulated by *boundaries*. Those boundaries expose *ports* to the outside of the system. Petri net transitions inside the boundaries connect to the module's ports, which means they can be synchronized with other modules via a *network* model.

2.2.2 Causal Block Diagrams

Causal Block Diagrams (CBD) are a general-purpose formalism used for modelling of causal, continuous-time systems. CBDs are commonly used in tools such as MathWorks® Simulink® [MathWorks, 2013]. CBDs use two basic entities: blocks and links. Blocks represent (signal) transfer functions, such as arithmetic operators, integrators or relational operators. Links are used to represent the time-varying signals shared between connected blocks. The simulation of CBDs on digital computers requires a discrete-time approximation.

A simplified meta-model of causal block diagrams is shown in Figure 2.4. It consists of two important super classes block and port. Relations are defined between ports.

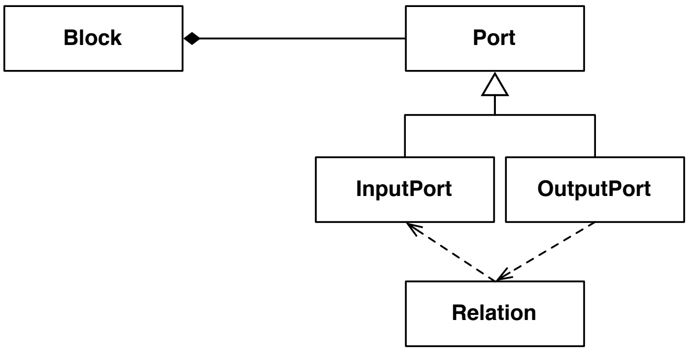


Figure 2.4: Simplified meta-model of the Causal-Blocks Diagram formalism

Different ports are defined in the meta-model. The *OutputPort* is the source of a link, it can be connected to multiple input-type ports. All other ports are input type ports. They are the destination of a link. An input-type port can only have a single incoming link. Though an output port can be connected to multiple input-type ports.

The *Block* is the super class for all defined blocks in the meta-model. Input-type ports and output ports must be connected to a single block. Depending on the block subclass, a block can have multiple input-type ports and usually a single output port associated with it. Though the *sub-model* block can have multiple output ports. The concrete syntax is very tool dependent and not the focus of this thesis. To avoid mistakes while modelling CBDs, the tool can support the user by creating the ports needed when a block is created.

Causal Block Diagrams can be mapped to ordinary differential equations when continuous-time blocks are used. In case discrete-time blocks are used, the CBD formalism can be mapped to difference equations. Another approach is to use a numerical method to solve the network, using a discrete time-step. Multiple numerical techniques are proposed in the literature to solve difference and differential equations [Press *et al.*, 1992]. In this work, the simplest technique, forward Euler, is used to solve these types of blocks. Our simulation model for CBDs uses the Python programming language and is based on [Posse *et al.*, 2002]. Simulation of CBDs involves two steps:

— Establishing an evaluation order (including the detection of loops)

— Solving the network

Solving a single timestep

To establish the evaluation order, a directed graph is constructed. The vertices in this directed graph consist only of output ports. The relation between the output ports is known by a relation, going through the block, between the output port and the input ports in the block. This relation can be of a direct feed-through type or a delayed type. When an output port has a direct feed-through relation with an input-type port, an edge is created between the output port and the output port connected to this input-type port. If the relation is a delayed one, a value for this timestep is already present at the input port of the block. The topological sort algorithm, first presented by Kahn in [Cormen *et al.*, 2001], is used to sort the ports from source to sink.

Note that in the first iteration a different dependency graph needs to be constructed. This is because the *initial condition* (IC) ports of the integrator, differentiator and delay blocks use the value on the IC port in the first iteration. The second step is done by querying the output port to produce an output value. The port in turn queries the block and calculates a new value based on its input ports. Since the blocks are solved from source to sink, the blocks can calculate the solution by querying its input ports. An example of a calculation is shown in listing 2.1.

Code Listing 2.1: Example of the calculation of a block

```
class IntegratorBlock(Block):
    def __init__(self, name, outport, inport, icport):
        Block.__init__(self, name, Block.INTEGRATOR)
        self.outPorts[outport] = Port(outport, self, Port.OUT)
        self.inPorts[inport] = Port(inport, None, Port.IN)
        self.icPort = Port(icport, None, Port.IC)
        self.influencePortMapping[outport] = [(inport, "D")]
        self.initialIPM[outport] = [(icport, "DF")]
        self.lastValue = 0
    def calculateBlock(self, iteration, port, timestep):
        if iteration == 0:
            self.lastValue = self.icPort.getLastValue()
            return self.lastValue
        else:
            influentSignal = self.inPorts.values()[0].getLastValue()
            self.lastValue = self.lastValue + influentSignal * timestep
            return self.lastValue
```

Solving Loops

A problem arises when a loop is present in the directed graph since the topological sort algorithm can only work on a directed acyclic graph. Because of this, the strongly connected components in the graph are first detected using Tarjans algorithm [Cormen *et al.*, 2001]. These strongly connected components are solved as a single block. A multitude of solvers can be used to solve these components ranging from Gauss-Jordan elimination for linear algebraic loops to iterative solvers for other types of loops. In this work, only linear algebraic loops are considered.

2.2.3 Statecharts

Statecharts are a popular visual formalism for describing reactive systems. The statechart formalism can be described in concisely by: *Statecharts = State Automata + Hierarchy + Orthogonal Components + Broadcasting [Harel, 1987]*. The following introduction is based on [Borland, 2003] and [Harel and Naamad, 1996].

The most basic elements in statecharts are blobs. Statecharts are a visual formalism and a blob is usually represented by a variable sized roundtangle. Blobs have a similar meaning to states in finite state automata (FSA). However, there are different kinds of blobs as well. This differs from the FSA specification in which there is essentially only one kind of state. These different kinds of states are called OR-blobs, AND-blobs and BASIC-blobs. These different types of blobs give rise to hierarchy in statecharts. It makes it easy to see that when a certain state is reached, some details may be left out. However, these details can be specified using OR-blobs. More specifically, a BASIC-blob is a state such that it has no sub-OR-blobs, sub-AND-blobs nor any sub-BASIC-blobs. OR-blobs are blobs that have one or more sub-OR-blobs. AND-blobs have, what are called orthogonal components. The orthogonal components of a blob are normally drawn with a dashed line and denote concurrent operations.

Transitions are used to specify system dynamics in the diagrams. A transition is simply an arrow that indicates the system can change its current state from the source state of the transition to the destination state of the transition. Transitions are normally labelled with a construct of the form $e[c]/a.$. This syntax denotes that the transition fires when event e occurs and condition c is true. The a usually means one of two different things, depending on the variant of statechart. In some definitions, $/a$ denotes that upon firing the transition, the a event is broadcast throughout the entire statechart, which may trigger more transitions. In the UML definition, $/a$ usually denotes a series of actions (such as computer code) to execute, upon firing the transition. Most statechart variants combine these two aspects. Different kinds of transitions exist. First, there is the initial transition. It is the default transitions that fires upon entering an OR-blob, AND-blob or the root. They force the diagram to be deterministic by denoting which is the current state upon entering an OR-blob. Initial states are usually drawn as a small filled-in circle with the connected transition pointing towards a blob and may only be labelled with an action (no trigger event).

Additional statechart features include history and deep history. More information can be found in [Harel, 1987].

Different semantics of the statechart formalism have been presented in the literature. In [Esmaeilsabzali *et al.*, 2009] an overview of the different semantic choices are presented. For example, MathWorks® Stateflow® is a widely used statechart product. Another popular semantics for statecharts is the STATEMATE semantics presented in [Harel and Naamad, 1996]. These semantics are used in the iLogix's statechart tool called *STATEMATE*. The tool is capable of modelling, simulating and generating code out of a statechart. The STATEMATE semantics are used in this work when referring to statecharts.

First, consider what is classified as a single step using the STATEMATE semantics. A step takes zero simulation time. That is, the clock that tracks the statecharts notion of time is not incremented during the execution of a step. Once an external event is generated, transitions that respond to this event become enabled. Recall that transitions are labelled $e[c]/a$ meaning the transition fires and action a is executed if event e occurs while condition c is true. So, if event e occurs but condition c is not true, this transition cannot be followed nor will the current state be updated. The current state refers to the state of the statechart model. If a state is current, then so is its containing state. This means the current state is actually a tree structure, extending from the root, which is always active, down to the basic states at the leaves of the tree.

If the condition c is true, the system proceeds to execute action a and update the current state to that of the transition's destination state. If the action generates another event which in turn triggers another transition to become enabled and fire, then this is also carried out within the very same step. This process is repeated every time the execution of a transitions action causes another event to fire. When no more internal broadcast events trigger more transitions to fire, the step is considered to be completed. The series of steps that leads to another stable configuration is called a macrostep while the individual steps in a series of steps are microsteps.

The fact that execution of a step takes zero time helps us realize an important distinction between simulating a statechart and generating executable code from a statechart. When simulating or animating a statechart the user may choose to speed up or slow down simulation time as the statechart knows it. The user can define the time model used upon running the simulation. Once executable code is generated, however, the time model is inherent to the statechart design and the system that runs the code. Moreover, a step clearly does not take zero time.

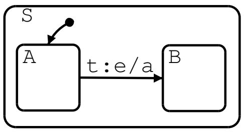


Figure 2.5: Statecharts semantics example from [Harel and Naamad, 1996]

Every operation that is carried out during the execution of a step solely depends on the status of the system at the beginning of the step. Reactions or updates from external and internal events can only be realized until after the step is completed. A maximal set of non-conflicting transitions is always executed. This means that if a transition is enabled, it will be executed as long as there is no non-determinism. It may happen that when several transitions are enabled, some may be conflicting and unable to be executed in the same step. A priority scheme is one way to resolve conflicting transitions.

In a step, the system will typically carry out operations of four types: transitions, static reactions, actions performed when entering a state, and actions performed when exiting a state. We discuss the simplest kind of step, involving a single transition in an ordinary, unadorned statechart. Consider Figure 2.5, and assume that the system is in state A and that event e has just occurred. The response of the system will be as follows:

1. Transition t becomes enabled and fires (there is no conditional constraint on this transition). The system will exit state A and will enter state B (the current state will be B after the step has completed)

2. Action a is executed.

3. More events are generated: $entered(B)$ as well as $exited(A)$

4. The condition $in(A)$ becomes false while $in(B)$ becomes true

5. The actions specified to take place upon exiting state A are executed.

6. The actions specified to take place upon entering state B are executed.

7. All the static reactions of state S that are enabled, that is, whose trigger is true, are executed. (This is because the system was in state S before the step and did not exit S during the step.)

8. All activities that were specified as being active within or throughout state A are deactivated, while those defined as being active throughout state B (but not necessarily those defined as being active within B) are activated.

More information on the STATEMATE semantics can be found in [Harel and Naamad, 1996].

2.2.4 DEVS

The Discrete EVent System specification (DEVS) formalism was conceived by Ziegler. It provides a basis for discrete-event modelling and simulation where the time base is continuous. During a certain time, only a finite number of events occur that can change the state of the system. In between these events, the state of the system remains the same. We will describe the DEVS with ports formalism, the introduction is based on [Wainer, 2009].

A system is modelled in DEVS using a composition of *Atomic* and *Coupled* DEVS components. An atomic model describes the behaviour of a discrete-event system as a sequence of transitions between states. it also describes how the system reacts to external input events and how it generates output events. The atomic DEVS model is specified as:

M = $<$ X, Y, S, δ_{int}, δ_{ext} , λ, ta $>$

where:

X = $\{(p,v)|p \in$ IPorts, $v \in X_p\}$: is the set of input events, where IPorts are the set of input ports and X_p are the set of values for the input ports.

Y = $\{(p,v)|p \in$ OPorts, $v \in Y_p\}$: is the set of output events, where OPorts are the set of output ports and Y_p are the set of values for the output ports.

S : The state set S is the set of sequential states.

δ_{ext} : $Q \times X \rightarrow S$: is the external state transition function with $Q = \{(s,e)|s \in S, e \in [0, ta(s)]\}$ and e is the elapsed time since the last state transition.

δ_{int} : $S \rightarrow S$: is the internal state transition function

λ : $S \rightarrow Y$: is the output function

ta : $S \rightarrow \mathbb{R}_0^+ \cup \infty$: is the time-advance function

The time base of a DEVS model is not explicitly mentioned but is continuous. At any given moment, a DEVS model is in a state $s \in S$. When no external event is given, the model remains in that state for the duration defined by the time-advance function (ta(s)). On expiration of this time, the model outputs the value λ(s) through a port $y \in Y$. After the output of the event, the model changes it state to a new state given by the δ_{int} function. This is called an internal transition. When an external event is received, an external transition occurs. The new state is determined by the function δ_{ext}(s,e,x) where s is the current state of the model, e is the elapsed time since the last transition , and $x \in X$ is the external event that was received. The definition of the time-advance function states that this can take a real value between 0 and ∞. When ta(s) = 0, the model will trigger an instantaneous transition, the state is called a transient state. When ta(s) = ∞, the model will remain in this state until an external event occurs. This is called a passive state.

Figure 2.6 shows an example state trajectory of a DEVS model. In the figure the system made an internal transition to state s2. In the absence of external input events, the system

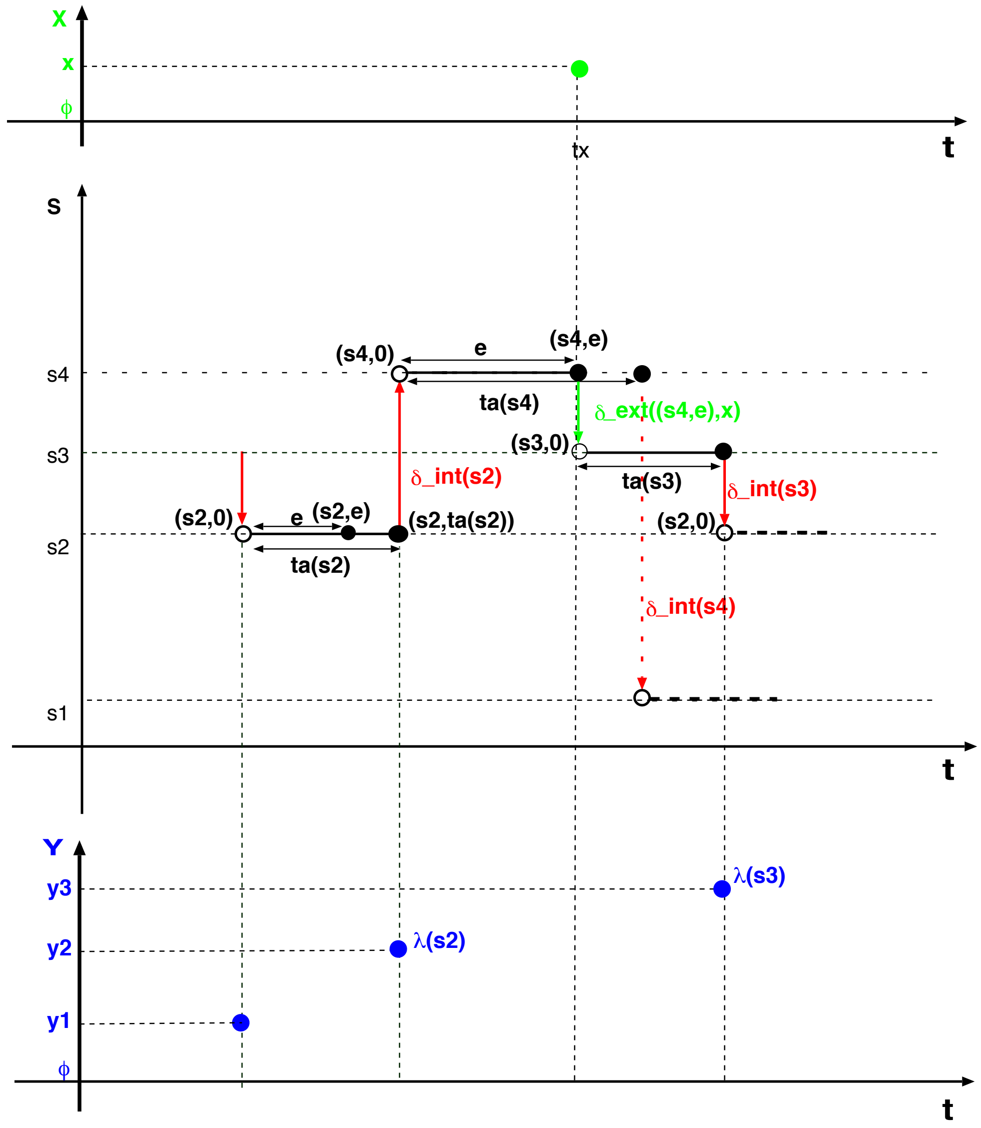


Figure 2.6: Example State Trajectory of a DEVS model, from [Vangheluwe, 2012]

stays in state s2 for a duration ta(s2). During this period, the elapsed time e increases from 0 to ta(s2), with the total state = (s2, e). When the elapsed time reaches ta(s2), first an output is generated: y2 = λ(s2), then the system transitions instantaneously to the new state s4 = δ_{int}(s2). In autonomous mode, the system would stay in state s4 for ta(s4) and then transition (after generating output) to s1 = δ_{int}(s4). Before e reaches ta(s4) however, an external input event x arrives. At that time, the system forgets about the scheduled internal transition and transitions to s3 = δ_{ext}((s4, e),x). Note how an external transition does not give rise to an output.

A coupled DEVS describes a system as coupled components of atomic DEVS models or coupled DEVS. The connections between the components denote how they influence each other: output events of one component can become, via a network connection, input events of another component. The coupled model is formally defined as:

CM = <X, Y, D, M_d |d$\in$D, EIC, EOC, IC, select>

where:

X : {(p,v)|p $\in$ IPorts, v $\in X_p$} is the set of input events, where IPorts represents the set of input ports and X_p represents the set of values for the input ports

Y : {(p,v)|p $\in$ OPorts, v $\in Y_p$} is the set of output events, where OPorts represents the set of input ports and Y_p represents the set of values for the output ports;

D : is the set of the component names and for each d $\in$ D

M$_d$: is a DEVS basic (i.e., atomic or coupled) model

EIC : is the set of external input couplings, EIC $\subseteq$ {((Self, in_{Self}), (j, in_j))|in_{Self} $\in$ IPorts, j $\in$ D, in_j $\in$ $IPorts_j$}

EOC : is the set of external output couplings, EOC $\subseteq$ {((Self, out_{Self}), (i, out_i))|out_{Self} $\in$ OPorts, i $\in$ D, in_i $\in$ $OPorts_i$}

IC : is the set of internal couplings, IC $\subseteq$ { ((i,out_i), (j, in_j)) | i,j $\in$ D, out_i $\in$ $OPorts_i$, in_j $\in$ $IPorts_j$ }

select : is the tiebreaker function, where select $\subseteq$ D $\rightarrow$ D, such that, for any nonempty subset E, select(E) $\in$ E

Figure 2.7 shows a DEVS coupled model with two subcomponents (A1 and A2). These basic models (atomic or coupled) are connected using the different ports. Because multiple components can be scheduled for an internal transition at the same time, there can be ambiguity. For example A1 and A2 both have an internal transition scheduled. Two scenarios are present:

1. Execute the internal transition of A1 first
2. Execute the internal transition of A2 first

The select function solves this ambiguity by defining an ordering over all the components of the coupled model so that one of these scenarios is chosen.

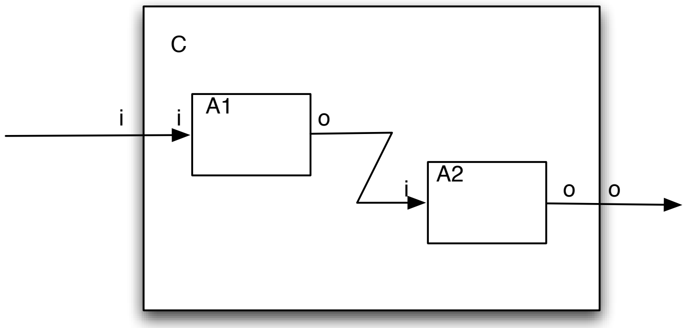


Figure 2.7: Coupled DEVS example

The DEVS formalism is closed under coupling so it is possible to construct an atomic DEVS model for each coupled DEVS model..

2.2.5 AUTOSAR

Software plays an increasingly important role in cars. About 30% of all innovations in current vehicles is related to software [Broy *et al.*, 2007]. To keep complexity under control and to create a competitive market for automotive software components, some leading automotive companies created the AUTOSAR consortium [AUTOSAR, 2012]. This consortium contains Original Equipment Manufacturers (OEM) and supplier companies. The AUTOSAR technical goals include modularity, scalability, transferability and reusability of functional components. To achieve these goals, the AUTOSAR initiative has a dual focus. On the one hand it defines an open platform (middleware) for automotive embedded software through standardized interfaces. On the other hand it provides a method to create automotive embedded systems. Using AUTOSAR, software can be developed mostly independently from the platform it will be deployed on.

AUTOSAR describes a metamodel for the deployment of automotive software components to a set of ECUs. The metamodel of AUTOSAR spans three different areas: (a) software architecture, (b) hardware and topology, and (c) ECU. The concrete syntax for the AUTOSAR language is only defined for a part of the software architecture. On the other hand AUTOSAR defines a middleware where the system can be deployed on.

A small introduction to the AUTOSAR concepts is given below. More information about AUTOSAR can be found on the AUTOSAR website. This thesis works with version 2.2.1 of the AUTOSAR standard. Though, the main concepts of AUTOSAR have remained in the newer versions of the standard.

Software Architecture Model

AUTOSAR describes software using a software component oriented approach. The component model is centred around standardised interfaces. This is to achieve scalability and transferability of these components.

The functional model of AUTOSAR consists of a set of *atomic software components*. Software components are atomic thus has to be mapped to a single control unit in the system. These components can interact with each other using *ports*. The service or data provided or required by a port are defined by its *interface*. This can be either a data-oriented communication mechanism (sender/receiver interface) or a service-oriented communication mechanism (client/server interface). The data-oriented interface can support 2 types of semantics. The first is "last-is-best", where only the last received value is stored. The other is a queued version where the data is stored in a queue until it is read.

Figure 2.8 shows an example software model, the software should switch on the lights when rain is detected. In this model three software components are present (There is a small difference between the sensor-actuator components (*Rain_Control* and *Light_Master* and the software component *Light_Logic* though this is outside this introduction.) The *Rain_Control* communicates using a sender/receiver interface with the *Light_Logic*. The *Light_Master* is a server that the client *Light_Logic* can interact with.

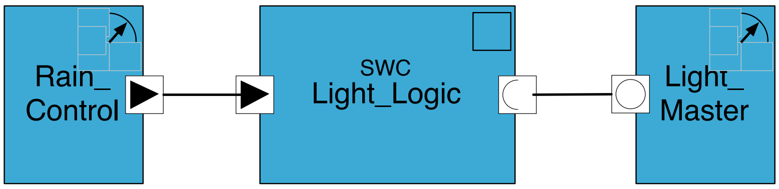


Figure 2.8: Example Software Component Model

Each software component defines its behaviour by means of a set of *runnables*. A runnable is a function that can be executed in response to *events*, for example from a timer or due to the reception or transmission of a data element. A runnable can also wait for the arrival of certain events for example when it needs another data-element to continue execution. These are called *waitpoints*. Finally, the runnable may need to update state variables, with exclusive read/write access. This is achieved using *exclusive areas*.

Each software component defines the interfaces using a *Software Component Template*. This contains all the information regarding the interfaces and the behaviour of the software component.

Hardware and Topology

The system model defines the available hardware that can be used in the system. This includes the number and types of the ECUs in the system. It also describes the communication hardware involved. Finally, a topology represents how the ECUs interact with the different communication busses (the topology).

System Configuration Model

The system configuration model defines how the software is deployed on the hardware. This includes the mapping of atomic software components to the hardware units. Signals that are communicated between software components on different ECUs have to be transmitted on a communication bus. A System Configuration for the Rain control is shown in Figure 2.9.

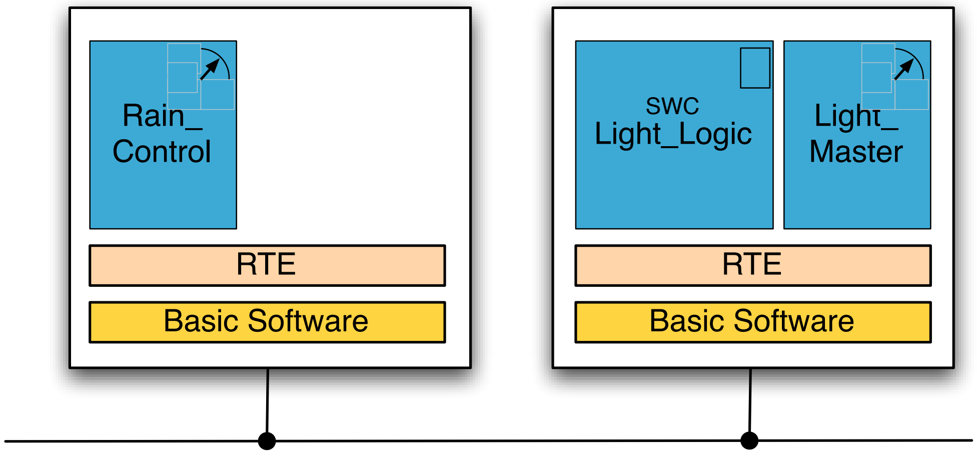


Figure 2.9: Deployed AUTOSAR example

ECU Model

To make software components independent from the hardware, the interface to this hardware must be standardized. This is done using the AUTOSAR basic software, shown in Figure 2.10.

This middleware consists of a real-time operating system based on the OSEK/VDX standard [OSEK, 2005]. The operating system schedules tasks in a fixed priority way. Some tasks can be preemptive while others are not preemptive. Since the concept of a task is not known at the functional level, the components must first be mapped to the processors and then the runnables must be mapped onto tasks. The mapping to tasks is

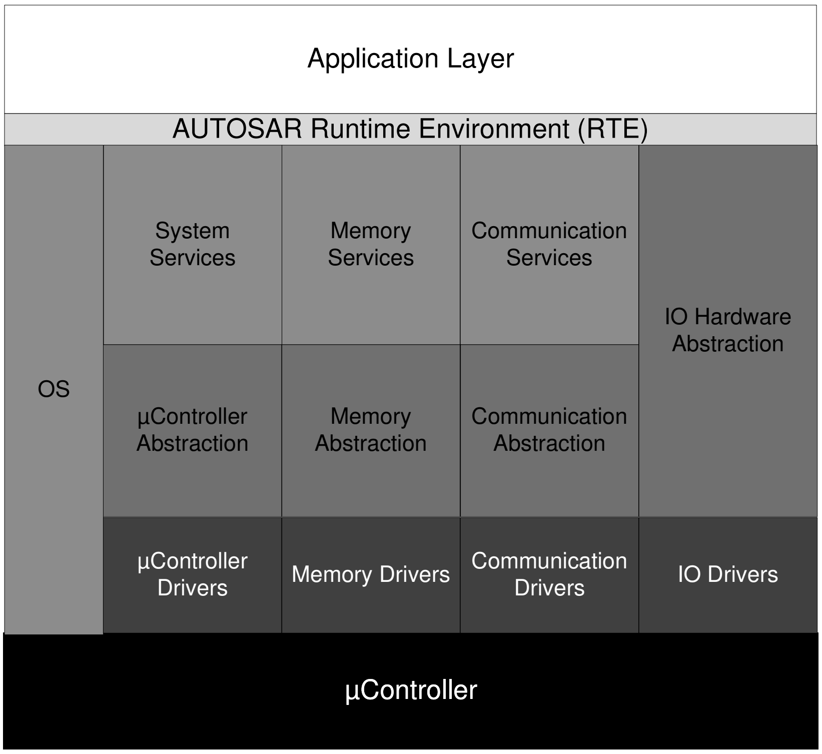


Figure 2.10: Structure of the AUTOSAR basic software

not necessarily one-to-one. The rules for mapping runnables to tasks are defined in the RTE specification, available on [AUTOSAR, 2012]. All tasks have to be assigned a priority to be scheduled by the operating system.

The middleware also contains services for sending and receiving messages on a communication bus. These are composed of signals that originate in the application layer. Communication signals and messages have certain configurable properties, such as the signal transfer property and the message transmission mode, that have an impact on the timing behaviour of the application. Table 2.1 shows the behaviour of transmitting a message based on the signal transfer property and message transmission mode when a signal is written to the COM module. For a cyclic transmission, a period is required. Other parameters are used to transmit the message multiple times or prohibit a transmission for a certain amount of time after a previous transmission.

Message Mode Signal Property	Direct	Cyclic	Mixed
Triggered	Immediate transmission	Cyclic transmission	Cyclic and immediate transmission
Pending	No transmission	Cyclic Transmission	Cyclic transmission

Table 2.1: Communication Properties of the AUTOSAR COM module

On the communication abstraction and driver layer, the most common automotive buses, for example the CAN-bus [Farsi *et al.*, 1999] and FlexRay-bus [Makowitz and Temple, 2006], are currently supported by the AUTOSAR communication stack. These also have many configuration parameters, such as the priority of the frames containing the message, that impact the real-time behaviour of the full system.

Other services exist for reading and writing values from the hardware periphery units like the Analog-Digital Converter (ADC), Pulse-Width Modulator (PWM), etc. Other services are provided for accessing memory and storage hardware.

The run-time environment (RTE) is used as a glue between the functional components and the AUTOSAR basic software. It is responsible for storing the internal messages using buffers or forwarding the external messages to the communication stack. It also activates the runnables when an event occurs.

Code Generation

Using the configuration templates, code can be generated on a per ECU basis. For each ECU an optimised middleware, containing only the required features for that specific ECU, can be generated. From the system template, the RTE code can be generated.

2.3 Tools

The tools used in this dissertation are described in this section.

2.3.1 AToM^3

AToM^3 or A Tool For Multi-Formalism and Meta-Modelling is a syntax directed editor tool. The meta- modelling layer allows a high-level description of models. Using this meta-information, AToM^3 automatically generates a tool to process these models. Model transformation can be performed on models conforming to a cross product of meta-models. Since models are represented as abstract syntax graphs (ASGs), model transformation is performed through graph transformation. The control flow mechanism for rule scheduling is limited to a priority-based flow. More information on AToM^3 can be found in [Lara and Vangheluwe, 2002].

2.3.2 AToMPM

The newer version of the AToM^3 tool is called A Tool for Multi-Paradigm Modelling and is presented in [Mannadiar, 2012]. AToMPM addresses usability, accessibility and/or scientific limitations common to popular DSM tools. In addition to several technical innovations and improvements, which include a Web-based client and support for real-time, distributed collaboration, its main scientific interest lies in its theoretically sound approach towards the specification of modelling language syntax and semantics and of model transformation rules and pre- and post-condition pattern languages. Rules and their schedules are normal models that adhere to a normal meta-model. AToMPM has an elaborate transformation execution facility: step-by-step and continuous execution modes, triple-outcome (i.e., success, not applicable, failure) rule and transformation control flow, pausing of ongoing transformations and a debugging mode are supported.

2.3.3 T-Core

T-Core is a minimal collection of model transformation primitives, defined at the optimal level of granularity presented in [Syriani and Vangheluwe, 2010]. T-Core is not restricted to any form of specification of transformation units, be it rule-based, constraint-based, or function-based. It can also represent bidirectional and functional transformations as well as queries. T-Core modularly encapsulates the combination of these primitives through composition, re-use, and a common interface. It is an executable module that is easily integrable with a programming or modelling language.

2.3.4 EMF

The Eclipse Modelling Framework (EMF) [EMF, 2012] is an Eclipse-based modelling framework and code generation facility. A model specification is described in XMI. The EMF provides the tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, UML, XML documents, or modelling tools, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications. A lot of transformation languages, both model-2-model and model-2-text, can be used in combination with EMF models.

2.3.5 IBM CPLEX

The IBM ILOG CPLEX Optimization Studio [IBM, 2009] is a mathematical optimisation package. CPLEX is used to solve linear programming (LP) and related problems like Mixed Integer Linear Programming (MILP). Specifically, it solves linearly or quadratically constrained optimization problems where the objective to be optimized can be expressed as a linear function or a convex quadratic function. The variables in the model may be declared as continuous or further constrained to take only integer values.

2.3.6 EB Tresos Studio

EB Tresos Studio is a commercial implementation of the AUTOSAR standard. It offers a tool for the configuration of an ECU based on an imported system description model. Subsequently, EB Tresos Studio can generate the embedded middleware and RTE code for an ECU from a fully configured model. Applications deployed in this thesis use this AUTOSAR implementation.

Chapter 3

A process language for MPM

As I sit down and start to work, I often panic. I stare at the empty piece of music paper. How can I say that my piece will be ready for performance next January when I do not have a recipe for making it happen?
— Lukas Foss

3.1 Introduction

In Chapter 1, the choice for a multi-paradigm modelling approach is made clear. Still, MPM does not have a standard way of representing processes. A process for MPM should make the different levels of abstraction clear. These levels of abstraction could be at the domain-specific design of the computational or physical components, the verification of the system at a high abstraction level, but also during deployment where different approximations can be used to obtain a better understanding of the parameters involved.

At the defined levels of abstractions it has to be clear what the languages and transformations are that are involved and how they are used together in a process. In recent work, Syriani defined transformations as *the automatic manipulation of a model with a specific intention* [Syriani, 2011]. Though from a tool building perspective this definition could be too restrictive, some activities require manual intervention before the activity can be completed. Therefore we define a transformation in this thesis as *the manipulation of a model with a specific intention.* A small example is the modelling of the physical part of a system. This could be regarded as the manipulation of an empty model in the physical modelling language, while the output is a non-empty model in the same language.

Since tool-support is crucial in the MDE (and MPM) lifecycle, it is key that the described process can be executed automatically. This means that a set of transformations can be scheduled after each other. Still, the manual (and semi-manual) transformations cannot be executed automatically. At this point, tools can still support the engineers by opening the correct formalism(s) in the modelling tool.

The process model has to act as a guide for the MPM design, verification and deployment of software-intensive systems. This means that the design of a large set of applications has to be described using the modelling language. The language should focus on the constructs of MPM, mainly formalisms and transformations and how these interact with each other. There should also be an explicit representation of control- and dataflow since the output of a single phase in the process does not necessarily means that the produced models are the input of the next phase. This also includes the use of control structures that allow parallel design, iterations, etc.

3.2 Available Methods, Techniques and Tools

In the modelling community, a lot of research has been done on (a) relations between different models and meta-models (megamodelling), (b) process modelling for MDE and (c) the chaining of different transformation executions.

3.2.1 Megamodelling

In [Vangheluwe and Vansteenkiste, 1996], Vangheluwe and Vansteenkiste introduce the concept of the *Formalism Transformation Lattice*. This graph represents different formalisms and transformations between them for the purpose of simulation.

In [Bézivin *et al.*, 2003], Bzevin et al. coined the term Megamodel with the definition: *A megamodel is a model with other models as elements*. In more recent work the definition is extended with *A megamodel contains relationships between models* [Barbero *et al.*, 2007]. He uses megamodelling for the purpose of model management, where it can provide a global view on the models and relations between them. It is also applied for creating traceability between models and elements within the models.

Favre proposes a different definition: *the idea behind a megamodel is to define the set of entities and relations that are necessary to model some aspect about MDE* [Favre, 2004]. The work is used to reason about relations between models. Other authors create and use the term megamodelling for different purposes.

Hebig et al. unify the different definitions and intentions of megamodelling in [Hebig *et al.*, 2011]. They propose the following definition of a megamodel: *a model that contains models and relations between them*. Figure 3.1 shows the core metamodel for megamodels. This can be extended for different purposes.

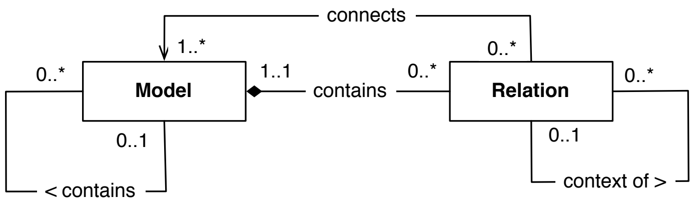


Figure 3.1: Core metamodel for megamodels as proposed by [Hebig *et al.*, 2011]

Since our primary focus is languages and transformations, the models in our megamodel will contain transformation definitions and languages. The relations are described by the input and output relations of the transformations.

3.2.2 Process Modelling

Over the years, the process engineering community has proposed various process modelling means for software development. Process modelling has a large following in research, resulting in many modelling languages. Rolland gives a definition of process modelling: *Process models are processes of the same nature that are classified together into a model. Thus, a process model is a description of a process at the type level. Since the process model is at the type level, a process is an instantiation of it. The same process model is used repeatedly for the development of many applications and thus, has many instantiations.* [Rolland, 1998]

A process model is used for three different goals:

Descriptive The models are used by an external observer to look at what happens during a process. It can be used to improve the process.

Prescriptive The models define a set of rules to prescribe a desired process. If this process is followed it would lead to a desired outcome.

Explanatory The goal of an explanatory model is to explain the rationale behind the process.

Several UML-based approaches and languages exist for process modelling. Some notable examples are given:

OMG's Software Process Engineering Metamodel (SPEM) [OMG, 2008b], formerly known as Unified Process Model (UPM), is designed for defining the process of using different UML models in a project. SPEM is defined as a generic software process language, with generic work items having different roles. It is merely a generic framework for expressing processes.

Bendraou et al. [Bendraou *et al.*, 2007] propose an approach, UML4SPM, to add execution support to SPEM2.0 by extending the SPEM standard with concepts and behavioural semantics. The process models are mapped to timed Petri nets for the purpose of model validation, and to BPEL to allow process monitoring.

Diaw et al. [Diaw *et al.*, 2010; Diaw *et al.*, 2011] integrate SPEM into MDE, and give it semantics in terms of UML state machines. The authors propose a metamodel that combines elements of SPEM 2.0, UML 2.2 and MOF 2.0. Tool support is provided to allow process enactment.

Chou et al. [Chou, 2002] present a process modelling language based on UML class diagram and activity diagrams, and then maps the models to a low-level process program.

UML 2.0 activity diagrams have been evaluated as a process modelling technique in [Russell *et al.*, 2006]. They found that it has very good support for defining control- and data flow. Though it has a lot of limitations for resource- and organization related aspects.

A comparison of several process modelling approaches and languages can be found in [Bendraou *et al.*, 2010] and in [Henderson-Sellers and Gonzalez-Perez, 2005].

Our primary focus for process modelling is a descriptive and prescriptive model where the control flow relations between the different transformations executions and the data-flow relations between the different models are clearly visible. Resource- and organizational aspects of the language are less important for our purpose. From these observations, a subset of the UML 2.0 activity diagram formalism is an appropriate choice as process modelling part.

3.2.3 Transformation Chaining Frameworks

Finally, transformation chaining is a very important technique for the automatic execution of the multi-paradigm modelling process. Different transformation chaining frameworks have been proposed in the literature, only a small set of these are discussed here.

Van Gorp et al. employ Activity Diagrams 1.0 to express chains of transformations [Van Gorp *et al.*, 2004]. Their main goals are understandability and reusability. Their notation uses regular States to denote types of models, and Object Flow States to denote transformations. The rather preliminary language uses Synchronisation Bars. They are used to denote synchronous execution (in case of multiple outputs of Synchronisation Bars), as well as multiple transformation inputs/outputs for a transformation (in case of multiple inputs of Synchronisation Bars). The language does not include decision diamonds and has no precise semantics, but is rather used as a documentation means.

[Vanhooff *et al.*, 2007] propose a transformation composition framework for MDE, where models and transformations are seen as building blocks of MDE. The approach uses a metamodel to specify transformations that can contain multiple input and output parameters. These can be connected to form chains of model transformations.

Other composition transformation frameworks like TraCo [Heidenreich *et al.*, 2011] focus on the safe composition of model transformations. They identified checks that can be performed to ensure intra- and inter-component consistency.

Lessons learned from these transformation chaining frameworks, like consistency management, can be used for enactment of our framework.

3.3 The FTG+PM

In the following paragraphs we construct the Formalism Transformation Graph and Process Model (FTG+PM). This language is used to define all relations between different formalisms using transformations, to build a process model next to this and to enact this process based on these languages and transformations. Experiences from the megamodelling, process modelling and transformation chaining communities are used to construct the FTG+PM.

3.3.1 The FTG+PM Language By Example

The language used to define FTG+PM consists of two sublanguages:

Formalism Transformation Graph (FTG) language: This language allows declaring a set of languages that can be used to model within a given domain, as well as the transformations between these different languages. This corresponds to a megamodel.

Process Model (PM) language: is used to describe the control and data flow between MDE activities. As already stated, a subset of the UML 2.0 activity diagrams are used for this.

In Figure 3.2, we present a slice of a FTG+PM model we have built using an editor, modelled and constructed using the AToMPM tool. Figure 3.2 (discussed in detail in Section 4.2) describes the artefacts and the process necessary to build software to control power windows of automobiles.

We can observe on the left side of the figure (with a light grey background), a set of domain specific languages represented concretely as labelled rectangles. Available transformations between those languages are depicted as labelled circles. The arrows from languages into transformations describe the inputs to the transformations and the arrows from the transformations into languages describe the outputs of the transformation. The FTG model is thus a graph describing the modelling languages and the transformations statically available to the engineers of a given domain.

On the right side of Figure 3.2 (highlighted in dark grey), a diagram with a part of the ordered tasks necessary to produce the power window control software is laid out. In the PM language, the labelled roundtangles (actions) in the Activity Diagram correspond to

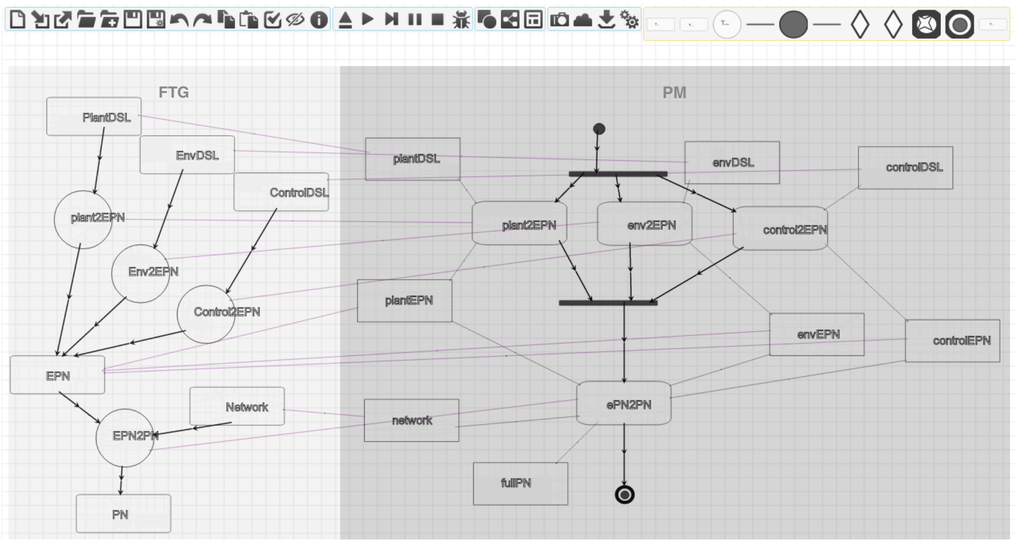


Figure 3.2: Example FTG+PM Model

executions of the transformations declared on the power window FTG. This typing relation is made explicit in Figure 3.2 by the thin horizontal links connecting the roundtangle PM action nodes to the circular FTG transformation elements. Labelled square edged rectangles (data objects) correspond to models that are consumed or produced by actions. A model is an instance of a language declared in the FTG part of the model with the same label. This typing relation is again made explicit by horizontal links connecting the rectangular PM model elements (object nodes) to the rectangular FTG language elements. Notice that in a PM model thin edges denote data flow, while thick edges denote control flow. Notice also that for each model input and output edge of a PM action a corresponding edge exists for the transformation typing it on the FTG side. The input and output models of an action are typed according to the input and output languages of the FTG transformation that types that action. Finally, the join and fork Activity Diagram flow constructs, represented in Figure 3.2 as horizontal bars, allow us to represent concurrent activities.

3.3.2 The Meta-Model of the FTG+PM

In Figure 3.3, we present a unified metamodel of the FTG and of the PM. The FTG language is presented on the left side of the metamodel. As mentioned previously, the two main concepts in the FTG are *Language* and *Transformation*. The signature of a transformation includes zero or more input languages, and one output language. Both the languages and transformations have a *definition* attribute. This attribute points to the definition of the language or transformation. A transformation has another attribute, *auto*, to define whether the transformation is automatic or done manually.

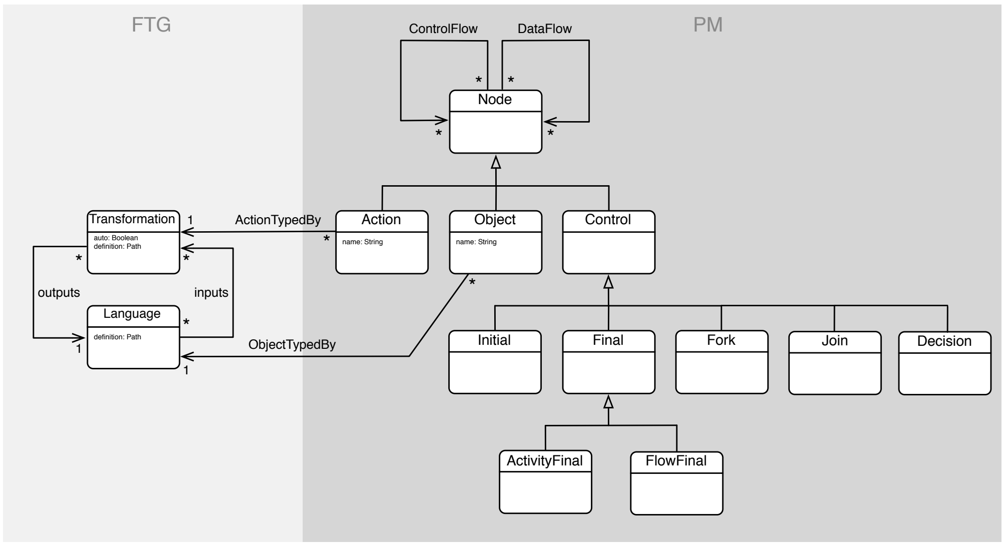


Figure 3.3: Formalism Transformation Graph and Process Model (FTG+PM) Metamodel

On the right side of Figure 3.3, the metamodel of the PM language can be seen. As mentioned previously the language is a subset of UML Activity Diagrams 2.0, with the additional feature that an *Action* is typed by a FTG *Transformation*, and an *Object* is typed by a FTG *Language*.

3.3.3 Intended Semantics of the FTG+PM Language

The intended operational semantics of the FTG+PM language are the same as those of the activity diagram. The action nodes, denoting a transformation have to be executed in the correct order. This *order* is based on the mapping of the activity diagram to coloured Petri nets as described in [Knieke and Goltz, 2010]. When an action node is encountered the action has to be executed. Depending on the state of the *auto* attribute in the transformation, the framework has to:

— false: open a modelling environment containing the input model(s) in the specified language(s) and the modelling environment of the output language

— true: automatically execute the transformation with the desired input models

This can result in the consecutive execution of different transformations. Our framework thus needs to support the chaining of transformations.

3.3.4 Chaining Transformations in the FTG+PM Language

The proposed FTG+PM language is partly implemented in the AToMPM tool. AToMPM contains its own transformation language. Transformations and transformation rules, in AToMPM, are treated as normal models conforming to an appropriate meta-model. Transformation rules, consisting of a left hand side (LHS), a right hand side (RHS) and a set of negative application conditions (NAC), are tried in an order given by a rule scheduling model, in this case described in a finite state automaton-like formalism. Since transformation rules and their scheduling are explicitly modelled within AToMPM using appropriate meta-models, defining higher-order transformations is straightforward

To execute a FTG+PM model, we transform the PM to the native transformation scheduling language of AToMPM. This is done as follows: (1) a PM *Action* node corresponds to the execution of a transformation defined in the FTG *Transformation* node typing it; (2) transformations are scheduled according to the control-flow defined in the PM. An example rule of this transformation from FTG+PM into AToMPM's transformation scheduling language is shown in Figure 3.4. Note that the LHS of a rule matches a pattern in the input model including a PM *Action* (round-tangle) typed by an FTG *Transformation* (circle), while the RHS rewrites it by building the scheduling of the transformation execution as a double round-tangle (a composite transformation application in AToMPM's rule scheduling language). The double round-tangle is then used to execute this transformation within the AToMPM environment. The scheduling language additionally includes single round-tangled nodes, corresponding to the execution of a single rule, and control flow arrows to impose the ordering of the scheduling of the transformations.

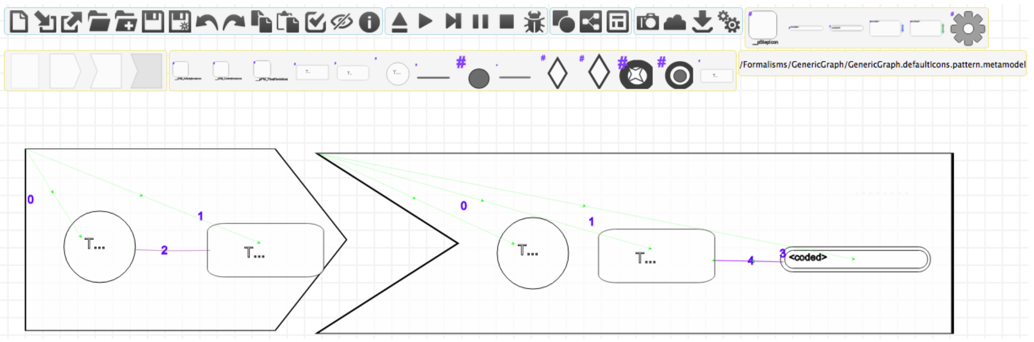

/Formalisms/GenericGraph/GenericGraph.defaultIcons.pattern.metamodel
0
1
2
T...
T...
0
1
T...
T...
4
3
<coded>

Figure 3.4: Transformation Rule to Map an Action Node to a Transformation

When executing a FTG+PM model, the input of a scheduled transformation depends on whether there are incoming data flow arrows: (a) if there are incoming data flow arrows into the action node, for each of these data flow arrows a transformation rule is created that opens the specified model in the appropriate formalism. The transformation rules that open the specified models are scheduled before the execution of the transformation defined by the action node; (b) If there are no incoming data flow arrow, the result of the previous transformation (present on AToMPM's modelling canvas) is used as the input.

A similar solution is used for the output of an action: (a) when a data flow arrow emanates from an action node, a transformation rule is created to save the model (specified in the location by the object node) and clears the modelling canvas. The transformation rule is scheduled after the transformation defined by the action node; (b) In case no data flow arrow exits the action node, the canvas is not cleared.

In the current implementation, there is no support for the (semi-) parallel execution of fork and join nodes since the current transformation language in AToMPM does not support this. Instead, the transformation towards the AToMPM transformation language sequentialises the different branches between the joins and the forks. This is done in the same way as described in [Bottoni and Saporito, 2009] where a marker is made at the top of the fork. Another marker is used to follow the chain until the join node is found. Afterwards the full branch is scheduled before the join node. This is done until all branches are sequentialised. When nesting occurs, the inner fork/join pairs are first sequentialised. Since the canvas can be used as the input for the next action node, the state of the canvas has to be saved before the fork node. This is done by inserting an object node, connected to the action node before the fork node. The output goes to the first actions of each of the branches after the fork. At the last action of each branch, a similar object node is inserted that is connected to the first action after join node. Because all models are saved and closed after the action node has executed and reloaded before starting a new action, the sequentialised process model preserves the original semantics.

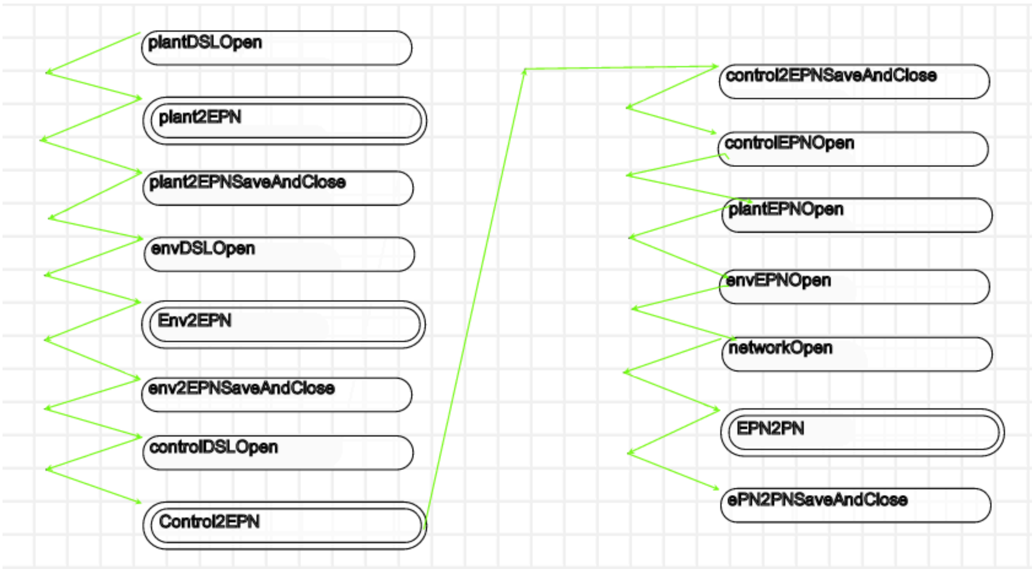


Figure 3.5: Result of the transformation to the native AToMPM transformation language

Figure 3.5 show the result of the transformation to the native AToMPM transformation language of the FTG+PM of Figure 3.2. This transformation can be executed to enable the automatic execution of a chain of transformations.

3.4 Related Work

A comparison of existing frameworks, both from the process modelling community and the transformation chaining frameworks, is done on basis of the presented requirements in the introduction. First, whether the approach has an explicit representation of the modelling languages and relations between the languages by means of transformation definitions. Secondly, whether the approach allows the composition of chains by means of an explicit representation of the process. Finally, we consider both automatic transformations where the execution of the transformation is completely automated and manual transformations where a modelling environment is set-up in the defined language(s). Table 3.1 shows the comparison of the different approaches. The ✓means that the feature is present in the tool, x means that it not present while $\sim$ means that it is unknown.

Table 3.1: Comparison of the approaches

Tool	Explicit Megamodel	Explicit Process Model		Transformations	
		Control Flow	Data Flow	Automatic	Manual
Oldevik et al. [Oldevik, 2005]	✓	x	✓	✓	~
Vanhooff et al. [Vanhooff and Baelen, 2006]	✓	x	✓	✓	x
UniTI	✓	x	✓	✓	x
TraCo	✓	x	✓	✓	x
Wagelaar [Wagelaar, 2006]	x	x	✓	✓	x
MoTCoF	~	x	✓	✓	x
Wires*	x	x	✓	✓	x
transML	✓	x	✓	✓	x
Epsilon	~	x	✓	✓	~
MCC	x	x	x	✓	x
Aldazabal et al. [Aldazabal *et al.*, 2008]	x	✓	✓	✓	~
Diaw et al. [Diaw *et al.*, 2011]	✓	x	✓	✓	~
FTG+PM	✓	✓	✓	✓	✓

Most approaches allow for the data-flow composition of model transformations where input- and output relations of the transformations are used to chain different transformations. The control-flow of these approaches is inferred from this data-flow composition. Oldevik proposes a framework for the data-flow composition of transformations in [Oldevik, 2005]. It uses UML activities to model these relations, though control flow is not taken into account. A definition for manual transformations is present, though it is not described how the framework copes with these transformation types. In [Vanhooff and Baelen, 2006], a data-flow composition of transformation framework is presented similar to the UniTI framework [Vanhooff *et al.*, 2007]. The concepts of these frameworks are extended by the TraCo framework [Heidenreich *et al.*, 2011] where additional validation checks are performed on the composition of the transformations. Wagelaar [Wagelaar, 2006] presents a DSL for the composition of transformations. The models are transformed to Ant scripts for the Ant software build process tool for execution. Seibel et al. present the MoTCoF framework [Seibel *et al.*, 2012] for the data-flow and context composition of model transformations. The meta-model of the approach is not shown, but most likely an explicit megamodel is present. Wires* [Rivera *et al.*, 2009] provides a graphical language for the orchestration of ATLAS Transformation Language (ATL) model transformations. It has modelling elements for complex data-flow for example decision nodes, parallel execution and support for loops. It does not however take manual activities into account.

The transML framework [Guerra *et al.*, 2010] is created for transformations in the ’large’. It provides meta-models for requirements, analysis, architecture and testing of transformations. The tool supports data-flow chaining of transformations by transforming to Ant scripts. The Epsilon Framework, presented in [Kolovos *et al.*, 2008], provides a model management framework where Ant scripts can be used to construct chains of transformations. It is not clear whether the Generic Model Manipulation Task can be used for the loading of a modelling environment though models can be loaded and stored using Ant scripts. Finally, Kleppe proposes a scripting language MDA Control Center (MCC) [Kleppe, 2006] for combining multiple transformations in sequence and in parallel.

In the process modelling community, frameworks for MDE are proposed as well, though these usually do not focus on transformation chaining, for example [Chou, 2002; Bendraou *et al.*, 2007]. Two examples however do take transformation chaining into account. In [Aldazabal *et al.*, 2008], Aldazabal et al. present a framework for tool integration where transformations can be chained. The process is modelled in SPEM or BPMN (Business Process Modelling Notation) and is transformed to BPEL (Business Process Execution Language) for execution support. They do not however have a megamodel to validate input-output relations. In [Diaw *et al.*, 2011], Diaw et al. present an adaptation of SPEM for the use in an MDE context. The composition is a data-flow composition like most transformation chaining approaches discussed above. Both frameworks allow the modelling of manual activities, though it is not clear how the frameworks handle these manual activities.

The FTG+PM combines the explicit modelling of the languages and transformations (megamodel) together with a process model that supports complex control-flow constructs. This allows the modelling of non-linear transformation chains for building complex applications. Transformations can either be executed automatic or require manual intervention. In the manual case the framework opens a modelling environment for the activity and continues the process when the activity is finished. The explicit modelling of all these components allows to reason about these complex chains of transformations.

We, however, did not take an a-posteriori integration of the different tools into account. We refer to Biehl et al. [Biehl *et al.*, 2012] for an investigation of the problems and solutions when considering an a-posteriori integration. It is noteworthy that SPEM is used for the design of custom tool-chains [Biehl and Törngren, 2012]. Model transformations assist the tool-chain developer to create a Tool Integration Language (TIL) model from the SPEM model. Semi-automated techniques for the specification, analysis and synthesis, support the development of tool chains that are described as TIL models.

3.5 Conclusion

In this chapter, we presented a platform for carrying out formalism transformations within MPM. We proposed the Formalism Transformation Graph (FTG) and the Process

Model (PM) to drive the MPM development process. Since the enablers of MPM are model transformations and language engineering, the FTG comprises of formalisms as nodes and transformations as edges, and shows the different languages that need to be used at each level of development for modelling at the right level of abstraction. As process modelling part, a subset of UML 2.0 has been chosen to focus on the control- and data flow in the process.

A small part of the FTG+PM is enacted by the use of model transformation. The FTG+PM language can be transformed to the native AToMPM transformation language that can chain the different transformations in our tool.

Chapter 4

Design, Verification and Deployment of the Power Window Case

Design is a funny word. Some people think design means how it looks. But of course, if you dig deeper, it's really how it works.
— Steve Jobs

4.1 Introduction

The power window controller is a complex, time-critical, safety-critical, hard real-time embedded system. When given the task to build the control system for a power window, two components need to be considered: (1) the physical power window itself, which is composed of the glass window, the mechanical lifting mechanism, the electrical motor and some sensors for detecting for example window position or window collision events; (2) the environment with which the system (controller plus power window) interacts. This includes both human actors and other subsystems of the vehicle, e.g., the central locking system or the ignition system.

The level of abstraction is associated with the task to be accomplished and is determined by the perspective on the system, the problem at hand, and the background of the developer. At a high level of abstraction, the tasks in our MPM process are the development activities starting from requirements to code synthesis. The detailed tasks are declared as transformation definitions in the FTG and instantiated as activities in the PM.

The MPM process comprises several activities with models at different abstraction levels. Models at a higher level (starting from requirements models and DSMs) are refined until the executable model level (C source code) is reached. For each new modelling language, concrete syntax needs to be defined in addition to abstract syntax, tailored to the domain expert working on the specification of that model.

The approach integrates *multi-view modelling* by building distinct and separate models of the power window system to model different aspects of the system for different intentions. Systematically and automatically deriving models of different complexity significantly increases productivity as well as quality of models. Though, this statement is hard to prove since it requires extensive empirical evaluation. The success of tools, like MathWorks® Simulink®/ Stateflow®, that uses a similar approach, can be regarded as some anecdotal evidence. We will also during this thesis show some more anecdotal evidence, in particular, the deployment activity follows this principle.

In the next few sections we will go through a possible set of model driven engineering activities when building the software *controller* for the power window system. We will start by formalising the requirements and start the modelling activities which involve developing domain specific languages for defining the *plant* and the *environment*. We will then carry on to explain how models for verification, simulation, deployment and finally code generation can be achieved from the three initial models.

4.2 Overview of the Power Window Exemplar

In Figure 4.2, we depict a condensed version of the FTG+PM model we have built for developing the Power Window software controller. At the top, requirements are modelled and further refined. After the requirements are modelled, domain-specific modelling is used to create a model of the plant, environment and control. These are combined using a model of the network, showing how the models communicate. From these domain-specific models a verification is done using Petri nets. In parallel, hybrid simulation is used to check the behaviour of the power window. When the design satisfies the requirements, deployment starts. This is done by transforming the control model of the power window to an AUTOSAR component model. An infrastructure is generated to create a performance model of the different software components in this AUTOSAR component model. The performance model is used in the deployment space exploration phase, where three levels of approximations are used to create a feasible (and optimal) solution. The final phase consists of generating the embedded code for the different electronic control units. The power window FTG+PM was built based on our experience with a concrete, AUTOSAR-based physical realisation of a power window shown in Figure 4.1. The physical realisation was created together with a colleague, Pieter Ramaekers. The verification and simulation activities of the case study are based on the experience of Pieter Mosterman and Hans Vangheluwe. These parts are based on the work presented by Mosterman and Vangheluwe [Mosterman and Vangheluwe, 2004].

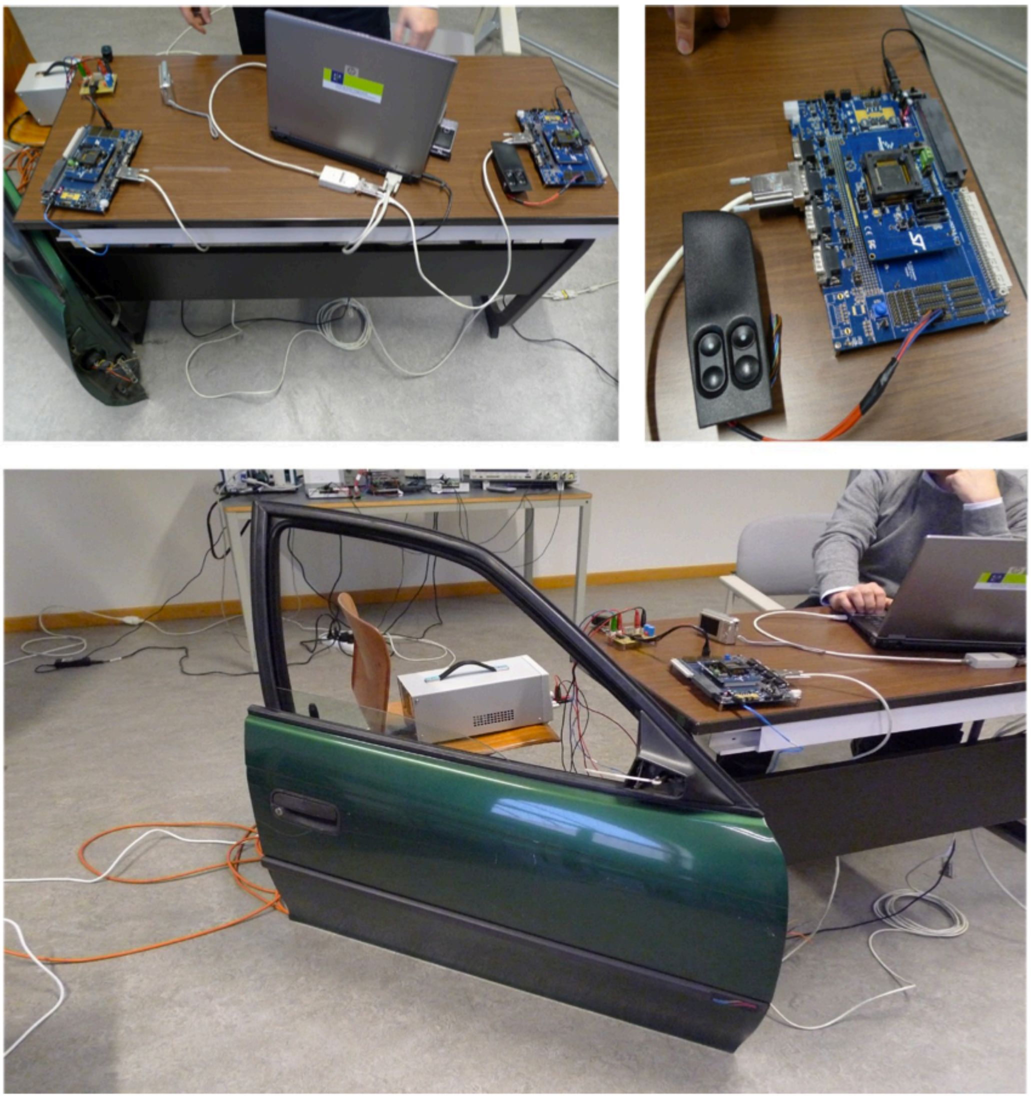
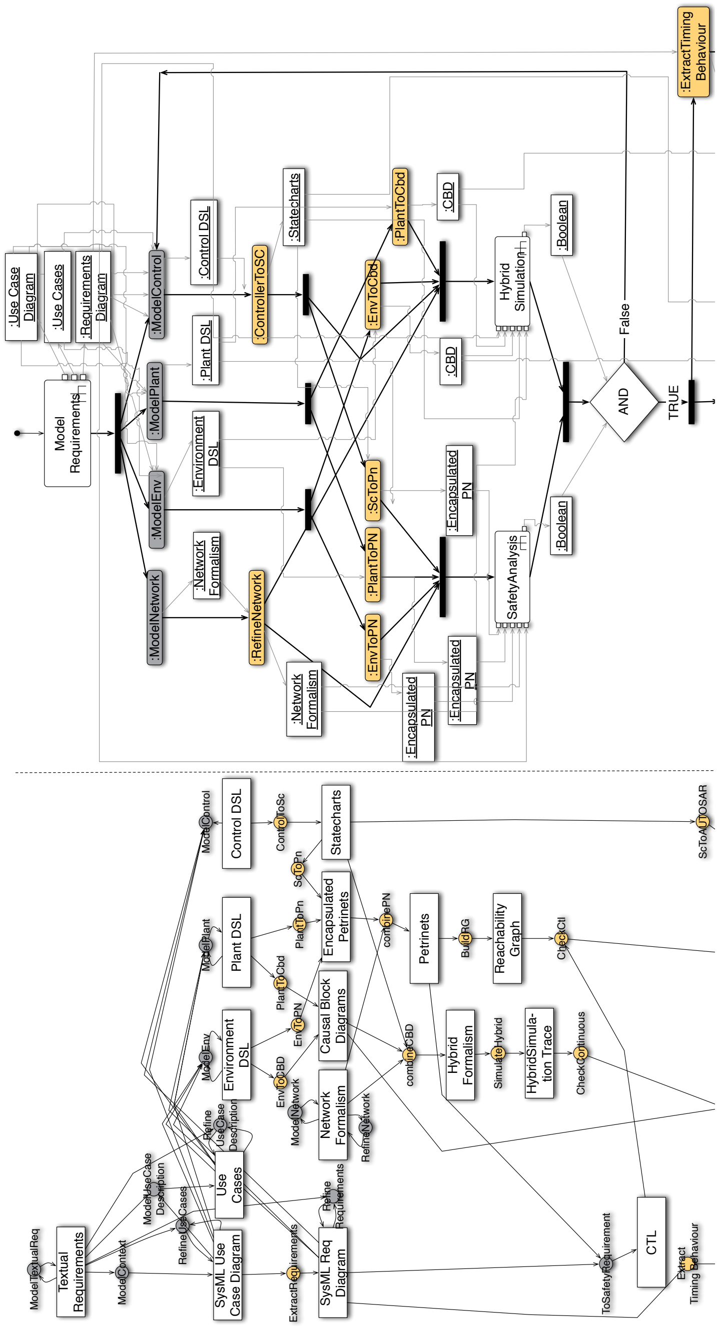


Figure 4.1: The physical implementation of the power window system


Model Requirements
:Use Case Diagram
:Use Cases
:Requirements Diagram
:ModelControl
:ModelPlant
:ModelEnv
:ModelNetwork
:Control DSL
:Plant DSL
:Environment DSL
:Network Formalism
:ControllerToSC
:RefineNetwork
:Statecharts
:PlantToCbd
:EnvToCbd
:ScToPn
:PlantToPN
:EnvToPN
:CBD
:CBD
:Encapsulated PN
:Encapsulated PN
:Encapsulated PN
:Network Formalism
Hybrid Simulation
SafetyAnalysis
:Boolean
:Boolean
AND
False
TRUE
:ExtractTiming Behaviour
ModelTextualReq
Textual Requirements
ModelContext
SysML Use Case Diagram
RefineUseCases
ModelUseCase Description
Use Cases
Refine UseCase Description
ExtractRequirements
SysML Req Diagram
Refine Requirements
ModelControl
Control DSL
ModelPlant
Plant DSL
ModelEnv
Environment DSL
ControlToSc
Statecharts
ScToPn
PlantToPn
PlantToCbd
EnvToPN
EnvToCBD
ModelNetwork
Network Formalism
RefineNetwork
Encapsulated Petrinets
Causal Block Diagrams
combinePN
combineCBD
Petrinets
BuildRG
Reachability Graph
CheckCI
Hybrid Formalism
SimulateHybrid
HybridSimulation Trace
CheckContinuous
ToSafetyRequirement
CTL
Extract Timing Behaviour
ScToAUTOSAR

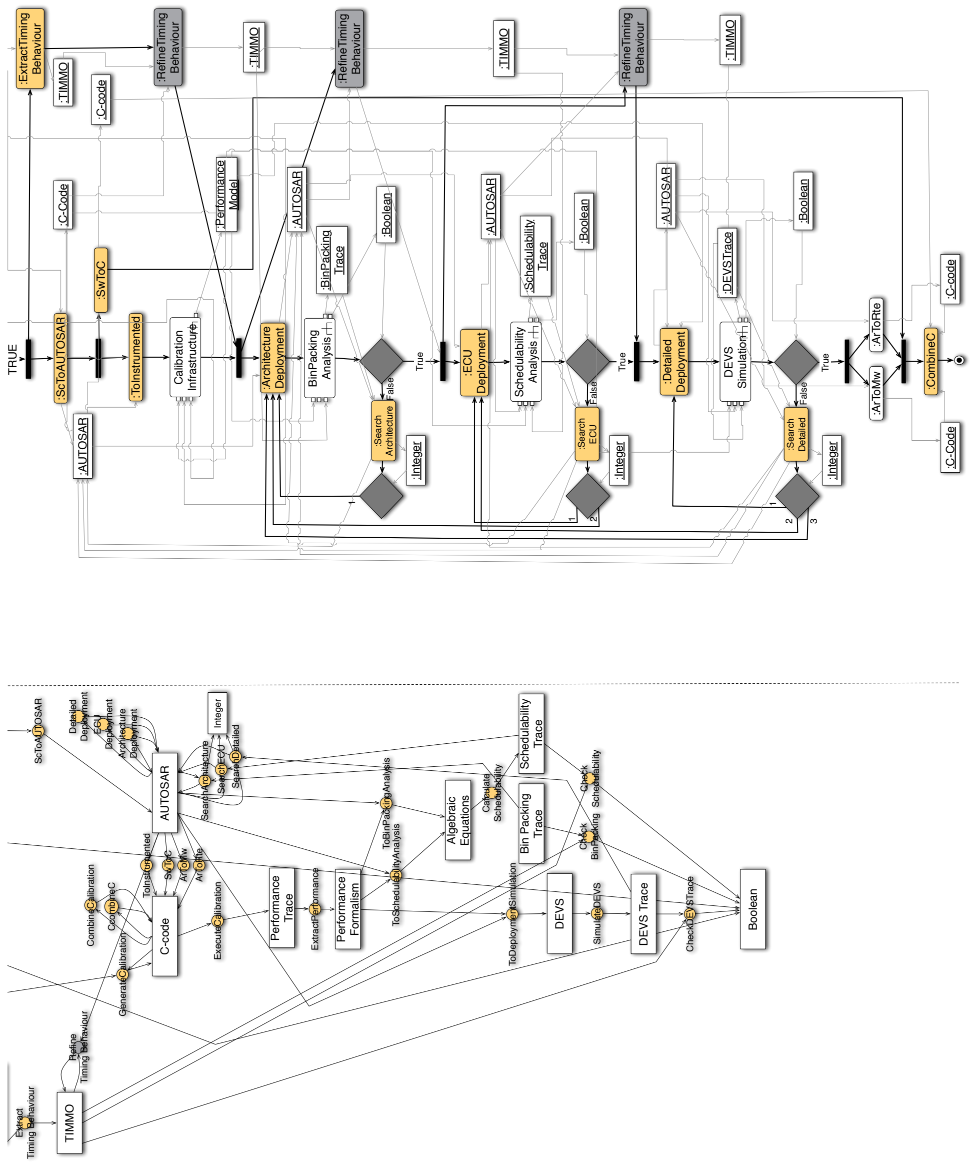


Figure 4.2: Power Window: FTG and PM

The power window system presented here has two windows, a driver side and passenger side window. The driver can control both the driver and passenger side windows while the passenger is only allowed to control the passenger side window. The passenger window conforms to the requirements shown in Section 1.3. The driver window does not have to take the object detection into consideration. The driver can also deactivate the operation of the window (lockout).

4.2.1 Requirements Engineering

Before any design activities can start, requirements need to be formalised so they can be used by engineers. SysML is used to formalise these requirements. The choice for using SysML[OMG, 2010] is pragmatic since it not only focusses on the software aspects of a system but also on hardware and other system aspects. Still, only the requirement part of SysML is used in this work to formalize the requirements.

Starting from a *textual description* containing the features and constraints of the power window, a context diagram is modelled as a *SysML use case diagram*. The use case diagram is shown in Figure 4.3.

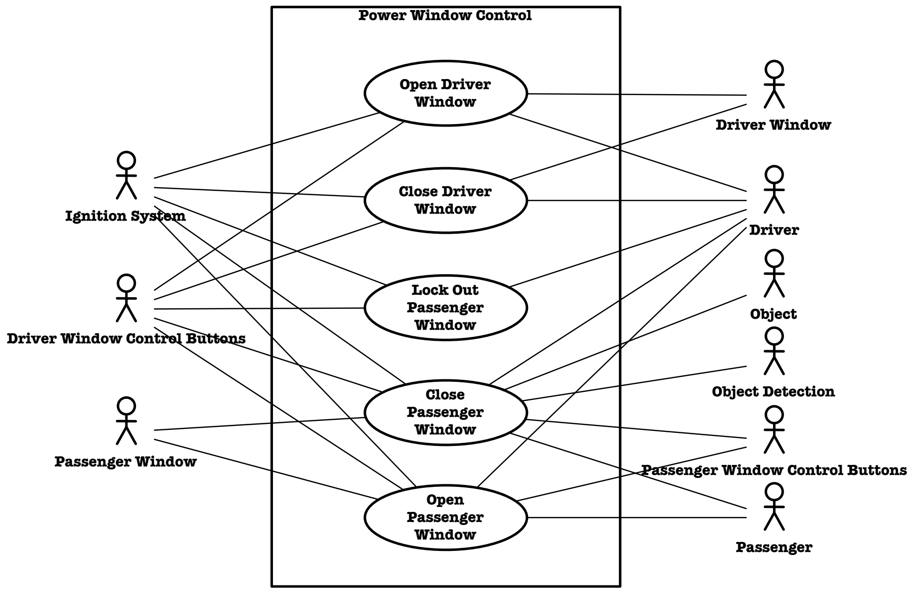


Figure 4.3: Power Window: SysML Use Case Diagram

The use cases are further refined and complemented with *use case descriptions*. The use case description of raising the window is written below. The others can be found in Appendix A.

Use Case 4	**Raise Passenger Window**
Scope:	System-wide
Level:	The Passenger or Driver wants to raise the passenger window by pushing the raise window button
Actors:	Driver (A0), Passenger (A1), Driver Window Control Buttons(A2), Passenger Window Control Buttons (A3), Passenger Window (A4), Object (A5), Object Detection (A6)
Preconditions:	The Ignition System is turned to the start, on, accessory position; the window is not locked out
Postconditions:	The window stopped

Main Success Scenario:

1. Passenger A1 pushes the raise button using A3.
2. Passenger Window A4 goes up.
3. Passenger A1 releases the raise button A3.
4. Passenger Window A4 stops.

Alternative Scenario:

1. Driver A0 pushes the raise button using A1.
2. Passenger Window A4 goes up.
3. Driver A0 releases the raise button using A1.
4. Passenger Window A4 stops.

Alternate Scenario, Misuse Case:

1. Passenger A1 pushes the raise button using A3 or Driver A0 pushes the raise button using A1.
2. Passenger Window A4 goes up.
3. Object (A5) is trapped between the window and is detected by the Object Detection (A6).
4. Passenger Window A4 goes down for 12.5 cm and stops.
5. Passenger A1 releases the raise button A3 or Driver A0 releases the raise button using A1.

Special Requirements:	— The window reacts within 200 ms. — The window is fully closed within 4 s. — Driver commands have priority over Passenger commands. — The maximum force exerted on the object is 100 N.

Technology and Data Variations List:

1. Detection via Hall effect
2. Detection via Infrared
3. No Detection Mechanism
4. Rocker switch
5. Push-pull switch (Government allows this type without object detection)

In the remainder of this dissertation we will focus on verifying the requirement presented in Use Case 4, alternative scenario: Misuse case. An environmental model will be constructed to check this particular case. During deployment, we will focus on the requirement that the window responds within 200 ms after issuing the command.

4.2.2 Modelling Activities

We consider that during the development of a software controller for a power window it is necessary to take into consideration both the description of the physical hardware itself – the *plant* – as well as the description of the environment interacting with the power window. While the fact that we need to take the plant into consideration when building a controller is self explanatory, the *environment* requires further analysis. The largest concern is the situation when somebody becomes unintentionally physically caught by a closing power window. Some power window systems include automatic reversal systems which detect if an object is blocking the windows path and automatically stops the window's movement. Other systems do not include the automatic stopping and backing system and a number of other strategies are put into place such that accidents where somebody becomes caught by the power windows (typically children) do not happen. Another strategy is for example the use of other types of buttons. In order to automatically verify the power window and the car in general has correctly implemented those strategies we will take into consideration the actors in the environment (e.g. adults, children) and model their interactions with the power window and relevant parts of the car.

We introduce two domain specific languages that allow us to describe power window *plants* and *environments*. The control model is specified using a statechart-like formalism. The presented modelling languages incorporate the different design choices. In the next paragraphs we describe the domain specific languages and one of its *models*. The semantics of these domain-specific languages are given by means of a transformation to another formalism.

Note that in the text that follows all the DSLs' grammars are described using metamodels expressed as class diagrams. Metamodels essentially identify and define the components of a language (described as *classes*), and the relations between those components (described as *associations* between classes). Note also that, because we have taken a modular approach to the development of the power window software, our formalisms allow encapsulation in the sense that the models of the different languages will be contained inside *boundaries*. These *boundaries* contain *ports* that are linked to the objects inside. The formalism inside the boundary can communicate with the outside world using these ports. In order to compose models describing different aspects of the power window, in what follows we also describe a *Network* language that connects *ports* of multiple formalisms together.

Power Window Description Language

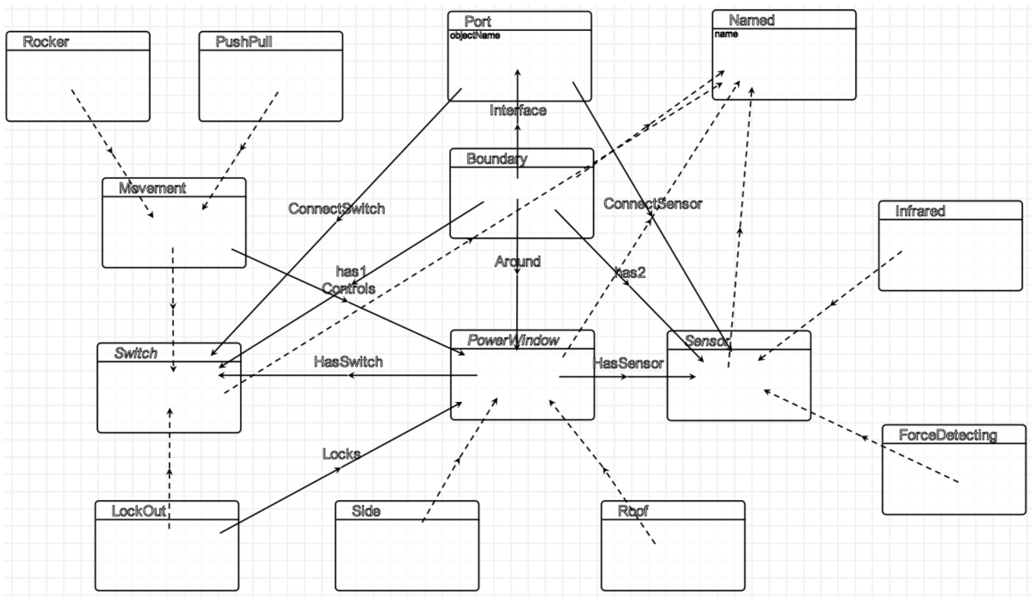


Figure 4.4: Power Window Description Language Metamodel

Figure 4.4 presents the metamodel for the language to describe the *plant* for the power window. The language allows the modelling of different design choices, like type of buttons, how lock-outs are connected, what type of sensor is used, etc.

The main class of the language is the *PowerWindow* class, which is abstract and is specialized as a *side* window or a *roof* window. A *PowerWindow* includes a set of switches that can be of two kinds: *Lockout* switches prevents control from other *PowerWindows* in the car (as specified by the *controls* association); *Rocker* or *PushPull* switches which allow controlling window movement. *Rocker* and *PushPull* switches have different physical characteristics: while *Rocker* switches are used to activate window movement by being acted upon on the horizontal axis, *PushPull* switches need to be acted upon on the vertical axis by being pushed down or pulled up. Finally, a *PowerWindow* may also have sensors for detecting if an object is blocking the window from going up. These sensors may be of two types: *Infrared* or *ForceDetecting*. *Infrared* sensors detect an object when a light beam is crossed; *ForceDetecting* sensors make use of the fact that if an object is being pushed up by the window then more current will be drawn by the electric motor.

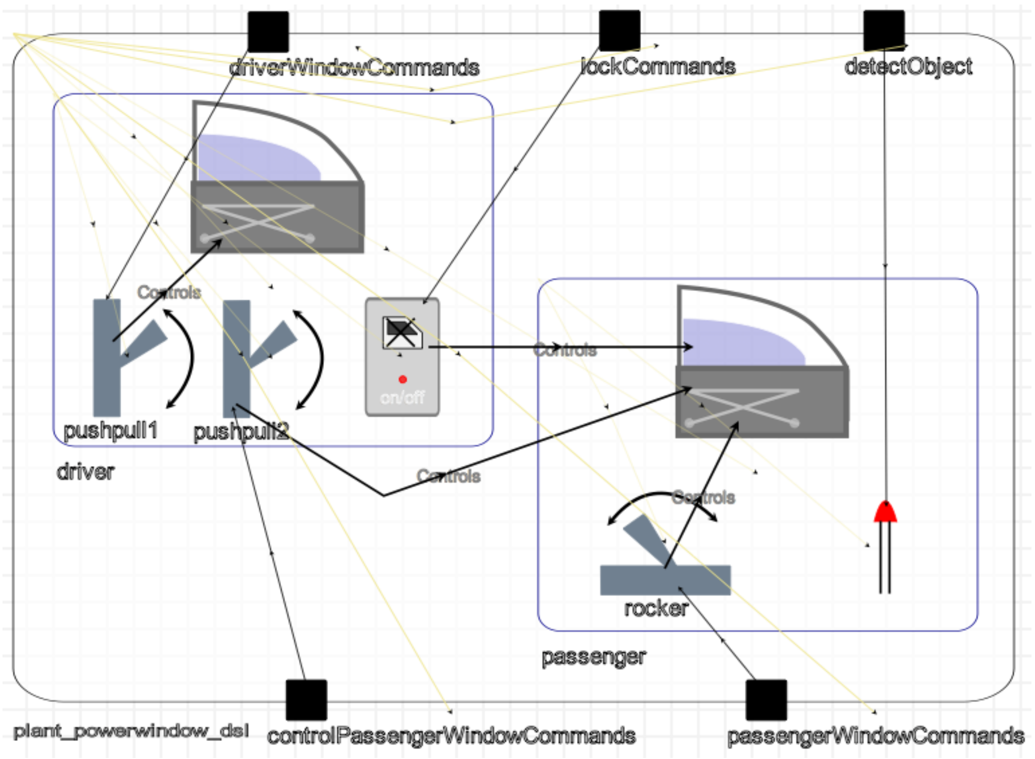


Figure 4.5: Example Model in the Power Window Description Language

In Figure 4.5 we present a model of the power window plant, where a configuration of two windows of an automobile are described. The model includes a *driver* and a *passenger* window, where the driver's window has three buttons: a push-pull button for controlling the driver's window, a push-pull button for controlling the passenger's window, and a lockout switch for disabling/enabling the control of the passenger's window. The passenger's window includes a rocker button and an infrared sensor meaning the window automatically stops lifting when an object obstructs its path.

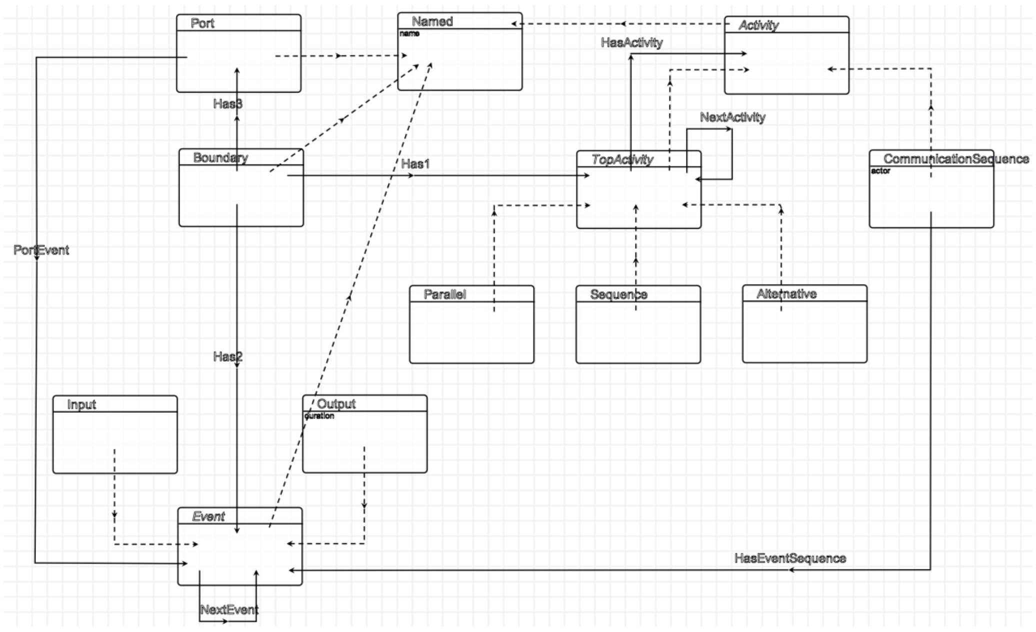


Figure 4.6: Environment Description Language Metamodel

Environment Description Language

The *Environment* description language, whose metamodel is described in figure 4.6, allows describing interactions of the outside world with the power window system. Models in the environment language implement test cases for the different requirements. The models give inputs to the different inputs of the system. Inputs have to be generated in sequence and/or in parallel. For example, to test whether the window goes down 12.5 cm when an object is detected and the window is going up, a command needs to be given to raise the window first and then a command to present an object. To check that the driver has priority over the passenger, two different commands have to be given in parallel. The developed language is inspired from the work of Dhaussy [Dhaussy and Roger, 2011] and has been applied to the verification of critical systems, e.g. in military aviation.

A model in the environment description language has a *top activity*, which can be of type *parallel*, *sequence* or *alternative*. A top activity may contain other activities, in an arbitrarily deep hierarchy. As the names indicate, activities that are executed in parallel will occur simultaneously, sequential activities will occur one after the other in a predefined fashion, and, from a set of alternative activities, one is chosen non-deterministically to be executed. The basic activity is the *communication sequence* which is a sequence of events being exchanged with another system. Events may be of type *output*, meaning they are sent towards the outside system, or of type *input*, meaning they are expected from the outside system. An event has an amount of time in seconds associated with it, which is the amount of time the event will take to complete.

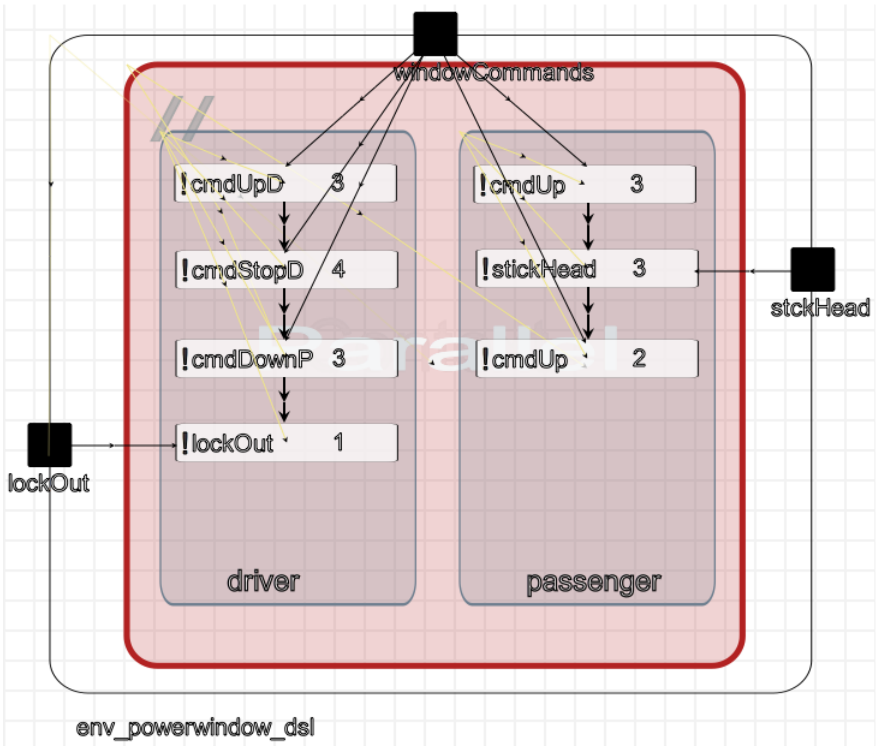


Figure 4.7: Example Model in the Environment Description Language

In figure 4.7 we present a model described in the environment description language. The model presents an example of an interesting interaction with the power window hardware described in figure 4.5 where the driver and the passenger both issue a sequence of commands in parallel. The model implements one of the alternative scenario's described in use case 5. The parallel block represented by the red square includes two command sequences that have a sequence of output events. Each graphical representation of an output event includes on the right of the box the amount of time the event will take until completion. In particular, the *stickhead* command means the passenger has blocked the window rolling up by blocking it with some object. Note that an environment could contain other actors, even other systems.

Control

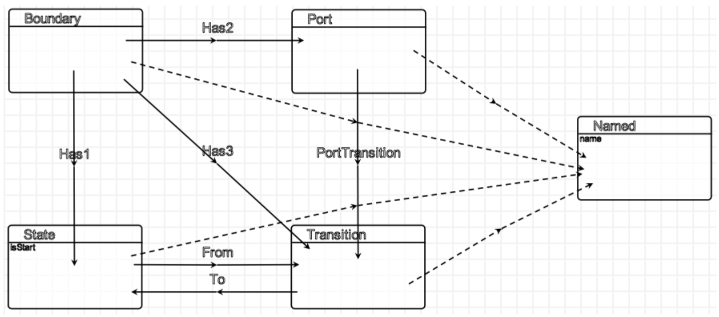


Figure 4.8: Control Description Language Metamodel

The control is modelled in a statechart like formalism. The hybrid statechart has input and output ports. Transitions can be defined over the incoming values of a an input port. It contains event detection when crossing a value from below (<!) or above (>!). There is also a notion of time by using the $after()$ directive. The actions are defined over the output ports of the hybrid statechart formalism.

In figure 4.9 a model for controlling a power window with object detection can be observed. The control logic states that the window can be either in neutral mode (with the electric motor stopped), moving up or moving down. The control logic will change state if a new control command will be issued by one of the buttons attached to the window. The model includes dealing with stopping window action if an object is detected blocking window lifting. In this case the window control logic goes into an emergency state and then into the neutral state which stops window movement.

In figures 4.10 and 4.11 we present additional models in the control description language required to describe the control aspects of a power window without obstacle detection, as well as of a lockout button.

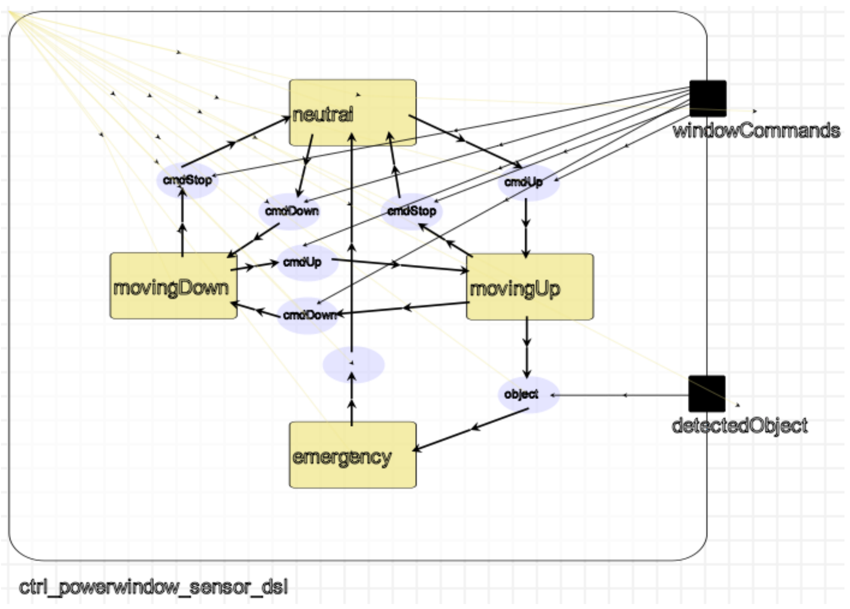


Figure 4.9: Window with Obstacle Detection Control Model

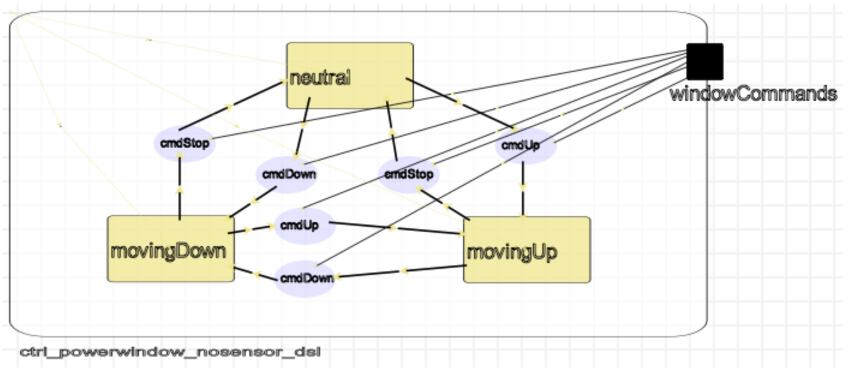


Figure 4.10: Window with No Obstacle Detection Control Model

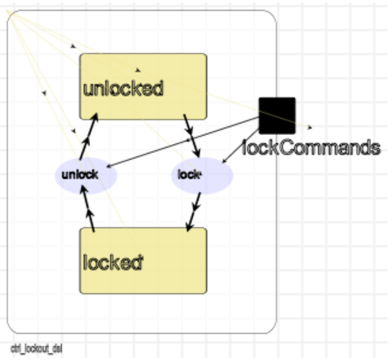


Figure 4.11: LockOut Control Model

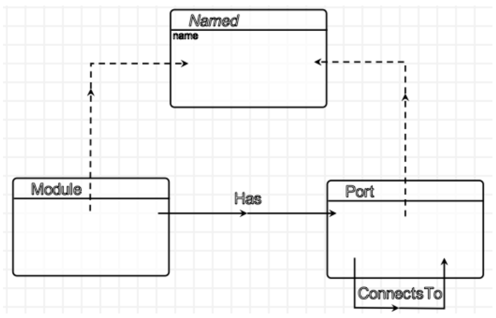


Figure 4.12: Network Description Language Metamodel

Network Description Language

It becomes now necessary to compose the *plant*, *environment* and *control* models presented in figures 4.5, 4.7, 4.9, 4.10 and 4.11. In figure 4.12 we present the metamodel of a Network Description language used for this task. The idea behind this language is very simple: components can be connected to other components, where a component is seen as a black box and connections are made by linking components' ports. Each of the formalisms introduced above has the notion of *ports*, represented in concrete syntax by the black squares over the boundaries of each model.

In figure 4.13 the network model for our power window example is presented. Note that the internal detail of each of the models is abstracted and only the connections between the ports are visible.

4.2.3 Verification Activities

We now describe the generation of a Petri Net from the DSL models presented in section 4.2.2. The Petri Net formalism is an automaton like formalism involving places and transitions (resembling the UML states and transitions) but also with the capability of describing concurrency. Tokens distributed across places allow describing that certain resources are distributed in the system and the non-deterministic firing of transitions simulates the consumption of resources in places and production of new resources in other places.

The goal of generating a Petri net from the DSL models is to build an artefact that can be used for the exhaustive exploration of functional scenarios of operating the power window. Many model checking tools exist that will take as input a Petri nets model and a property and will decide if the property is true or not in the model. This Petri net is used to verify that properties hold in our power window definitions. For example, it is important to show that if the driver is commanding the passenger window to go up or down, then the passenger cannot operate his/her window; or, that when an obstacle blocks a sensor-equipped power window when going up, the window will stop and go

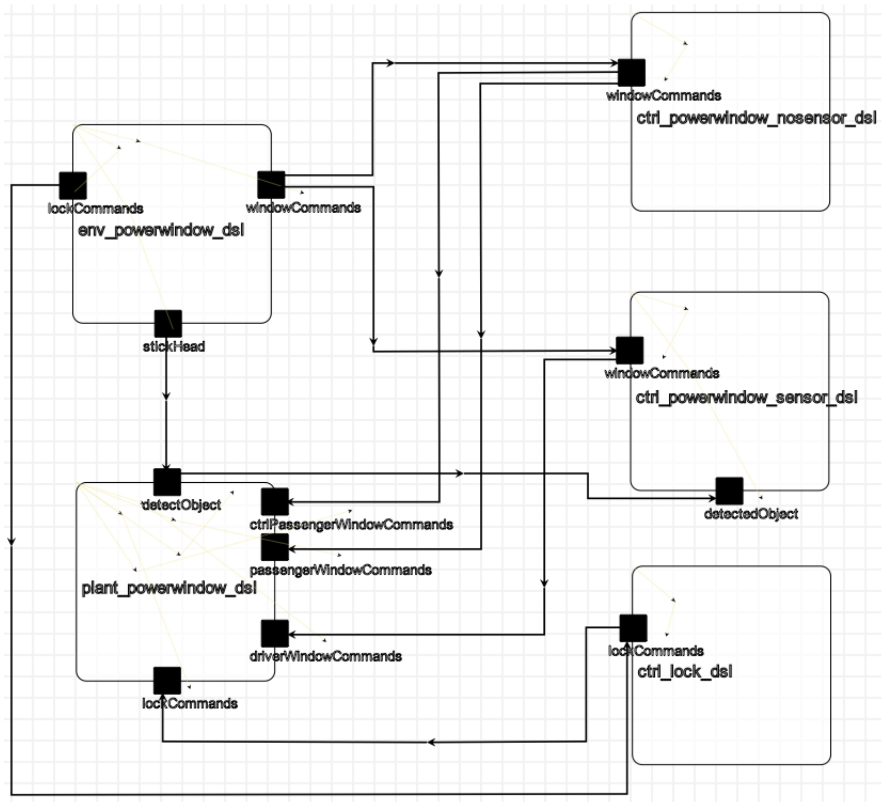


Figure 4.13: Network Model for the Powerwindow

down. In this section we will describe how the Petri net representing the behaviour of the composition of the DSL models presented in section 4.2.2 is obtained by using a set of transformations.

In what follows, we start by presenting the transformation of each of the DSL models presented in section 4.2.2 into Petri nets. We will also transform the network model into a specialized network model that will be used in the composition of the Petri nets obtained from the DSL models. Notice that Petri nets are a discrete formalism, as opposed to the causal block diagram formalism presented in section 4.2.4 which is a formalism capable of representing continuous behaviour. The discrete nature of Petri nets comes from the fact that every state of the execution of a Petri net model can be identified as a set of tokens occurring in a set of places.The choice for the Petri net formalism stems from the inherent concurrency and analysis capabilities of the Petri net formalism. The result of this is that not only the controller but also the plant and environment of the system can be easily modelled.

Transformation of the Environment Model into Petri Nets

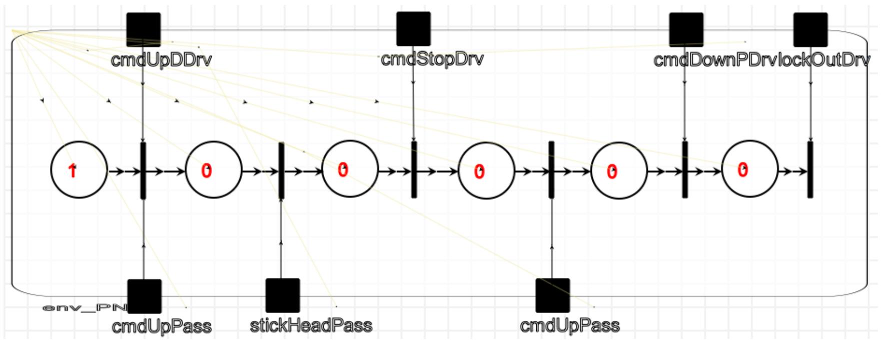


Figure 4.14: Transformed Environment in Encapsulated Petri Nets

In figure 4.14 we depict the result from the transformation of the environment DSL model presented in figure 4.7 into modular Petri nets. The transformation recursively treats all the *parallel*, *sequence*, *alternative* and *communication sequence* activities in the environment model and builds the necessary modular Petri net for them. In the modular Petri net in figure 4.14 we can observe that the two communication sequences from the environment model in figure 4.7 have been merged into one single sequence of transitions. This is so because the two activities occur during the same timeline, where each relevant discrete moment is represented by a transition. The transformation from the environment DSL into a Petri net will compute how many discrete moments are required and will generate a corresponding amount of Petri net transitions. The ordering of the events depends on the amount of time each of them requires to be completed. The Petri net transitions are synchronized with ports of the module in order to output the events to other components. Some of the events can happen simultaneously (because their time distance from the start of the activity is the same), which means the same Petri net transition connects to two ports.

The translations of the *alternative* and *sequential* blocks are less complex than the ones of the *parallel* block because the timelines for each of those blocks do not need to be composed, although they need to be assembled together such that one or the other is chosen (alternative), or they are sequentially executed (sequence). A final observation on the transformation of environment models into modular Petri nets is the fact that *input* events coming from the outside of the components are treated by synchronizing the Petri net's transitions with them. This is done in such a way that a transition of the component can only fire when receiving an input from an external component.

Transformation of the Plant Model into Petri Nets

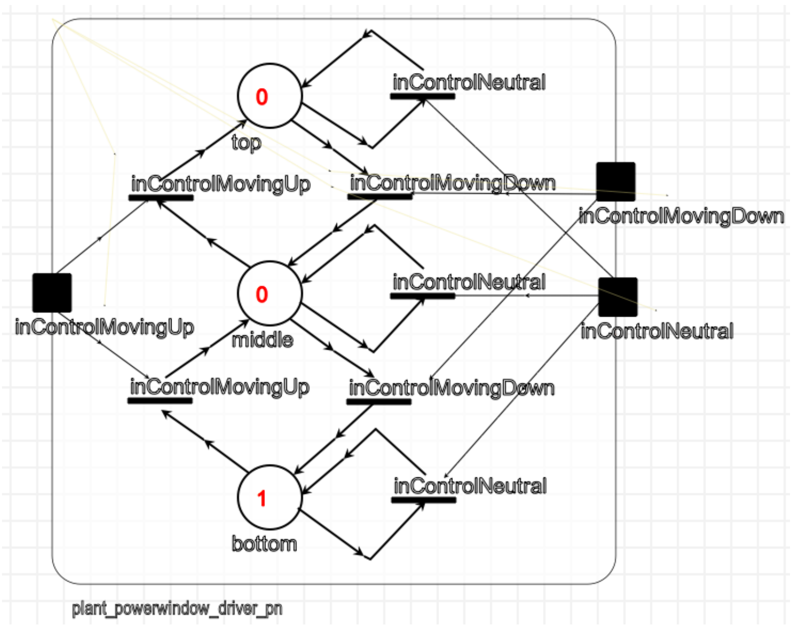


Figure 4.15: Transformed Driver Window Plant Model

Because of the requirements of the power window system, the *driver* window does not include an obstacle detection sensor, while the *passenger* window does include an infrared sensor. Two modular Petri nets are generated from the Plant DSL model: in figure 4.15 a discrete behaviour of a power window without an obstacle detecting sensor can be observed. During operation the window can either be at the *bottom* of the frame (meaning the window is completely open), somewhere in the *middle* of the frame (meaning the window is partially open), or at the top of the frame (meaning the window is closed). The behaviour of the plant is dependent on the behaviour of the controller which can give a command to go *up*, *down* or the *neutral* command. In figure 4.16 the behaviour of a power

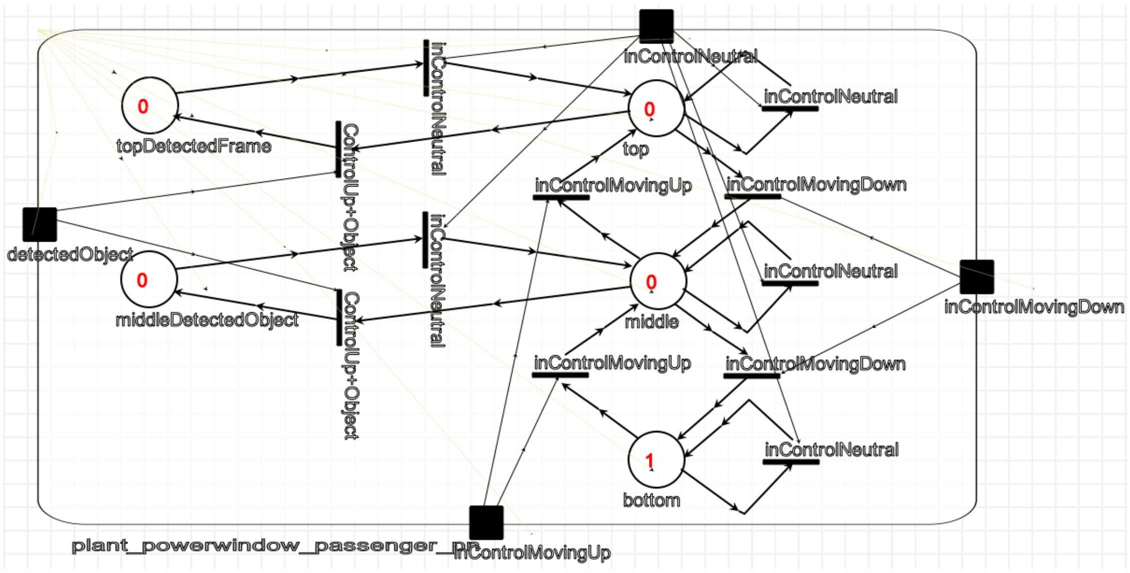


Figure 4.16: Transformed Passenger Window Plant Model

window plant with an infrared sensor is shown. The basic states are the same as in the modular Petri net power window without the infrared sensor, but additional states where an obstacle blocking window movement up is detected have been added.

The modular Petri nets in both figures 4.15 and 4.16 include the necessary ports for communication with the controllers. Notice that we could have chosen to represent the physical behaviour of the power windows differently if a more precise behavioural representation would be required. For example, we could have a finer representation of the position of a window in its physical frame or a finer representation of obstacle detection by adding more intermediate places to the Petri nets presented in figures 4.15 and 4.16. In fact, we have chosen as example the infrared obstacle sensor which outputs a binary signal (obstacle detected or no obstacle detected). In order to generate a modular Petri net representation of a power window with a *force detecting* obstacle sensor, we would need a representation of force detection which would be finer than the one required for the infrared sensor. This is so because power consumption values in the power window electrical motor need to be monitored such that the window is not stopped unless sufficient resistance force is applied to the window.

Transformation of the Control Models into Petri Nets

Figures 4.17, 4.18 and 4.19 represent the modular Petri Net behaviour of three controllers: the controller for a power window switch without obstacle control, the controller for a power window with obstacle control, and the controller for the lockout switch. Unsurprisingly, these are the switches that can used when building power window plants.

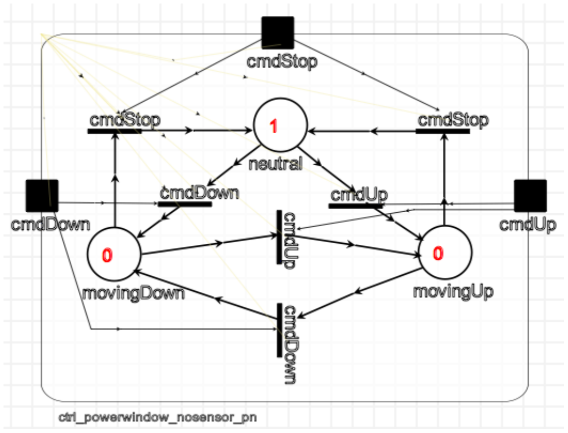


Figure 4.17: Transformed Window without Obstacle Detection Control Model

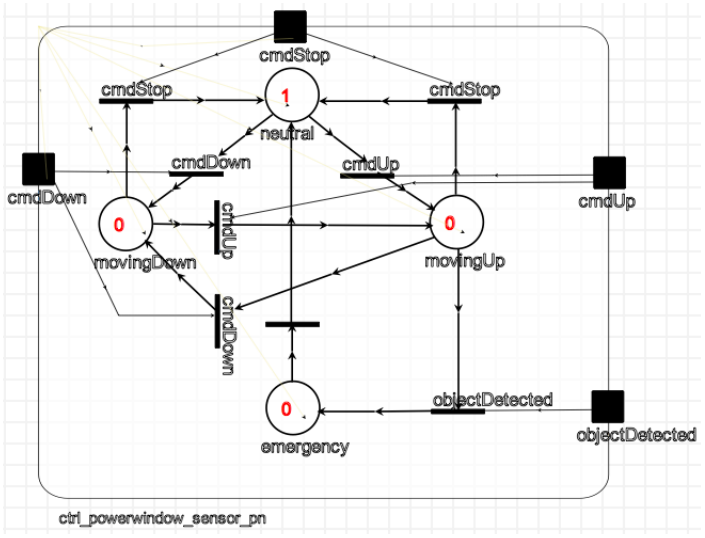


Figure 4.18: Transformed Window with Obstacle Detection Control Model

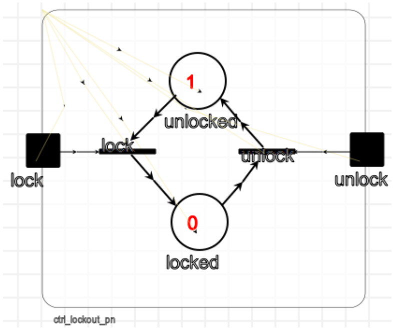


Figure 4.19: Transformed LockOut Control Model

The semantics of the simplified statecharts we have used to define the controllers, can be easily simulated using Petri nets, the resulting modular Petri nets presented in figures 4.17, 4.18 and 4.19 are structurally very similar to their counterparts in figures 4.10, 4.9 and 4.11 respectively – statechart *states* are transformed into modular Petri net *places* and statechart *transitions* are transformed into Petri net *transitions*. The modular Petri net's transitions are synchronized with module's *ports* having the same name.

Transformation of the Network Model

The network model we have presented in figure 4.12 connects the *environment*, *control* and *plant* domain specific components we have defined in section 4.2.2. The *network* at the domain specific level is transformed into a specific network that connects the modular Petri net components described in the above. The result of this transformation is presented in figure 4.20.

At the domain specific level many details of the behaviour of the modular components and how they are linked together is abstracted. This is so because the domain specific languages for the power window are built to allow, in principle, the automotive engineer to be as expressive as possible using minimalistic domain specific constructs. For example, because the complete set of window movement ports is abstracted at the DSL level, they need to be expanded both at the component and at the *network* level. Also, in the domain specific network it is abstracted how the controllers send signals to each of the power windows defined in the plants. As can be observed in figure 4.20 this issue is tackled in our example by connecting the appropriate kind of controllers to the modular Petri net generated plants for the two power windows present in the plant in figure 4.5.

Composition Transformation

The composition transformation builds the connections between the modular Petri nets, by using the network model in figure 4.20. For reference, we present in figure 4.21 the set of Petri nets that are to be composed, without their communication ports. In figure 4.22 we present the composed version of figure 4.20. Note that, not to overwhelm the reader, in figure 4.20 we provide only some of the composition links between the individual nets.

Verification

Finally, from the Petri Net model a coverability graph is constructed. The safety properties of the power window case study are checked by using a CTL (Computational Tree Logic) formula. For each of the properties that need to be verified a CTL formula is needed. More information can be found in [Mosterman and Vangheluwe, 2004]. An example formula, that checks that the window can never go up when an object is detected is shown below.

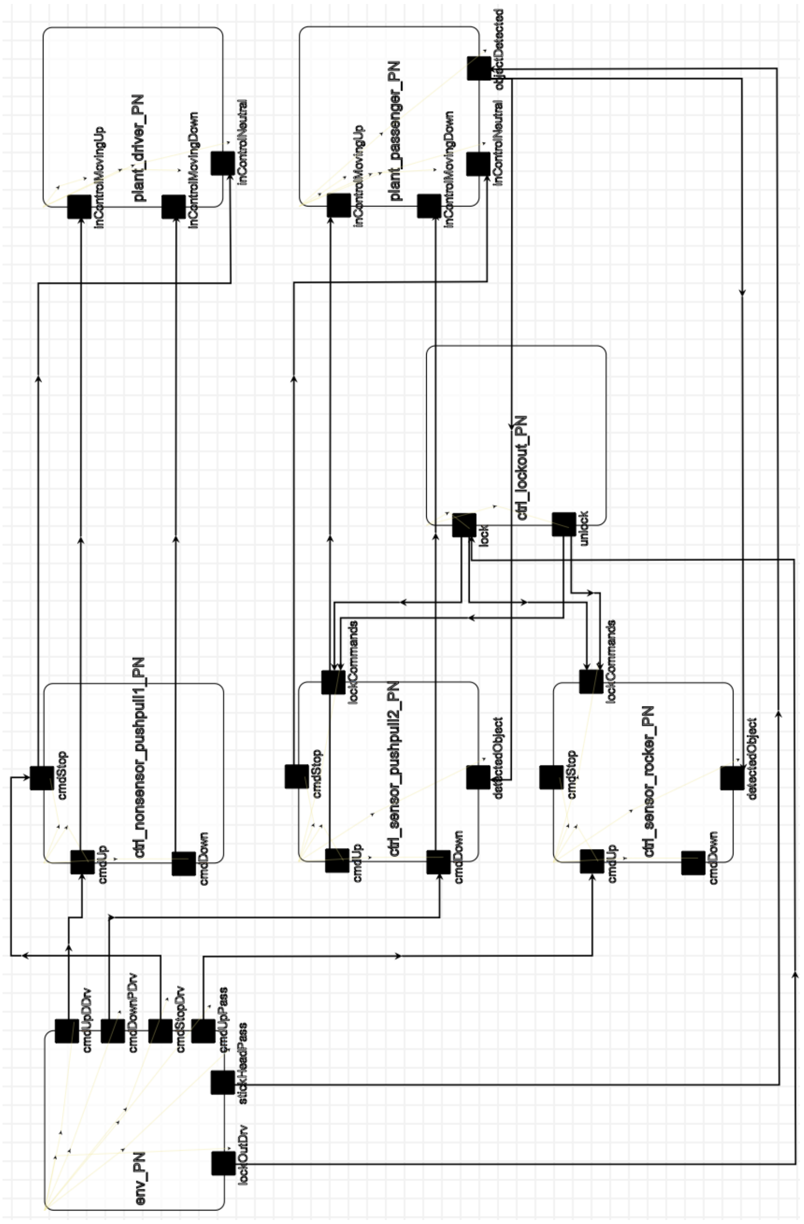


Figure 4.20: Transformed Network Model

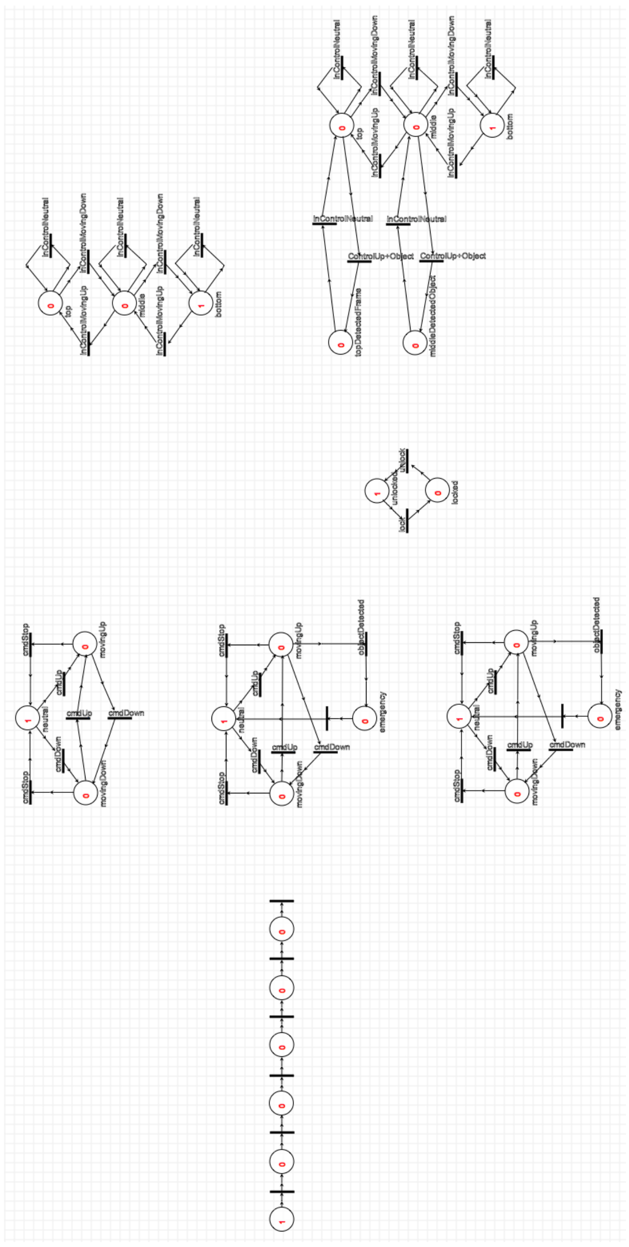


Figure 4.21: Uncomposed Petri Net Model of the Powerwindow Software

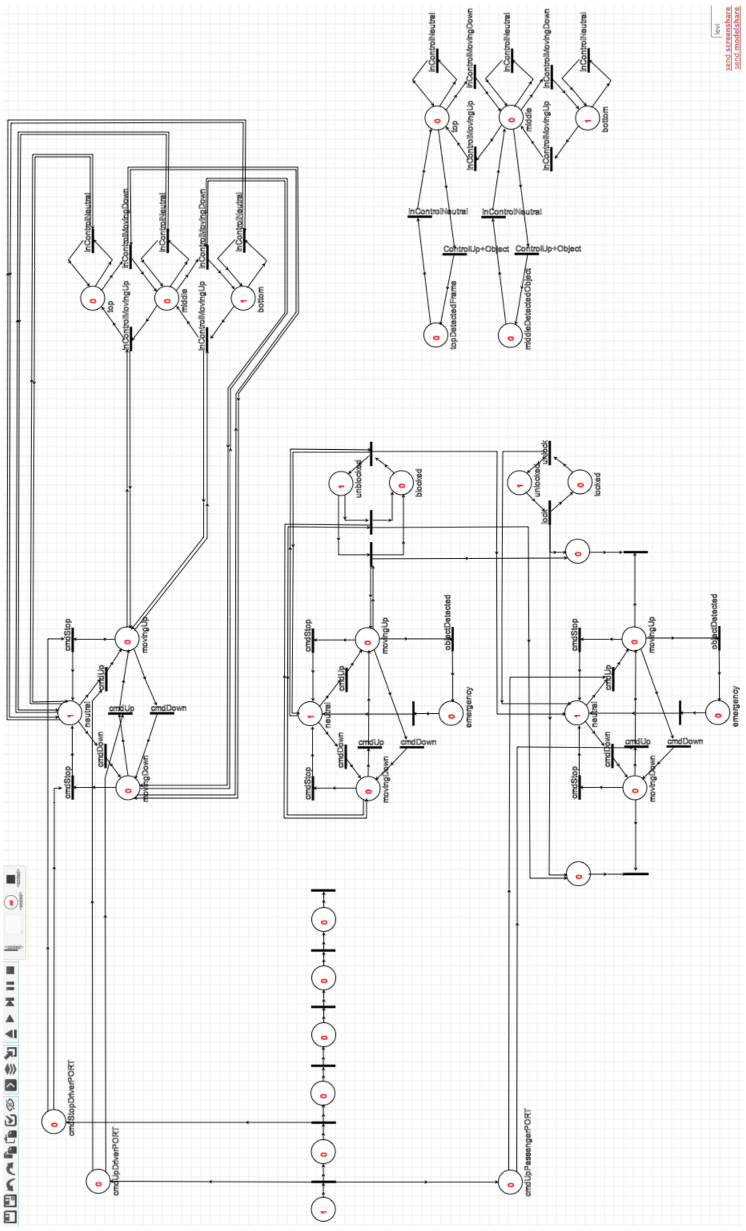


Figure 4.22: Transformed Composed Petri Net Model of the Powerwindow Software

$AG\neg(movingUp \wedge MiddleAndDetectedObject)$

4.2.4 Simulation Activities

In this section, the generation of a hybrid simulation model from the DSL models in section 4.2.2 is described. The hybrid simulation is composed out of the Causal Block Diagram formalism and the statechart formalism. On the one hand, the physical components of the system and the environment are described using this continuous-time formalism. The CBD formalism is chosen because ODEs can easily be represented using this formalism. The formalism allows for a numerical simulation of the model where the traces can be compared to the requirements.

On the other hand, the controller is described in a discrete-event based formalism hence the name hybrid simulation model. Different approaches are possible for the composition and execution of hybrid simulation models: (a) the creation of a super-formalism, (b) transformation to a common formalism and, (c) co-simulation.

The goal of generating a hybrid simulation model is to check certain functional properties of the interaction of the control software with the physical plant. For example that the window is lowered 12.5 cm on detection of an obstacle or to evaluate that the force on the object does not exceed a certain threshold. In this section we describe the generation of a hybrid model of the power window using a set of transformations. In chapter 5, we detail the execution of the hybrid simulation.

Transformation of the Environment Model into a Causal Block Diagram

In figure 4.23 we show the result of the transformation of the environment DSL model of figure 4.7 into a causal block diagram. The model is encapsulated in a *child* block that allows hierarchical modelling so it can be used as a whole when composing the full hybrid simulation model. The most important block involved in this model is a source block that generates a sequence of output values. This sequence is based on a vector containing tuples of time and output value. The block computes its output values based on a piecewise constant reconstruction of the signal based on the input samples.

The transformation creates a ”sequence block” for each of the unique states in the environment model. The vector of tuples inside a created block is based on the time defined in each state of the environmental model. As a parameter, the transformation needs the translation between the event name (in the DSL) and the corresponding value that can be used within the Causal Block Diagram. It also needs a default value when the event is not applied.

The generated model is encapsulated in a child-block. The child block encapsulates a set of blocks, replacing them by a single block. It is used to hierarchically model the system and reduce the visual complexity. A child block can have multiple input and output ports.

These ports are linked to the in- and output-blocks within the submodel. The model in figure 4.23 outputs all the signals to the parent model using the output ports.

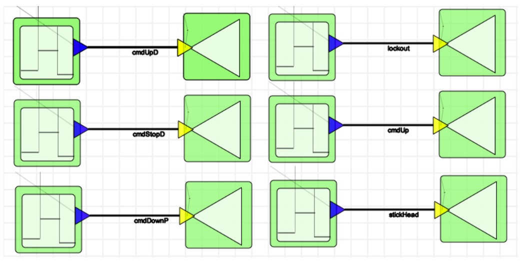


Figure 4.23: Causal Block Diagram of the environment model

Transformation of the Plant Model into a Causal Block Diagram

The plant model of the power window can be transformed into a continuous model of the behaviour of the up and down movement of the window. Like in section 4.2.3 two models are generated from the DSL model. Both transformations use information like window height, motor gain and window friction from the DSL model.

Figure 4.24 represents the model of the window without any obstacle detection. The model is a simple model of the up- and downward-movement of the window. It outputs the position of the window so it can be observed during simulation. As with the environment model, it is encapsulated in a child block. If the window is not on top or bottom, the input and output commands for the up- and down-command are added together to get a single up or down command for the motor. This is multiplied with a motor gain while the friction is subtracted via a feedback loop. After integration of the acceleration we obtain the window speed from the input. The window speed is integrated to get the window position.

The window with an infrared obstacle detection is shown in figure 4.25. The logic of the model is similar to Figure 4.24. The biggest difference is the output of the detected object when both an obstacle is present and the window is moving up.

Encapsulation of the Control Models

To allow the statechart to be used in a Causal Block Diagram, it needs to be encapsulated. The encapsulation of the statechart requires three blocks as seen in figure 4.26. The first is the State Event Locator (SEL). This block translates incoming signals into events so they can be fired by the statechart. The second block contains the statechart defined in section 4.10. Finally actions have to be translated back to the continuous domain by the transducer block. Both the SEL and transducer block contain a lookup table that are generated from the hybrid statechart formalism.

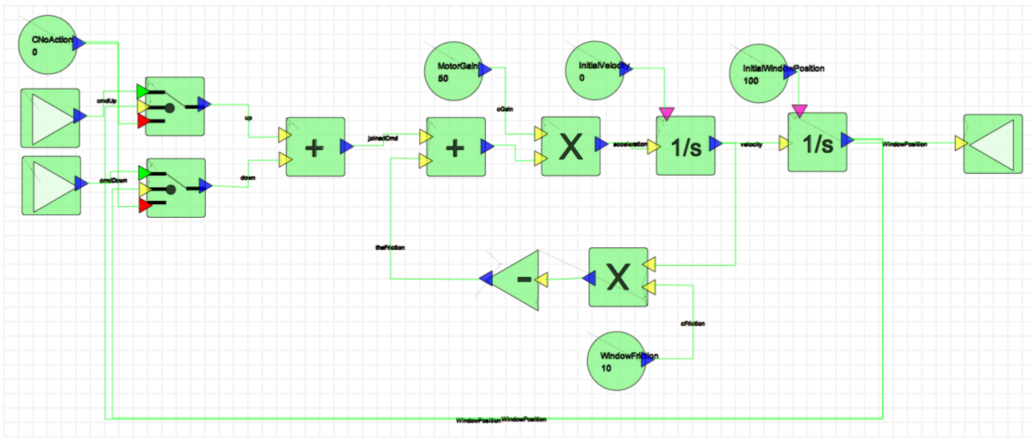


Figure 4.24: Model of power window plant without an obstacle sensor

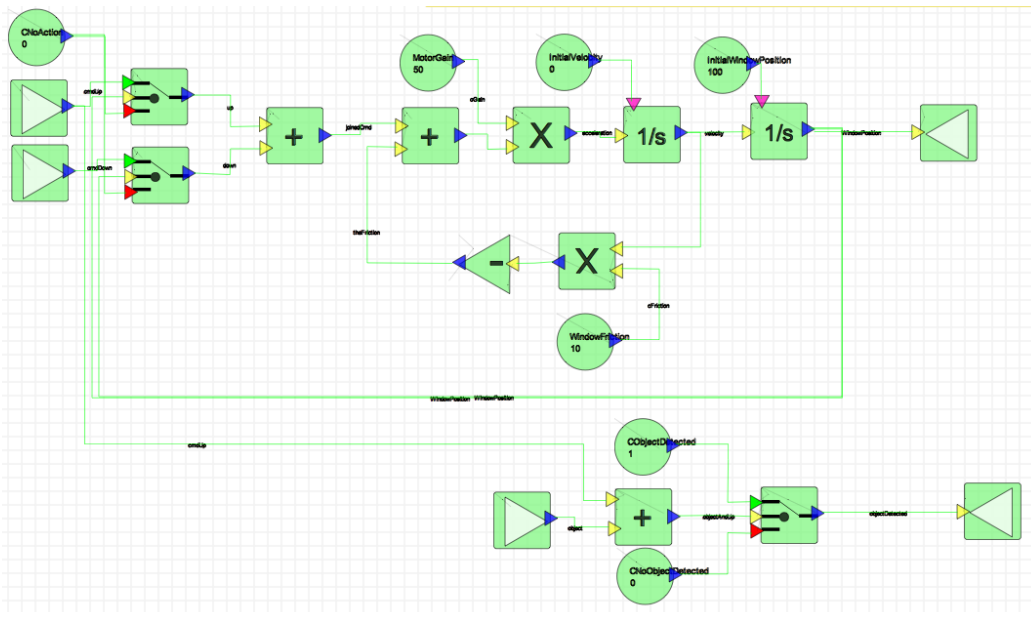


Figure 4.25: Model of power window plant with an object detection

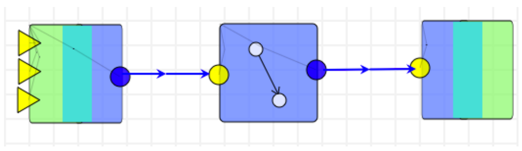


Figure 4.26: Encapsulated control

Transformation of the Network Model

The transformation of the network model is similar to the transformation defined in section 4.2.3.

Composition Transformation

The composition transformation composes the different models obtained above using the network model. The name of the ports of the different child blocks match the name of the ports in the network model and can thus be matched easily.

The composed model can be seen in figure 4.27. Since the formalism needs input values on all of the defined input ports, the unconnected ports need to be connected to a default value. In this case the non issued commands for the controller parts are given a no action value.

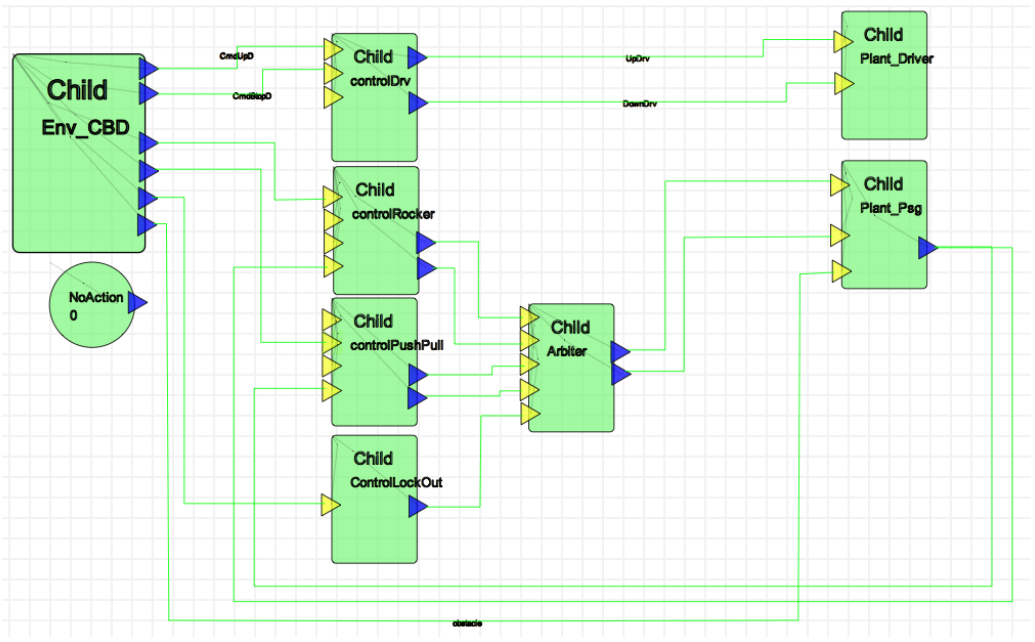


Figure 4.27: Composition of the full hybrid simulation model

Verification of Behaviour

To check if our design conforms to the requirements the traces of the simulations are compared to the requirements. This is done for each of the environment models constructed to check one or more requirements of the system. While we checked this using a manual approach, it is possible to do this in an automatic way. For example, Mosterman and

Prabhu [Prabhu and Mosterman, 2004] use a set of dedicated CBD blocks for this purpose. This results in the automatic verification of the models where the requirements are also explicitly modelled.

4.2.5 Deployment Activities

This section is devoted to the exploration of the deployment space. During the process of deployment onto hardware a plethora of configuration choices have to be made in the middleware. These choices range from the mapping of software components to a hardware platform. But also lower level decisions like mapping of software functions onto tasks and assigning these tasks a priority. On the network side we have similar choices like the mapping of signals to messages and low level parameters that affect the sending and receiving of messages on the bus.

As a deployment platform we will use the AUTOSAR platform. Because of the size of the AUTOSAR meta-model, we defined a simpler meta-model based on the AUTOSAR concepts. Some parts of AUTOSAR are omitted, like processor, pin and IO configuration. The deployment model focusses on the software, hardware and system architecture. At the lowest level, it encompasses the configuration of the real-time operating system and the CAN communication stack.

The deployment space exploration consists out of four independent parts: (a) converting the statecharts to an AUTOSAR software component diagram, (b) generation of a calibration infrastructure, (c) the deployment space exploration and (d) code generation activities. Because of clarity reasons, we focus only on the passenger side of the system. The other side should be added as well for a full deployment.

Converting statecharts to an AUTOSAR software component model

From the statechart of the control part, a software component is generated containing the logic in a single runnable. Though for every incoming port to SEL block and outgoing port in the transducer block, a sensor-actuator block is created. These sensor-actuator blocks can access the hardware of the platform (for example an Analog-Digital converter or a general purpose input-output pin). The logic in these blocks contains a function to translate the electrical signal values coming from the sensor to an engineering value. Or on the actuator side to control motors, etc.

Other parameters must be added as well. Examples include the events triggering the components (for example a timing event) and datatypes exchanged between the components have to be set. Because of this, we generated the software component model using a manual transformation.

Figure 4.28 shows the generated AUTOSAR component diagram of the driver side. The passenger part is omitted from the model. All the software components contain a single runnable.

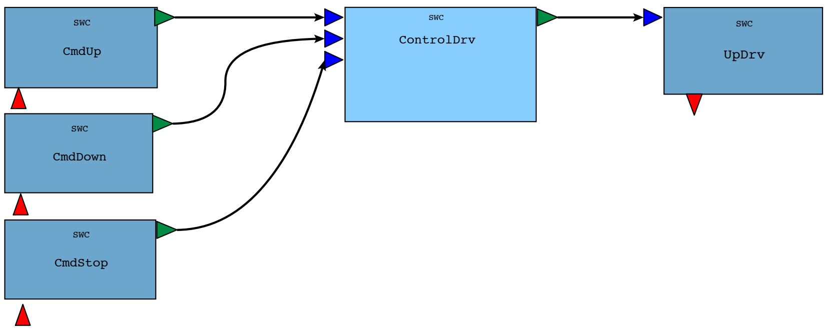


Figure 4.28: AUTOSAR software component model

Calibration infrastructure generation

The exploration of the deployment space relies heavily on simulation and analysis. A crucial step in this process is the calibration of the models involved. Parameters to be estimated are for example network throughput, memory consumption and execution time of the different software components. For calibration of performance models of software intensive systems, that have a tight combination of the physical and computational components of the system, the input values of the software input components originate in the environment of the system and in the feedback loops that exist between the computational and physical components. As a consequence, a trace-driven approach to supply the input components with input signals is not feasible because of the effects of the software on the physical components and vice versa. Using the models developed in the previous section, we can generate a calibration infrastructure to measure the execution times, memory consumption, energy consumption, etc.

In our example we will use the target hardware to run the computational components of the system while using a host computer to execute the simulation models of the environment and plant. Signals generated by the environment and plant are transmitted by a bus, for example a serial connection, to the target board. The target board executes the computational components while measuring the calibration parameters. These are transmitted back to the host computer together with the output values that are used by the plant model.

Instrumented code can be synthesised from the AUTOSAR software components. These instrumentations measure certain properties of the AUTOSAR basic block. The instrumentation uses a call to a tiny middleware to read out sensors (like a timer, instruction counter, etc). Also a small run-time environment is generated that will create the buffers for the communication signals and trigger the software functions at the right time. Also the basic blocks are executed in an atomic way so no effects from interrupts or preemptions can distort the measurement. In Listing 4.1 a small example of an instrumentation is shown.

Code Listing 4.1: Example instrumented code of a sensor-actuator component

```
{

    startMeasure1(); /*Instrumented code*/ status = CmdUp_RunRead();
    stopMeasure1(); /*Instrumented */ TxDistribution(__ID_CmdUpSensor_RunRead,
    getInterval1()); /*Instrumented*/
}
```

From the environment and plant models a simulator is generated. This is very similar to the generation of the hybrid simulation model in section 4.2.4.

Finally from the network model and hardware model the infrastructure is generated that captures, transmits and receives values on the host computer and target board. A template middleware is needed for each used target board involved in the hardware model.

When the measurements are collected from the calibration infrastructure, the results are annotated in a performance model. This performance model combines the type of processor with the software functions within a software component.

Execution Time (μs)	Distribution
20.000	7500
20.875	7499
21.375	1

Table 4.2: Example of a performance annotation between the ControlDrv and the MPC5567 hardware type

An example of the performance annotations created by the calibration infrastructure is shown in Table 4.2. The calibration step is further detailed in chapter 7.

Deployment space exploration

In our example we will use an automatic deployment space exploration technique. It uses a platform-based approach by defining three different abstraction levels. At every level a transformation is defined to evaluate the real-time properties of the configuration.

At the first abstraction level the architecture is explored. The transformation maps the software components to the defined hardware components in the hardware model. It needs, as an input, a hardware model of the different components. Note that in more complex explorations this hardware model can also change. Changes include the number of hardware components, the type of processor, the type and number of communication buses, etc. Figure 4.29 shows an example hardware model.

Since an AUTOSAR software component is atomic, it needs to be mapped to a single hardware component independent of the number of software functions in the software component. Sensor-actuator components are special since they need to be in the vicinity of their respective sensor or actuator. The system architect pre-maps these components. Normal software components can be mapped to any hardware platform. Figure 4.30

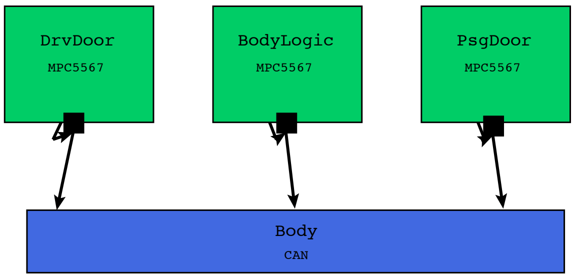


Figure 4.29: An example hardware model

shows a possible configuration of the mapping between the hardware model shown in figure 4.29 and the software component model in figure 4.28. Signals that are communicated between software components mapped to a different hardware component result in a signal transmitted on the bus.

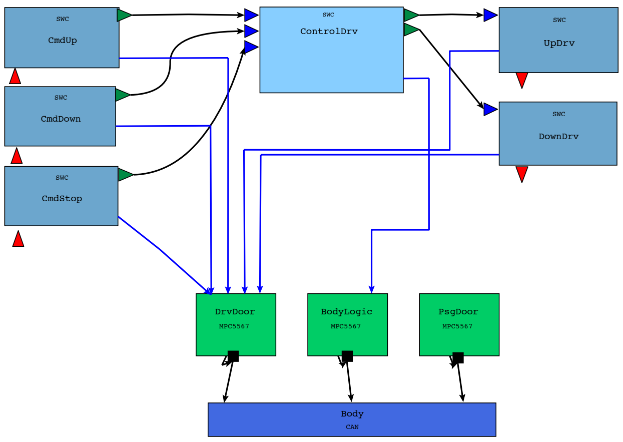
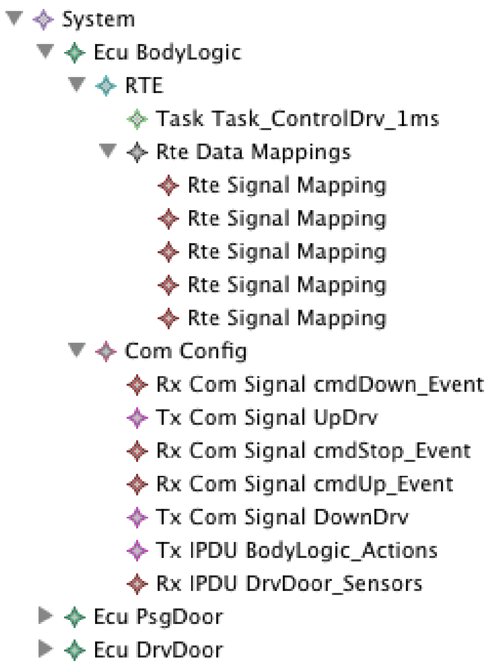


Figure 4.30: The software components mapped to the hardware model

The configuration can be evaluated using a bin packing check. As input it uses the performance model and architecture model. The bin packing check is a simple algebraic equation to evaluate the usage of a hardware component. The algorithm calculates the sum of each execution time of the software function mapped to the ECU divided by the period of the software function (functions with data-sent or data-receive events are assigned the period of their respective parent). The same is done for the signals on the bus (without the overhead caused by frame headers and trailers). An example of the bin packing check for the BodyLogic component with the mapping of figure 4.30 can be seen below:

$$\begin{aligned} &\ldots \\ &\textit{Usage of Processor BodyLogic} = (((21.375/100))/1) = 0.21375 \\ &\ldots \end{aligned}$$

The second step of the deployment exploration process is the mapping of the software functions to tasks on the operating system and assigning them a priority (the AUTOSAR operating system uses a fixed priority preemptive scheduler). Also depending on the bus type, the signals are mapped to messages on the bus and assigned a priority (in case of an event-triggered bus) or a slot (in case of time-triggered bus). Properties of signals and messages are set. Figure 4.31 shows a partial deployment model build using the Eclipse modelling framework. The software functions are mapped to tasks. All properties of the task are set. The signals are also mapped to messages and assigned their properties.



```
▼ System
    ▼ Ecu BodyLogic
        ▼ RTE
            Task Task_ControlDrv_1ms
            ▼ Rte Data Mappings
                Rte Signal Mapping
                Rte Signal Mapping
                Rte Signal Mapping
                Rte Signal Mapping
                Rte Signal Mapping
        ▼ Com Config
            Rx Com Signal cmdDown_Event
            Tx Com Signal UpDrv
            Rx Com Signal cmdStop_Event
            Rx Com Signal cmdUp_Event
            Tx Com Signal DownDrv
            Tx IPDU BodyLogic_Actions
            Rx IPDU DrvDoor_Sensors
    ▶ Ecu PsgDoor
    ▶ Ecu DrvDoor
```

Figure 4.31: An example of a partial deployment. The example is made with a reduced version of the AUTOSAR meta-model in e-core.

Configurations at this level of abstraction can be checked using schedulabilty analysis. Schedulability analysis [Tindell and Burns, 1992] is a well known technique for estimating whether a task will make the imposed deadlines.

Finally the low-level deployment starts. This is done by defining hardware buffers for the reception and transmission of messages. Since the hardware platform only has a limited amount of buffers in the communication controller the mapping is not a one-to-one mapping. The drivers and interfaces of the communication stack are configured and software buffers are defined if needed. Some hardware-specific options are also configured. Figure 4.32 shows a full deployment configuration.

The last method of evaluation is a low-level deployment simulation. In our example we use a DEVS deployment simulation model. In Listing 4.2 a code snippet from an atomic

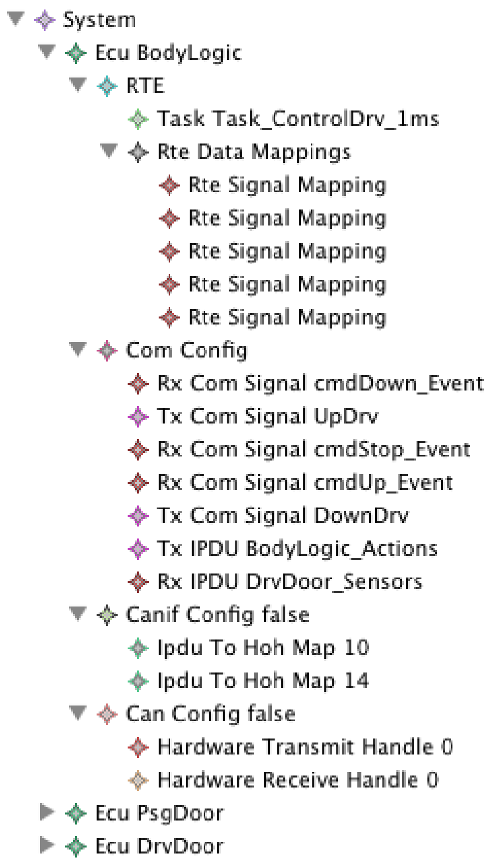

System
Ecu BodyLogic
RTE
Task Task_ControlDrv_1ms
Rte Data Mappings
Rte Signal Mapping
Rte Signal Mapping
Rte Signal Mapping
Rte Signal Mapping
Rte Signal Mapping
Com Config
Rx Com Signal cmdDown_Event
Tx Com Signal UpDrv
Rx Com Signal cmdStop_Event
Rx Com Signal cmdUp_Event
Tx Com Signal DownDrv
Tx IPDU BodyLogic_Actions
Rx IPDU DrvDoor_Sensors
Canif Config false
Ipdu To Hoh Map 10
Ipdu To Hoh Map 14
Can Config false
Hardware Transmit Handle 0
Hardware Receive Handle 0
Ecu PsgDoor
Ecu DrvDoor

Figure 4.32: An example of full deployment. The example is made with an reduced version of the AUTOSAR meta-model in e-core.

DEVS model can be seen. The coupled DEVS of part of the deployment is shown in figure 4.33. The appropriateness of DEVS and the design of the low-level simulation model can be found in chapter 6.

Code Listing 4.2: Example of an atomic DEVS model

```
class CanBusDEVS(AtomicDEVS):
    def __init__(self, name, speed):
        AtomicDEVS.__init__(self, name)
        self.state = CanBus(speed)
        self.INFRAMES = self.addInPort("CANFramesIn")
        self.NOTIFY = self.addOutPort("CANBusIdle")
        self.OUTFRAMES = self.addOutPort("CANOutFrames")
    def intTransition(self):
        self.state.onInternal()
        return self.state
    def extTransition(self):
        inFrame = self.peek(self.INFRAMES)
        self.state.onExternal(inFrame, self.elapsed)
        return self.state
    def timeAdvance(self):
        return self.state.getTimeLeft()
    def outputFnc(self):
        out = self.state.getOutput()
        if out is not None:
            self.poke(self.OUTFRAMES, out)
            self.poke(self.NOTIFY, out)
```

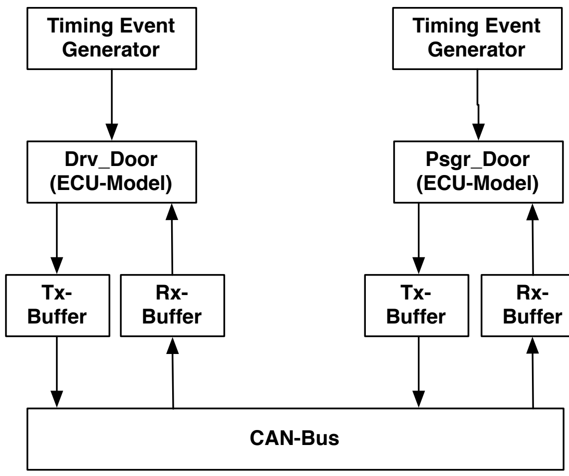


Figure 4.33: Example of a coupled DEVS model

The techniques and rationale involved in exploring the design space using a transformation based approach is detailed in chapter 8.

Code generation

The final step is the generation of the code that runs on the target platforms. This includes the generation of the middleware (adapted for the application), the application source code and the run-time environment to glue the middleware and application together. Details of this transformations can be found in the AUTOSAR specifications. A commercial tool, like EB Tresos Studio, is able to generate the code for the two ECUs.

4.3 Discussion

The case study is inherently complex in nature. Though some aspects are still not considered:

— Feature diagrams, commonly used for product line engineering are not used in this case-study. The software intensive systems today however, have a lot of variants. Feature diagrams can be used at the top of the FTG+PM to help create product lines. An overview of variability techniques can be found in [Czarnecki *et al.*, 2012].

— At the end of the case study we did not include a model based testing step. Hardware-in-the-loop approaches are very common for testing software intensive systems. In this environment, the plant and environment is simulated while the actual controller with the deployed software is executed. Because of the distributed nature of software intensive systems, restbus simulation can be used to simulate the rest of the controllers [Köhl and Jegminat, 2005].

— Control theory uses a mathematical foundation to produce a controller from a plant model, for example a PID (Proportional-Integral-Derivative) controller. Control engineers usually represent this using CBDs. These controllers are usually encapsulated within the states of the statechart (for example an adaptive cruise control with different models where a PID controller is only used in one or two states). Thus a hybrid simulation is also needed at this level. A technique for simulation of event-scheduling formalisms with an encapsulated ODE is discussed in [Lacoste-Julien *et al.*, 2004].

Besides these aspects, the proposed FTG+PM can be used as a guide for the design, verification and deployment of other software intensive systems since they all have similar requirements. The FTG+PM however can be adapted to the needs of the designers. Adaptations could include the use of other formalisms, for example for the physical part, Modelica [Mattsson *et al.*, 1997] and Bond Graphs [Paynter, 1961] are both well known general purpose physics modelling formalisms that can be used in between the domain specific design and the causal block diagram formalism. They could even replace the CBD formalism, though this has some side effects in the FTG+PM.

4.4 Related Work

In [Prabhu and Mosterman, 2004], Prabhu and Mosterman implement the power window exemplar using the tools provided by MathWorks. Starting from requirements, they design the controller and plant models of the system. Model-based testing is used as verification of the software. Finally, code is generated and deployed on a hardware platform. Mosterman and Vangheluwe [Mosterman and Vangheluwe, 2004] evaluate multi-paradigm modelling as an approach for the design of software-intensive systems using the same example application, the power window system. We extend the work of these authors with the explicit modelling of the languages and formalisms involved in the process using the FTG+PM. Also the domain specific design, the calibration of the simulation and analysis models and the deployment step have not been previously discussed using an MPM approach.

4.5 Conclusions

We have applied the FTG+PM framework to a non-trivial case study of the design of a software intensive system, namely the automotive power window controller. We have constructed the FTG and PM for the target domain. This encompasses the various phases of the MPM development of the power window. As part of each phase, we defined the appropriate formalism(s) and the relations between them. The process model was used to guide the development of the power window controller. The FTG+PM provides a complete model-driven process that is based on meta-modelling, multi-abstraction and multi-formalism modelling, and model transformation. The power window FTG+PM is a skeleton that can be extended, refined or adapted for various techniques and technology.

Chapter 5

Hybrid Simulation

In a race, the quickest runner can never overtake the slowest, since the pursuer must first reach the point whence the pursued started, so that the slower must always hold a lead
— Aristotle, Physics VI:9, 239b15

5.1 Introduction

In this chapter we present an MPM approach to the design of a hybrid simulation model. Hybrid modelling and simulation is studied extensively in academia and industry. We extend this work by explicitly modelling the interfaces involved in the co-simulation of different formalisms and the generation of these interfaces using a transformation based approach. This allows tool builders to reuse the existing simulation tools without the need for creating a new simulation tool. Transformations are used for creating the semantic adaptation between the different models in the different formalisms.

5.2 The Power Window Case Revisited

In chapter 4 we saw the use of hybrid simulation for simulating the interactions between the physical window and the designed controller. The continuous dynamics of the power window includes the ascending and descending of the window pane using an electric motor. The raising and lowering of the window originate in the environment, where users can push the up- and downward buttons or put an object between the frame and the window, to change the dynamics of the window, for example in use case 4 described in Section 4.2.1.

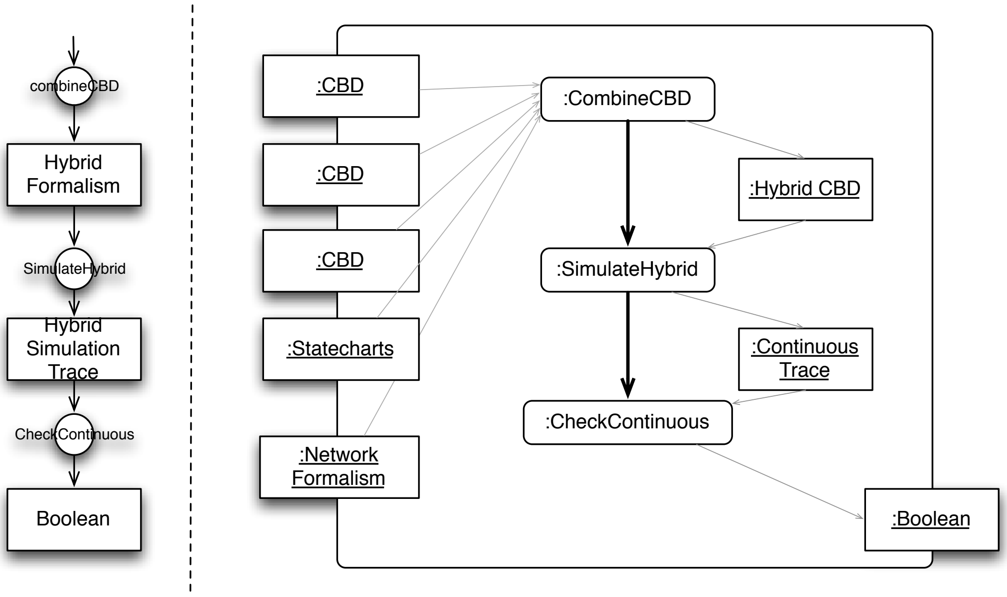


Figure 5.1: Hybrid Simulation Slice of the power window

Figure 5.1 shows the FTG+PM slice describing this process. In this chapter, we show how a simulation model can be constructed for hybrid systems where everything is explicitly modelled. This chapter focusses on the step *SimulateHybrid* and shows a technique that can be used for simulation of the power window's behaviour.

5.3 Co-simulation of CBDs

To explain the changes needed for the hybrid simulation on the CBD side we look at the use of sub-model blocks in the CBD formalism. Sub-models are blocks that themselves contain a model in the CBD formalism. They can be used for hierarchical modelling of a system which enables the reuse of components. Sub-model blocks may have multiple input and output ports.

One way to solve (i.e., compute the signal values of) a model when a sub-model block is present, is to symbolically flatten the model. The top model replaces the sub-model block with all the blocks (including ports and connections) that are present in the sub-model and reconnects the input and output ports correctly. The names of the blocks, ports and signals are made unique. As it is assumed that names are unique at every level of a hierarchy, unique names can easily be guaranteed by pre-fixing names of sub-model elements with the name of their parent. Since a sub-model can contain other sub-models flattening has to be done in a recursive way. This is not trivial when modelling large systems, since reflexive transparency is difficult to maintain.

The second option is a black-box approach where a distinct CBD simulator is used to solve the sub-model. This is called co-operative simulation or co-simulation. To be able to solve the sub-model, the simulator needs the output-input relation (direct feed-through or delayed) to be specified for the sub-model block. This can be used by to detect and isolate algebraic loops and to create a correct order of execution. When the output value of a port of a sub-model is required, the solver asks the sub-model solver for an output value on this port. A dependency graph for only this port is then constructed, sorted and solved. Finally, when the timestep is fully solved at the parent model, the other blocks in the sub-model are also solved to synchronise the timestep. This is done because there is a possibility that not all blocks are present in the created dependency graph of the port. When black-box sub-models are occur in an algebraic loop, another set of problems are encountered, though this is not the focus of this introduction.

The simulation of the system thus involves multiple solvers. These solvers work together and exchange information. The co-simulation of two CBD models working at the same execution rate is fairly easy, since they are both continuous-time formalisms. The problem is much harder when considering different rates of simulation. Zero-order hold (ZOH, holding the value of the signal for the sample period) may need to be used to assure the availability of signal values at times they are normally not computed at.

5.4 Co-Simulation with statecharts

Co-simulation of statecharts, a discrete-event formalism, and CBDs, a continuous-time formalism, is a different matter. The statechart accepts and produces discrete events while the CBD only produces continuous time signals (or discrete-time approximations thereof). Because of this, an interface is needed to meaningfully connect models in these different formalisms.

For the co-simulation of causal-block diagrams with a statechart as a sub-model of the CBD, we use the statechart-like formalism used in chapter 4.2.2. In this formalism, the events and conditions are described over the input ports of the statechart, while the actions are described over the output ports. The expressions conform to a language, ARKM3 (A keRnel for Multi-Paradigm Modelling Meta-MetaModel). ARKM3 in fact is developed as the kernel language for the AToMPM-tool and has a Python-like syntax. The language can be interpreted by the ARKM3 interpreter or compiled to Python code. Expressions defined in ARKM3 include the crossing of a threshold value from below (>!) or the crossing from above (<!). An extra property is defined on the output signal. This can be ″zero-order hold″ or ″impulse″. Zero-order hold means that the signal value on the output port will remain the same until the value is changed by another transition. The impulse will only output the value for a single sample time and then reverts to a default value.

As already stated in Section 4.2.4, the hybrid model is transformed to an encapsulated block containing:

— A statechart, containing only events

— A State Event Locator (SEL), transforming the expressions over the input ports to an event used by the statechart. The SEL contains a look-up table containing the different events in the hybrid statechart with an event that can be used in the statechart. The SEL block creates a mapping between the expressions over the signal values of the CBD execution traces to statechart events. Each entry of the look-up table contains on the left-hand side an expression used in the hybrid statechart model. The right-hand side contains an equivalent event for the statechart. This allows us to use the statechart simulator without any adaptations. The interface on the left-hand side accepts signal values while the interface on the right-hand side sends out events.

— A transducer, transforming the events coming from the statechart to a signal value that can be understood by the CBD. Like the SEL, the transducer also contains a look-up table. This look-up table contains the statechart's produced events on the left-hand side and translates them to signal value that can be used by the CBD simulator on the right-hand side. The interface on the left-hand side only accepts events and sends out signal values.

The SEL and transducer blocks work on the same rate as the surrounding CBD-model but have their own simulator. The simulator receives the values from the CBD model and evaluates the expressions (compiled or interpreted). When an event is located, a statechart event is given to the statechart simulator that in turn can react to this event. The opposite process takes place in the transducer where an event is translated to a continuous signal. In the hybrid statechart formalism, the $after()$ directive can be used to denote a transition depending on time. For each of the transitions with the $after()$ directive, a timer is created. When entering the source state of the transition with an $after()$ directive, a dedicated event is transmitted to start the timer in the SEL. For this purpose, a feedback link is needed from the statechart to the SEL. Since the SEL knows the time-step of the block, time keeping (synchronised with the CBD-model) is easy. When the timer reaches the predefined value, an event is created so the transition is fired in the statechart. It is however possible that another event already fires another transition to another state. This means that the timer should be cancelled. Therefore, on exiting the state, an event is created to reset and stop the timer. This principle is generalised to multiple transitions with the $after()$ directive.

A transformation is used to create the SEL, statechart and transducer model relieving the designer in the creation of the interfaces needed for co-simulation of a continuous-time and discrete event formalism. The transformation is possible thanks to the use of a meta-modelled language, namely ARKM3. Unique events

5.5 Test Model: The Bouncing Ball

A bouncing ball problem, shown in Figure 5.2 is a classic example of a hybrid system. The continuous dynamics of the ball (when in free flight) are given by the following

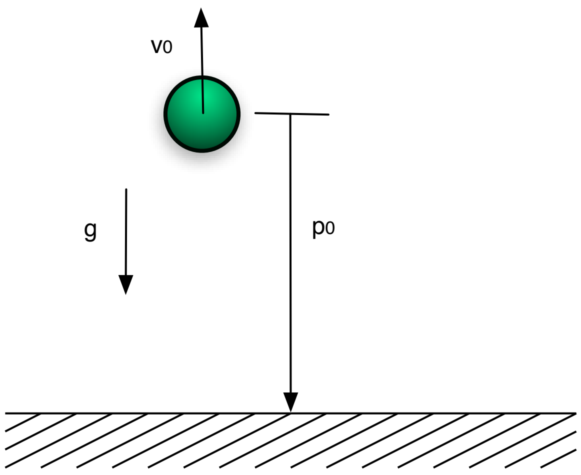


Figure 5.2: The bouncing ball problem

equations:

$$\frac{dv}{dt} = -g, \frac{dp}{dt} = v$$

where g is the acceleration because of gravity: -9.81 m/s^2. p is the position of the ball and v is the velocity. Both velocity and position are continues variables. When the ball has a (partially elastic) collision with the ground, part of the energy is dissipated and the speed of the ball is reduced. We use the hybrid formalism to model the reinitialisation of the dynamics when the ball collides with the ground. This model is shown in Figure 5.3. The statechart reacts when the position of the ball crosses zero, from above. A signal from the statechart resets the position of the ball to zero and resets the integrator to calculate the velocity with the opposite velocity minus an elasticity factor. Figure 5.4 shows the created interface for the bouncing ball model. The state event locator receives the value of the position and translates a crossing to an event e. The statechart reacts to this event by the sending the action event b. The transducer translates this to a value of 1 on the T output port.

Figure 5.5 shows the result of the simulation. The first graph shows the collision events, the second shows the acceleration of the ball which is constant for the simulation period. Finally, the velocity of the ball and the position of the ball are shown with respect to the simulation time.

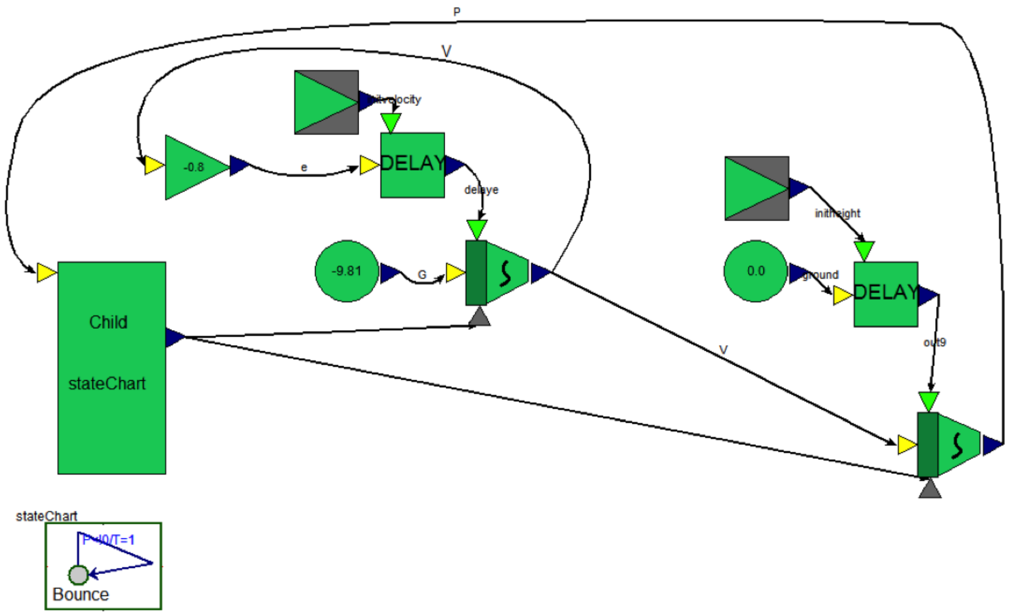


Figure 5.3: CBD and Statechart model of the bouncing ball

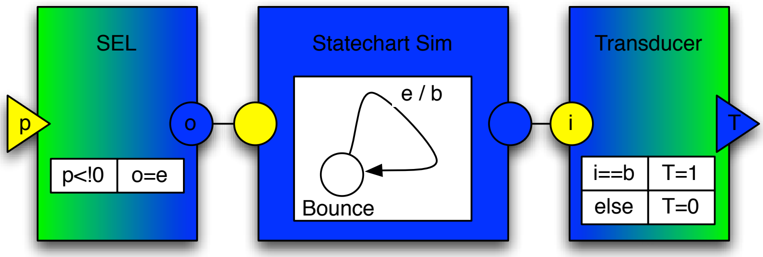


Figure 5.4: The SEL and Transducers created for the bouncing ball model

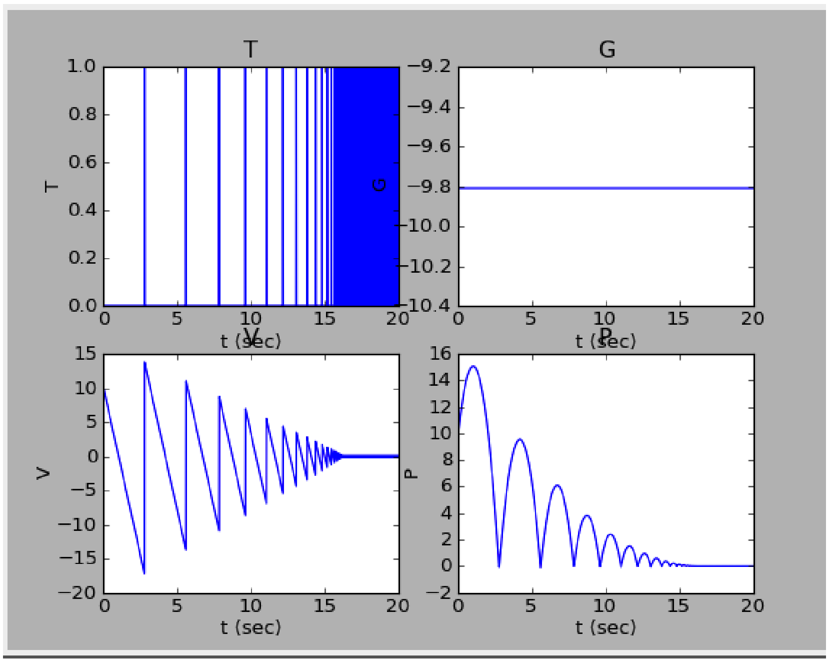


Figure 5.5: Result of the bouncing ball hybrid simulation

5.6 Results of The Power Window Simulation

The bouncing ball problem gave a simple example of the hybrid simulation model. For the power window exemplar however, the hybrid statechart has a notion of time. This can be seen in Figure 5.6, where the transition from the *emergency* state to the *neutral* state is fired *after* a certain amount of time.

Figure 5.7 shows the result of simulating the power window passenger side with the applied environment model of the defined use case 4. The passenger pushes the up button for six seconds. The window moves up until an object is detected by the infra-red sensor. As can be seen, the window moves down for 1 second after which it resumes going up.

Figure 5.8 shows the generated SEL, statechart and transducer for our power window model. As can be seen, a feedback loop exist between the statechart and the SEL. On entering and exiting the *emergency* state, dedicated events are created for starting and stopping the timer.

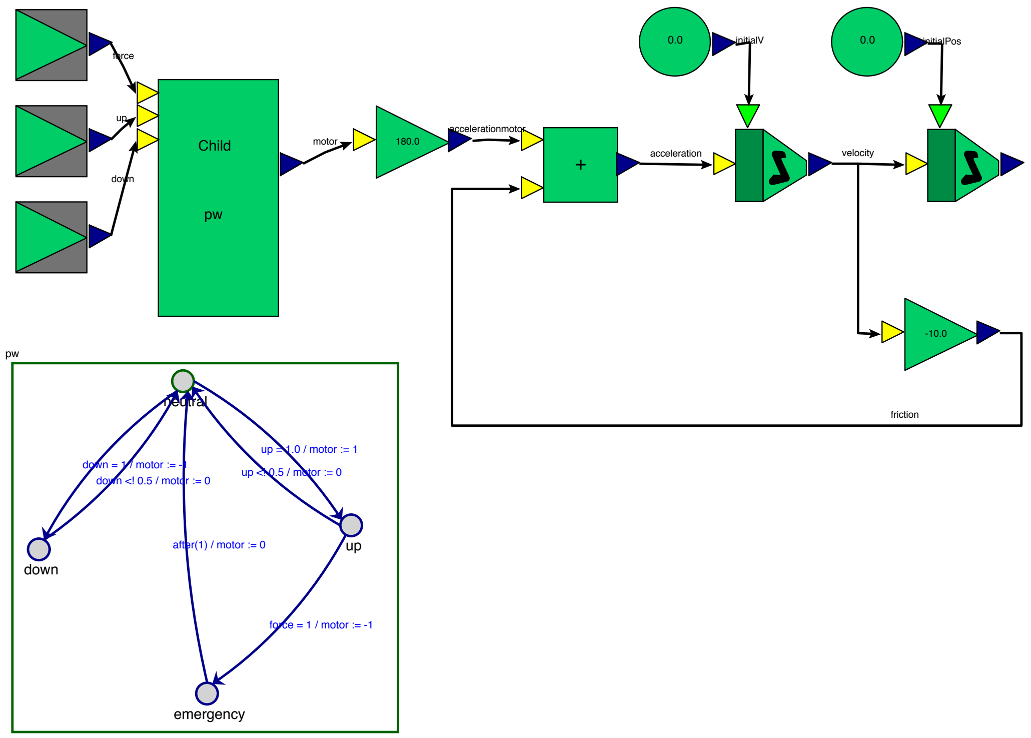


Figure 5.6: Hybrid model of the power window

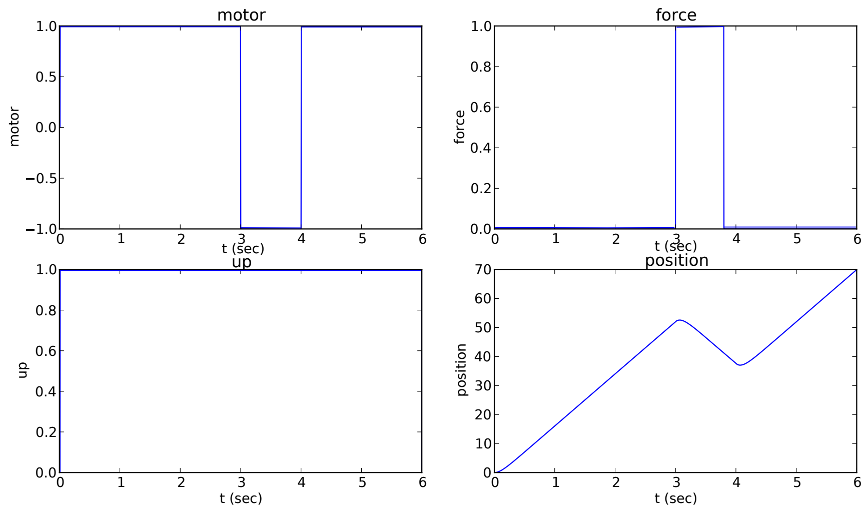


Figure 5.7: Results of the power window hybrid simulation

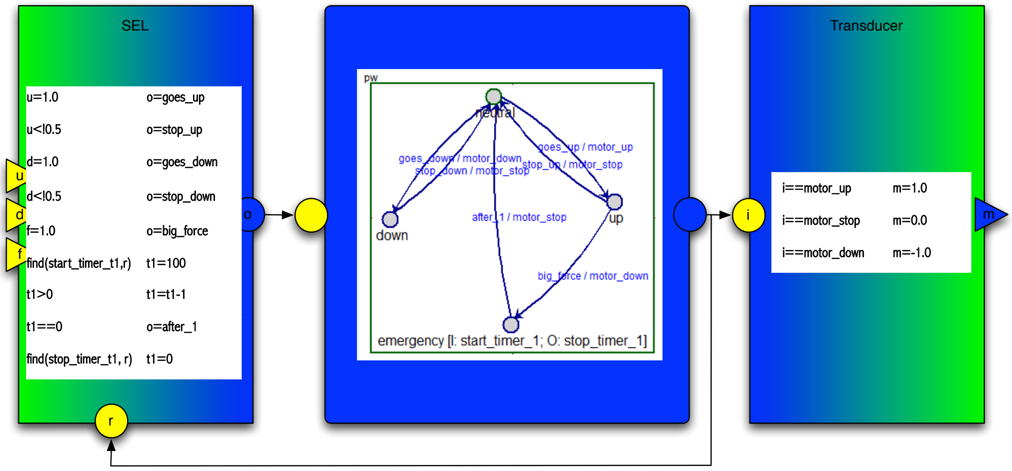


Figure 5.8: SEL, statechart and transducer of the power window simulation

5.7 Related Work

As already mentioned, the novelty of the approach is quite limited since many hybrid simulation approaches already exist in academia and industry. One notable example is the co-simulation of Stateflow® and Simulink® by MathWorks®. Here, causal block diagrams are co-simulated with a statechart-like formalism as in our approach.

The Modelisar Functional Mock-up Interface (FMI) [Blochwitz *et al.*, 2011] is an industrial standard for model exchange and co-simulation.With co-simulation, data is exchanged between different subsystems at discrete communication points. Between the synchronisation points, the subsystems are independently solved by their solver. A master algorithms controls the data exchange between subsystems. The FMI allows standard, as well as advanced master algorithms, e.g., variable communication step sizes, signal extrapolation, and error control.

Modhel'X[Boulanger and Hardebolle, 2008] and Ptolemy[Eker *et al.*, 2003] are component oriented multi-formalism approaches. They consider that the semantics of a modelling language is given by its Model of Computation (MoC). An MoC is a set of rules defining the relations between the elements of a model, the operational semantics. The meta-model is similar for all languages but the semantics are given by a corresponding MoC, thus defining different behaviours. The heterogeneous models have a hierarchical organisation, each with their own MoC. At the boundaries between the different MoCs combinations can occur. In the Ptolemy approach this boundary is fixed and coded statically in the kernel of the tool. ModHel'X on the other hand allows the explicit specification of the boundary.

Our work is very similar to the Modhel'X approach. The interactions between the continuous part and the discrete part are explicitly modelled using the ARKM3 language. Transformations are provided to transform the action language, present in the hybrid statechart to an event list that can be understood by the statechart simulator.

A comprehensive overview of the field of numerical simulation for hybrid dynamic systems can be found in [Mosterman, 2007]. The overview addresses topics such as event detection and location, mode transitions, reinitialisations and classifies pathological behaviours that require special attention.

5.8 Conclusions

In this chapter an MPM approach to hybrid simulation is proposed. The solution models the interface between the different formalisms explicitly using the ARKM3 language. Model transformations are used to automatically derive a boundary model from a hybrid formalism. The contribution is shown on the hybrid simulation of CBDs and statecharts. The simulation model is validated using the simulation of the bouncing ball problem and the power window system.

Chapter 6

Simulation of Deployed Systems

If you're trying to train a pilot, you can simulate almost the whole course.
You don't have to get in an airplane until late in the process.
— Roy Romer

6.1 Introduction

During the process of deployment onto hardware a plethora of configuration choices have to be made in the middleware. These choices range from the mapping of software functions onto tasks and assigning these tasks a priority, to parameters that affect the sending and receiving of messages on the bus. Because of the impact these choices have on functional and extra-functional behaviour of the system, a method is needed to evaluate candidate deployment solutions.

A complicating factor is the reusability of functional components. There is no guarantee that a component will behave as intended in a new hardware/middleware configuration with respect to performance requirements such as timing. This non-compositionality means that during integration, these behavioural properties must be evaluated.

In the literature, performance analysis models have been proposed for this purpose. A well known technique is schedulability analysis, that uses the worst-case timing behaviour of the different components and checks whether an application can make the proposed deadlines. Examples of schedulability analysis can be found in [Lakshmanan *et al.*, 2010; Pop, 2007; Palencia and Gonzalez Harbour, 1998; Tindell and Clark, 1994]. These

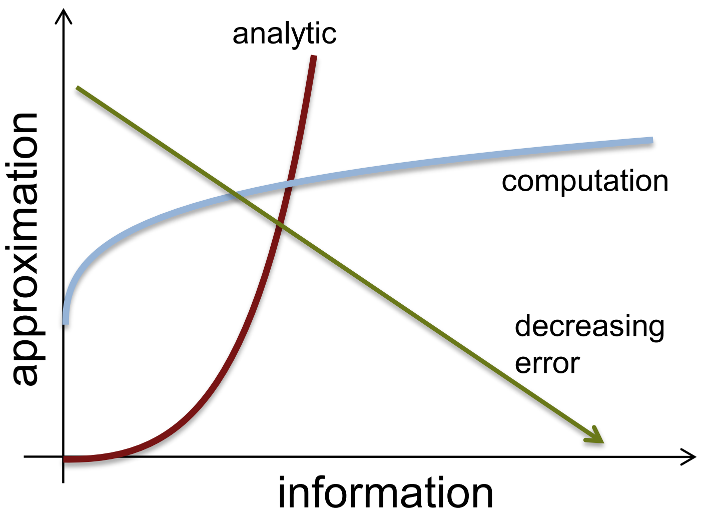


Figure 6.1: Comparing analysis and simulation, from [Mosterman, 2011]

models, however do not take a lot of low-level parameters into account, like software and hardware buffering and communication stack parameters. Taking these low-level parameters into account using an analysis model is technically hard. By increasingly adding information to an analysis model, the approximation increases exponentially as shown in Figure 6.1. Simulation models are more appropriate in this case.

Another issue with analysis models arises: many applications are not time-critical. The analysis models are constructed for the purpose of worst-case and best-case performance analysis. Though performance of the system depends on the average response time, which needs to be analysed and minimised. This is also true for many time-critical functions where usability needs to be analysed as well as safety. Finally, in a periodic activation model, each time a message is transmitted or received, a task (message) may need to wait up to an entire period of sampling delay to read (forward) the latest data stored in the communication buffers. Analysis models add worst case delays at each of these step to obtain the worst-case latencies of paths. The probabilities of these worst-case delays is very small, as shown in [Di Natale *et al.*, 2009].

In this chapter, the appropriateness of the DEVS formalism is evaluated for the purpose of low-level deployments modelling and simulation. A simulation model is constructed for the power window system. Finally, a generic model is derived from this model and a transformation is defined to automatically construct the simulation model from the deployment model.

6.2 The Power Window Case Revisited

To show the contribution, the power window case is also used. Figure 6.2 shows the FTG+PM for the purpose of simulation of the deployed power window. As can be seen, as a deployment model, the AUTOSAR model of the power window is used. The plant and environment models are used for co-simulation purposes so the overall behaviour of the system can be observed. From a fully deployed AUTOSAR model and a performance model (containing the timing execution distributions or Worst-Case Execution Time behaviour of the software components on the hardware) a transformation is used, *ToDeploymentSimulation*, to create the DEVS simulation model. This model is executed by the DEVS simulator to obtain a set of traces that can be checked with the requirements of the system.

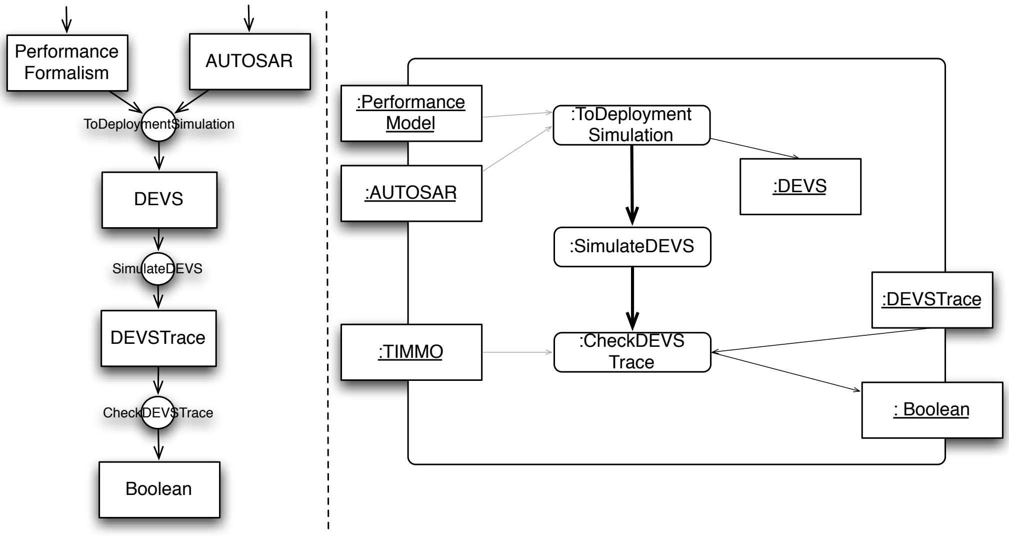


Figure 6.2: The DEVS Simulation Slice of the Power Window need to add a CBD port to this

6.3 Appropriateness of DEVS for Deployment Modelling and Simulation

DEVS is a formalism for modelling discrete-event systems in a hierarchical and modular way, rooted in systems theory. As discussed in Chapter 2, DEVS support two kinds of models. The first defines the behaviour of a primitive component and is called the atomic model. The other is the coupled model and is a composition of two or more atomic models that are explicitly connected. The coupled model itself can be part of an other coupled model. This allows for a hierarchical construction of DEVS models. We use the Classic DEVS variant with ports as it best matches the AUTOSAR semantics.

From an abstract point of view, the DEVS formalism provides excellent features for modelling AUTOSAR based systems. Here is a list of some properties of DEVS and their mapping to properties of automotive software and systems.

— Concurrency: Multiple processors and communication buses are concurrent in an automotive system. The semantics of DEVS coupled models supports concurrency by appropriate interleaving of the discrete-event behaviour of individual sub-models.

— Time: Real-time performance is a crucial property of automotive embedded software. End-to-end latencies are part of the requirements for these applications. The time advance function of an atomic DEVS model can be used to model latency.

— Events: Event-triggered and time-triggered architectures use triggers in the form of either external events or timing events to start certain pieces of functionality. DEVS implements reaction to events using the external transition functions.

— Priorities: Some automotive buses use a priority-based mechanism to arbitrate the bus (for example, the CAN-bus). DEVS supports this by means of a tie-braking function to select an event from the set of simultaneous events.

— Simulation of the physical part of the system: DEVS is a very general formalism and is able to simulate different other formalisms [Vangheluwe, 2000]. This generality stems from the infinite possible states that DEVS allows to model and the (continuous) time elapse between the different state transitions. The hierarchical coupling techniques are used to integrate the different formalisms using DEVS as a common denominator.

Some concrete examples of the mapping of AUTOSAR concepts to the DEVS formalism are given below.

— Runnables, the RTE and other basic software components are executable entities. They are pieces of code that change the state of the system during their time of execution. These can be mapped to the time-advance and the internal transition functions of an atomic DEVS model.

— The runnables are mapped onto tasks that run on a given processor. The tasks run on the AUTOSAR operating system. When an executable entity has finished its execution, the operating system could change state. This also maps to the internal transition function of an atomic DEVS.

— Messages and input/output hardware can interrupt the execution of the tasks. This could take time and could possibly trigger the execution of a runnable. DEVS can implement this using the external transition function.

6.4 The AUTOSAR Simulation Model

In this section we construct a simulation model of an AUTOSAR-based power window. The simulation model is used to investigate the timing behaviour of the application.

The application controls the window on the passenger side, though both passenger and driver are allowed to open or close the window. When an object is present while closing the window, it will automatically detect this and lower the window. Figure 6.3 shows the AUTOSAR platform independent software architecture.

The two 'Control' components read out sensor signals from the buttons that control the window. The driver side component is also responsible for applying child protection on the power window and checking whether the ignition of the car is on. The 'Load_Sensor' component reads out the resistive force being placed on the window. When the execution of the runnables inside these components have finished, they make the sensor values available to the 'Logic' component. The 'Logic' component decides how to control the window using these sensor values and calculates the direction and speed of the window. The 'DC_Motor' component uses this to physically control the window. Both the 'Control' components, the 'Logic' component and the 'Sensor_Load' component are triggered by a periodic timing event (every 1 ms). The 'DC_Motor' component is triggered by the arrival of the 'Direction' signal from the 'Logic' component. In this case, all software components contain exactly one runnable.

The hardware contains two microcontrollers. One on the driver side and the other on the passenger side. Both hardware units communicate through a CAN-bus with a bandwidth of 500 kbit/s.

Figure 6.3 shows the full deployment of the application on the hardware. Major design decisions include the mapping of the components to the different control units. On the driver side, a single task executes the 'DriverControl' runnable. This task also transmits three signals on the bus, mapped to two different messages. The arrival of two signals in the communication stack cause the transmission of a message on the bus, while the other signal is only stored in the message without causing a transmission. On the passenger ECU two tasks are configured, one high priority task executing all the time-triggered runnables and a lower priority task executing the 'DCMotor' runnable.

6.4.1 The coupled DEVS model

Figure 4.33 shows the coupled DEVS model representing the deployment of the power window application. A short overview of the components is given below:

— DEVS Driver model: This atomic DEVS model represents the behaviour of the AUTOSAR operating system and the task running on the driver ECU. The time-triggered events from the timer module interrupt the execution of the model. During execution, the model sends two messages to the CAN transmit buffers.

— DEVS Passenger model: This atomic DEVS model represents the behaviour of the AUTOSAR operating system and the task running on the passenger ECU. The time-triggered events from the timer module and the reception of messages from the CAN receive buffer interrupt the execution of the model.

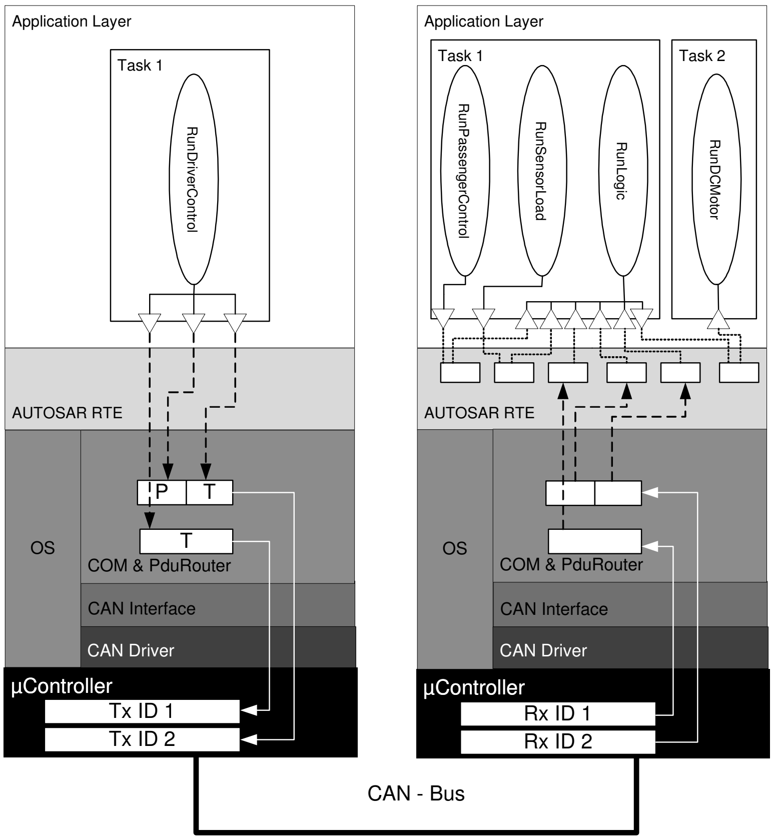


Figure 6.3: The power window application deployed on the hardware. P(ending) is a signal that does not cause the message to be transmitted in contrast to the T(riggered) signals. Signal and message names are removed for reasons of clarity.

— DEVS timing event generators: These generates the timing events to activate the time triggered runnables. An event is generated every 1 ms.

— CAN transmit buffer: The output buffers of the CAN bus contain the frames that have to be sent over the CAN bus. Because of the priority mechanism, it is necessary to keep these in the buffer.

— CAN receive buffer: The input buffers receive messages from the CAN bus. It forwards all received messages to the passenger model.

— CAN bus: The CAN bus is responsible for the physical transmission of frames. The delay that occurs when sending this frame depends on the size of the message and the speed of the bus. When two or more frames are available in the output buffers, the select function needs to select the message with the highest priority.

6.4.2 The model of a single control unit

The simulation model of the AUTOSAR basic software is abstracted to the component level. Execution times of these components are based on scenarios to match the implementation behaviour of the component.

Both driver and passenger models are similar, only the configuration of the runnables, tasks and Basic Software (BSW) parameters differ. The state of the atomic DEVS model of a control unit (driver or passenger) can be divided into 3 major parts.

At the finest level of granularity there are the runnables. All the runnables in the power window application read the input values, do calculations on them and then transmit the result if needed. While doing these calculations, the runnable is executing for a certain amount of time. This is reflected in the behaviour of the runnables in our simulation model. The time-advance for a single runnable is a configuration parameter of the simulation model.

Although this type of behaviour works for the 'Driver_Control' component, a finer granularity should be used to describe the way the runnable is executed. The runnable reads out three values from the hardware in a sequential order, sending out the value after every read in a different signal. This can be modelled by splitting up the execution in three parts using a state machine, where every state ends with a write operation of a signal.

The next abstraction level in the state of a control unit introduces tasks. A task contains a set of runnables. When the task gets activated, it chooses the first runnable in the set where the state has been changed to activated. The task also keeps track of the time the runnable has executed, since the task could be preempted by an other task with a higher priority. Tasks are also responsible for sending the signals and messages through the communication stack. It therefore has access to the RTE and communication stack modules. These modules can be regarded, like the runnables, as entities that change the state of the model while taking a specific amount of time.

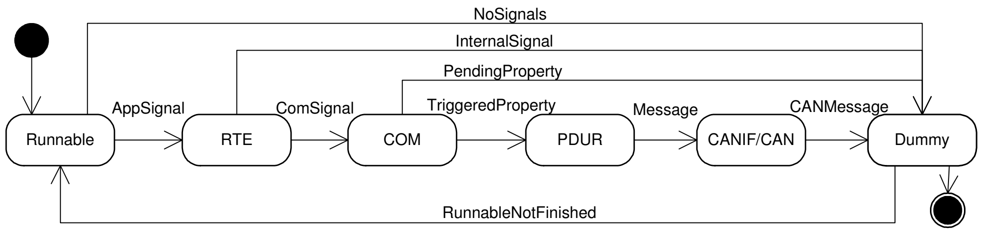


Figure 6.4: State diagram of a task

Here is an overview of the responsibilities of each individual state in the task model, shown in Figure 6.4:

— RTE: In the run-time environment state, the task keeps track of the data buffers of the interfaces. When an intra-ECU signal is written, the RTE state places this value in the corresponding receive buffer. To signal the underlying operating system model that the work is done, the task state is changed to a 'DummyState' by the internal transition function. In some cases, the RTE state produces activation events that are used by the operating system model to activate certain runnables/tasks, as in the 'DCMotor' case. For external communication in the 'DriverControl' component, the RTE state changes the application signals to communication signals and transitions to the COM state.

— COM: The COM state mimics the behaviour of the COM module in the AUTOSAR basic software. When the COM is activated, it places the messages received from the RTE state in the configured message buffers. The COM state checks the signal properties and message modes. Based on this, it decides whether to make the message available for the PDU-router state or to transit to the 'DummyState'.

— PDU-R: The corresponding AUTOSAR module is normally used to route the messages to the correct interface. Since we only have a single bus in this case study, the PDU-router state is not used, it only makes the received messages from the COM state available to the CANIF/CAN state after execution.

— CANIF/CAN: This state represents the AUTOSAR interface and driver of the CAN bus. The CAN and CANIF modules are used to place the messages into the CAN transmit buffers. The module adds the message priority and length to the message. Occasionally, it buffers certain messages when the hardware buffers are full. These modules need to get executed in an atomic way, so the task cannot get preempted during the execution of these modules. The CANIF/CAN state has a similar behaviour, it adds message priority, length and the number of the buffer before making the CAN message available to the operating system model.

— DummyState: The DummyState is introduced to notify the operating system that there could be CAN messages pending for transmission to the buffers, as is the case in the 'DriverComponent' or that there is an activation event pending when the 'runLogic' has finished execution. It does not take any time to execute. In the 'DriverComponent', it can happen that the runnable is not fully finished after the DummyState. In this case the task reactivates the runnable. Yet another situation can occur on the passenger side, where multiple runnables are mapped to a single task. Here, the task looks whether another runnable can be executed. If no other runnables are available, the task gets suspended by the operating system.

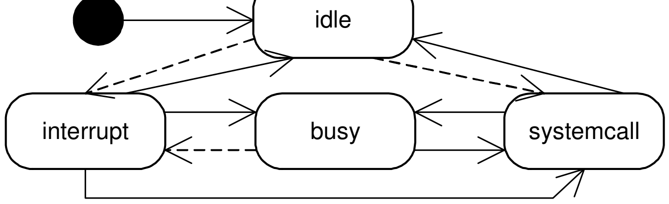


Figure 6.5: State diagram of the operating system. Full lines represent internal transitions, dashed lines external transitions

At the coarse grained level, shown in figure 6.5, there is the AUTOSAR operating system. The operating system keeps track of the tasks in the model. If there are no tasks running, the operating system is in an *idle* state. The model can only get out of this state by means of an external event. External transitions can occur when a timing event is received by the control unit. The operating system checks, based on the received event, whether runnables and tasks need to be activated. Since this also takes a certain amount of time, the operating system changes to the *systemcall* state to compensate for this delay.

After this delay, the model goes to a *busy* state where the substates of a task are executed. The time-advance of the current state is looked up, by querying the current running runnable or executable entity in the running task. On the passenger side, another type of external transition can occur because of the reception of the CAN messages. In the AUTOSAR implementation, the ECU will respond to an interrupt and take the message out of the hardware buffer and process the message using the communication stack. The simulation model reflects this behaviour by interrupting the current state for processing the message in the *interrupt* state. After the processing it returns to the previous state.

When an internal transition occurs in the *busy* state, the model notifies the running task that the current entity has finished its execution. When the task is in the 'DummyState' it reads out the activation events and the messages to be transmitted on the CAN-bus. It causes a transition of the operating system, to suspend the current task and/or activate other tasks based on the activation events, possibly even preempting or suspending the running task. This is reflected as before in the *systemcall* state.

While intuitively the executable entities, like the RTE, runnables and other communication stack modules, are responsible for keeping track of the elapsed time and their execution time, in the model they are not. Since the code for the RTE, COM, PDUR and CANIF is shared between the different tasks running on the operating system, the tasks are responsible for storing the time-advance and elapsed time of the entities executing within the task. This is to prevent a mismatch in timing behaviour when a task is preempted by another task.

6.4.3 The CAN-bus model

The CAN-bus model introduces the delays imposed by physically transmitting a frame on the bus. Figure 6.6 shows the state diagram of the simulation model. The model starts in an IDLE state with an infinite time advance. This represents the state when no messages are being transmitted on the bus. It changes state when one or more messages are put into the CAN transmit buffer. The tie-braking function checks the priority of the message and selects the one with the highest priority.

It then changes to the BUSY state. This state reflects the physical processes of transmitting the frame onto the communication medium. It stays in this state based on the length of the message and the configured bandwidth of the bus: $t = (1/speed) * size$. On completion,

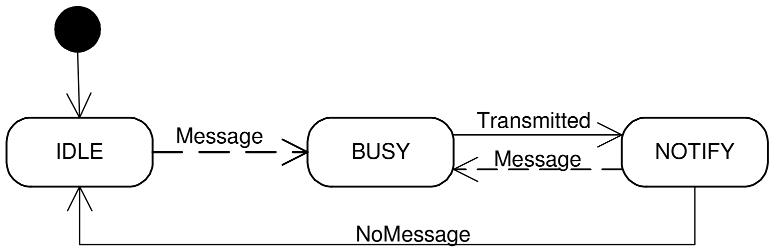


Figure 6.6: State diagram of the CAN-bus simulation model. Full lines represent internal transitions, dashed lines external transitions.

the model writes the message to the CAN receive buffer on the passenger side. It also notifies the transmit buffers that the bus is ready for arbitration. In case there are pending messages, it returns to the BUSY state. Otherwise the bus returns to the IDLE state.

6.5 Co-Simulation

In the previous section all aspects of the computational part of the power window system has been modelled. From this model, it is already possible to evaluate the timing behaviour of the application, because the DEVS formalism interleaves the executions of the different runnables, basic software modules and buses with the incurred delays. We cannot however check the behaviour of the physical system with this model. As with the hybrid simulation of Chapter 5, a simulation of the DEVS formalism with the plant and environment model is needed to evaluate the full behaviour of the system together with the execution of the software functionality within the defined runnables.

For this, three behaviours need to be added to the previous described model:

— Execution of the software functions: The implementation of the function is executed. The output values of the runnable are written in the RTE.

— Passing data in the events: The functions need the input data of other runnables. Like in the implemented AUTOSAR system, the values are passed through the communication stack, while building up a frame that can be transmitted in the bus. The communicated data is piggy-backed in the event that is exchanged between the different coupled DEVS models.

— Integration of the plant and environment models in the simulation model: The plant and environment CBDs need to be co-simulated with DEVS. It is easy to accomplish this by embedding the full CBD model and solver in an atomic DEVS block. The time-step is generated by the time-advance function of the DEVS atomic model. Since the same time-step is used for all the blocks in the model, it is necessary to change the select function of the coupled model so it reflects the same source to sink execution as in the CBD simulation model. The topological sort algorithm is used for this purpose.

The continuous signals from the environment and plant model are stored in the RTE of the ECU model. This is a simplification we made during the design. The different runnables sample (polling-based) the values, as in the AUTOSAR implementation. When an interrupt based approach is needed, a dedicated atomic DEVS model can be used to detect the interrupt and store the value again in the RTE. The RTE can start different tasks or set a number of events in the RTOS, reflecting the actual behaviour.

6.6 A Generic Simulation Model

The simulation model previously described using the power window case study is a generic model. In this section the implementation, using the Python programming language, is described as well as the transformation from a model towards this simulation model.

6.6.1 Implementation

The implementation of the simulation model is done using the pyDEVS tool [Vangheluwe, 2013]. Figure 6.7 shows a simplified class diagram of the implemented simulation model.

EcuModel

The ECU atomic model has four distinct high-level states already shown in Figure 6.5: (a) idle, (b) busy, (c) interrupt and (d) systemcall. For communicating with the other atomic models in the coupled DEVS model, the EcuModel has four input ports and two output ports. Depending on the configuration not all of them need to be connected to another model:

— FromRxBuffer: this port receives an event, with a piggybacked CAN frame, from the connected RxBuffer. Depending on the configuration of the AUTOSAR ECU model, the external transition functions will either do the unpacking of a frame in an interrupt based way or in a polling based way:

  — Polling: store the frame in the CAN/CANIF module. This will set the state of the ECU to an *interrupt* state. The internal transition time is a configurable parameter of the model.

  — Interrupt: go through the several layers of the communication stack and unpack the different signals to application signals in the RTE. The state of the ECU is also changed to *interrupt* state, but the time-advance is a sum of the different configurable timing parameters of the communication stack modules. Note that several task activation events can be generated from the unpacking of the frame

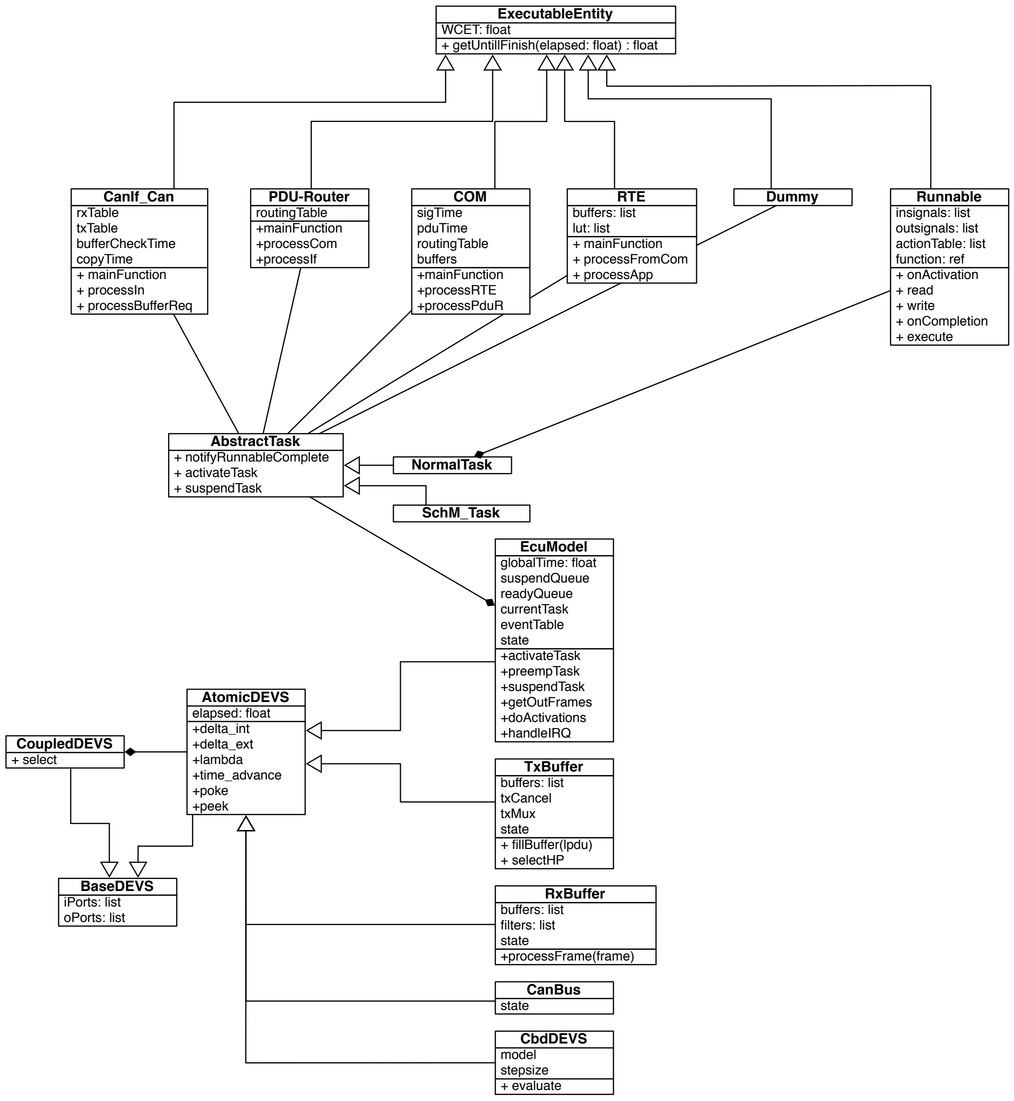


Figure 6.7: Simplified class diagram of the Simulation Model

into signals. These are stored and evaluated after the *interrupt* time-delay is finished. It results in the transition of the *interrupt* state to the *systemcall* state.

— FromTxBuffer: the FromTxBuffer port receives an event to notify the processor that a frame has been transmitted on the CAN bus. This is to allow notification of transmission events to the application layer. These events are, like the frames, processed within an interrupt or by using a polling mechanism. The behaviour is as described above. The FromTxPort is also used for returning frames out of the TxBuffer when the configuration has the *cancellation* option defined. This will store the frame in the CAN/CANIF software buffer for resending when the TxBuffer is available.

— FromScheduleTable: this port receives timing events for the activation of time-triggered tasks. A lookup table is used to activate tasks this way. The EcuModel will activate a set of tasks by changing to the *systemcall* state.

— FromPlant: allows the IO events from the plant and environment models to activate tasks and set the values in the application layer. The data from these IO events is stored in the RTE. If an interrupt is attached to it, tasks can be activated in the model. If the driver is configured as polling based, no time penalty is incurred since this is already reflected in the polling loop of the IO task.

— ToTxBuffer: for sending an event, with a piggybacked CAN-frame, to the TxBuffer

— ToPlant: this port is used for sending an event to the plant model, it contains a piggybacked list of *CBD Inport names* and values so the plant can respond to a change in the system behaviour.

The state of an ECU contains a set of tasks. These tasks can be in four states reflecting the behaviour of the AUTOSAR OS: (a) running, (b) suspended, (c) waiting and (d) ready. As already explained using the case study, the task executes the different runnables and depending on the configuration transmits messages using the communication stack. A special type of task: *SchM_Task* is used for calling the different main functions of the communication stack. Most of the communication stack design choices that are available in the AUTOSAR standard are present in the simulation model. This is because the configuration of these parameters has a big impact on the timing behaviour of the application. These include, from the COM module: transmission signal properties: *TRIGGERED* and *PENDING*, transmission modes: *DIRECT*, *CYCLIC* and *MIXED*, and other configuration properties like the *Minimum Delay Timer* (MDT) to prevent a message being transmitted before a certain time has passed and the *N-Times* parameter to retransmit messages for a number of times. From the CAN interface and CAN driver module, *software buffering*, *cancellation* and *buffer multiplexing* is supported. These options define when and how a certain frame is placed in the hardware buffer and thus have an impact on the behaviour of the transmissions of frames on the bus.

TxBuffer and RxBuffer models

The TxBuffer and RxBuffer work independent of the processor. In most common processor system, these buffers are within the communication controller of the ECU. The CAN controller, implementing the CAN-bus protocol makes sure how a message is placed on the bus. Both the TxBuffer and RxBuffer have a specifiable number of internal buffers. Each ECU model that needs to transmit or receive a message on the CAN bus needs its own set of buffer models. The TxBuffer has two input and two output ports:

— FromEcu: The TxBuffer receives a piggybacked frame from the ECU Model via this port. It also contains the buffer number where the frame should be placed.

— ToCAN: Depending on the configuration of the CAN module, either the first or highest priority message is put on the port to the CAN-bus. The event contains a complete frame. Because multiple buffers in the whole system can transmit a message at the same time, the tie-braking function is used to select the highest priority message amongst the competing messages.

— FromCAN: The CAN-bus, transmits a notification message to the buffer when the transmit of the frame is complete. All transmit buffers receive this event, though only the source buffer of the transmitted message forwards this to the ECU.

— ToEcu: This port is used to notify the processor that the frame has been transmitted on the bus. It also possible to change the content of a buffer when *Cancellation* is supported. The TxBuffer returns the content of the buffer to the ECU Model for storage in the software buffers.

The RxBuffer is used to filter, store and forward messages to the ECU Model. The filter is implemented, as in most controllers, on a per internal buffer basis using the ID of the accepted frame together with a mask. The acceptance filtering starts at the first internal buffer and accepts or denies the message. The filtering mechanism stops when the frame is accepted in one of the buffers or there are no buffers left. The input and output ports are very similar to those of the TxBuffer model.

CanBus model

The CAN-bus model implements the physical transmission of the CAN-frame. It has three ports to communicate with the buffer models:

— InFrame: The InFrame port accepts incoming frames. It connects to all the TxBuffers. Frames are only accepted when the bus is in an IDLE and NOTIFY state.

— Notify: The Notify port lets the TxBuffer models know that the bus is no longer in a busy state (transmitting a frame on the bus).

— OutFrame: A frame is put on the OutFrame port when the transmission delay is completed. It connects to all the RxBuffer models.

Timing Event Generation

The timing event generation implements the schedule table mechanism of the AUTOSAR OS. It generates timing events to activate periodic tasks. The only output port transmits this event to the attached ECU model. The ECU model activates the needed tasks based on this timing event.

CbdDEVS model

The CbdDEVS atomic model implements all the aspects of the co-simulation part. It contains the CBD model and solver presented in chapter 5. The internal transition time of the atomic model is set to the time-step of the CBD model. At each internal transition, the evaluate function of the CBD simulator is called. This results in the creation of (a set of) events with piggybacked values that are used by the ECU model. External events also contain values for the input ports of a CBD model. The model contains a single input and a single output port to transmit these values.

Events

Since the events in the simulator contain a lot of piggybacked values, we take a closer look at different events transmitted between the different DEVS components.

— Frame: The frame has a CAN ID that is used for the priority on the bus. The size attribute is used to compute the incurred delay on the CAN bus. The data attribute contains an IPDU. The frame is used as an event between the TxBuffer model, CAN-bus model and RxBuffer model.

— LPDU: The LPDU contains the frame information together with a buffer ID. It is used between the RxBuffer and ECU model and the TxBuffer and ECU model.

— IPDU: The IPDU contains a sorted list of signals. The IPDU is used in the communication stack and is packed and unpacked by the COM module.

— Signal: The signal contains a value that can be used the application. The RTE converts application signals to communication signals so they can be used in the COM module.

— Activation Event: The activation event comes from the scheduleTable model and contains a set of tasks IDs and runnable IDs that need to be activated on reception of the event.

— IO Event: The IO event is used between the ECU model and the CBD models. It contains a name/value pair.

6.6.2 Transformation to the Simulation Model

The simulation model has to be initialised with all the parameters like the number of ECUs, the tasks and runnables per ECU and all of the properties needed for it, like the priority etc. A transformation is defined using the MOF Model to Text transformation language (MTL) [OMG, 2008a]. This is a template based transformation language, allowing us to output Python code and inject the configuration parameters from the model. Listing 6.1 shows an excerpt of the transformation. The transformation starts by generating all the components of the ECU model and setting the parameters. This code is generated for each of the ECUs present in the source model. Finally a coupled model is constructed using these components.

Code Listing 6.1: excerpt from the transformation to the simulation model

```
[for (e : Ecu | aModel.theSystem.ecu)]

#making Ecu ECU_[e.name/] of type [e.type.name/]
rte_ECU_[e.name/] = RTE([e.type.rteTime/])
com_ECU_[e.name/] = COM( [e.type.comSignal/],  [e.type.comIPDU/])
        [for (pdu : IPDU | e.comConfig.ipdus)]
                [if (pdu.oclIsTypeOf(RxIPDU))]
com_ECU_[e.name/].addRxRecord("[pdu.name/]",
     [ '[' /]
     [for (sig : ComSignal | pdu.MappedSignals)][sig.name/]
     [/for][ ']' /])
                [/if]
                [if (pdu.oclIsTypeOf(TxIPDU))]
com_ECU_[e.name/].createPduBuffer("[pdu.name/]",
      [pdu.MappedSignals->size()/])
                        [if (pdu.oclAsType(TxIPDU).mode = 0)]
com_ECU_[e.name/].addTxPduRecord("[pdu.name/]","DIRECT")
                        [/if]
                        [if (pdu.oclAsType(TxIPDU).mode = 1)]
com_ECU_[e.name/].addTxPduRecord("","CYCLIC")
                        [/if]
                        [for (signal : ComSignal | pdu.MappedSignals)]
                         [if (signal.oclAsType(TxComSignal).pending)]
com_ECU_[e.name/].addTxSignalRecord("[signal.name/]",
        "[pdu.name/]", [pdu.MappedSignals->indexOf(signal)/],
        "PENDING")
                         [/if]
...
[/for]

#make the simulator
class SystemSim(CoupledDEVS):
    def __init__(self, name="thesim"):
        CoupledDEVS.__init__(self, name)
[for (e : Ecu | aModel.theSystem.ecu)]
                self.ECU_[e.name/]Table = self.addSubModel(
                           ScheduleTable("ECU_[e.name/]Table",
                           ECU_[e.name/]Table))
                self.ECU_[e.name/] = self.addSubModel(
                           OSSim("ECU_[e.name/]",
                           ECU_[e.name/]_tasks,
                           [e.name/]EventMap, rte_ECU_[e.name/],
                           com_ECU_[e.name/],
                           PDUR(0),canif_ECU_[e.name/]))
                self.connectPorts(
                           self.self.ECU_[e.name/]Table.OUTPORT,
                           self.self.ECU_[e.name/].SCHEDULETABLEORIOIN)
[/for]
...
```

6.7 Experimental Setup and Results

6.7.1 Calibration of the model

Before using the simulation model, it has to be calibrated (i.e, parameter values have to be estimated). Since we focus on the real-time behaviour, time delays for all actions need to be measured on all used hardware platforms before the simulation model can be used. The model is then calibrated using these values. Here are the measurements that need to be completed:

— Execution time of all the runnables or states in the runnables, without the calls to the RTE;

— Execution time of activating or suspending tasks as well as the context switching times;

— Execution time of the transmission and receiving of messages in every part of the communication stack including the RTE.

The execution times can be a distribution of execution times based on a scenario or the worst-case execution times depending on the interest of the developers. Chapter 7 focusses on obtaining the different calibration parameters using a MPM approach.

6.7.2 Results

The results of the simulation model execution of the power window application are shown in Figure 6.8. The Gantt-chart shows the interleaving of the different tasks and which executable entity is being executed in blue. The results shows how both processors execute the defined tasks in the correct order. The driver unit sends out two different messages on the CAN-bus during the execution of Task_1ms. These signals are received by the passenger ECU after the delay introduced by the CAN-bus. The interrupt and system call states of the processor are shown in red. In parallel, the passenger ECU executes the passenger input components and logic in its Task_1ms. Based on the inputs, it controls the direction of the window in the second task. No buffers are overwritten before being read during the execution of the model.

Since we are interested in the behaviour of the system, a similar graph as in Chapter 5 can be constructed from the co-simulation with the plant and environment models. Figure 6.9 shows the environment inputs and the response of the power window system.

The results obtained from the simulation model helps the AUTOSAR developer to analyse the impact of the deployment choices on the behaviour of the system. It can be used to explore the various trade-offs while deploying automotive applications to AUTOSAR based ECUs. Figure 6.10 shows the simulation of the same system with a different configuration. We observe in this figure that the response time of the system is different (much slower) than in the first configuration. Another simulation result can be observed in Figure 6.11 where the transmission properties of a signal are changed from *Pending* to *Triggered*. As can be seen, the CAN-message 1 is transmitted two times.

6.8 Related Work

The DEVS formalism has also been used for developing embedded real-time applications. In [Wainer *et al.*, 2005], a model-driven method to develop these real-time embedded applications is introduced. For the evaluation of AUTOSAR-based systems both analysis techniques and simulations methods are available. On the analysis side, techniques are available to predict the timing behaviour after deployment. These techniques can yield worst and best case response times of the tasks and messages [Hamann *et al.*, 2006; Pop *et al.*, 2002].

On the simulation side, models can be built at various levels of abstraction, ranging from functional simulation with little timing information to true cycle simulations using binary code.

Metropolis [Balarin *et al.*, 2003] is an inter-disciplinary research project that develops a design methodology, supported by a comprehensive design environment and tool set, for embedded systems. Metropolis is able to devise a simulation model from the defined model in the Metropolis language [Balarin *et al.*, 2002]. This simulation model is written in Java or C++. The underlying code is specific to the Metropolis approach. In this chapter, a general-purpose simulation formalism is used, DEVS, for the simulation of deployed software-intensive systems.

In [Krause *et al.*, 2007], SystemC was evaluated as a language for modelling and performance evaluation of AUTOSAR based software. As in our approach, a lot of low-level details are taken into account for the simulation. They do not however take the plant and environment models into account while simulating the system.

Another approach is incorporating the effects of scheduling in Simulink® models [Henriksson *et al.*, 2002]. The application components are simulated in combination with

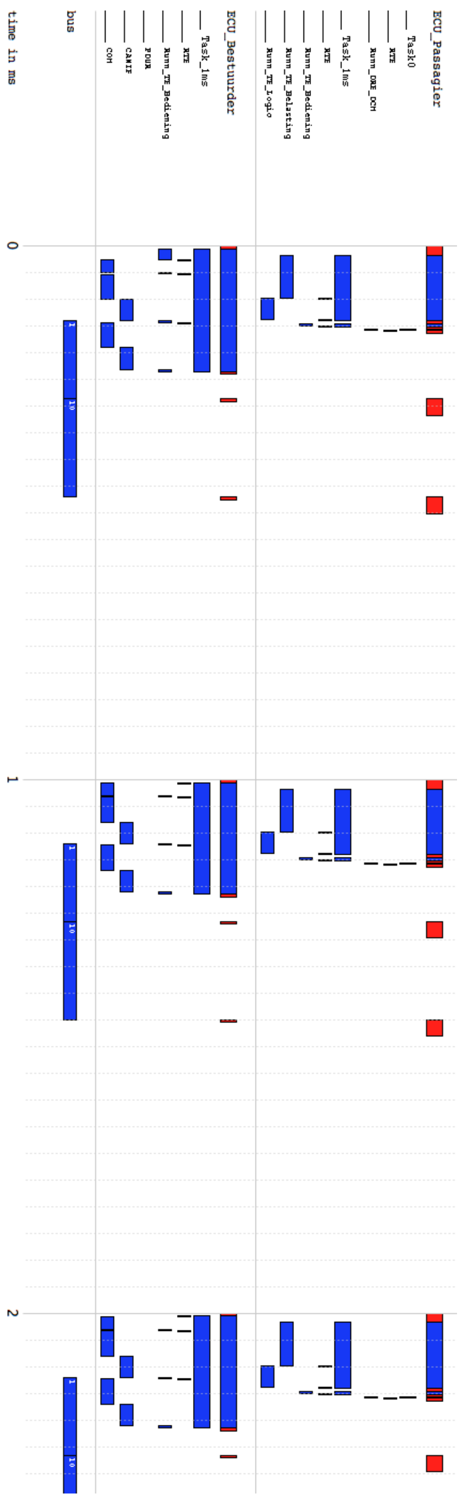


Figure 6.8: Gantt chart of the power window timing behaviour

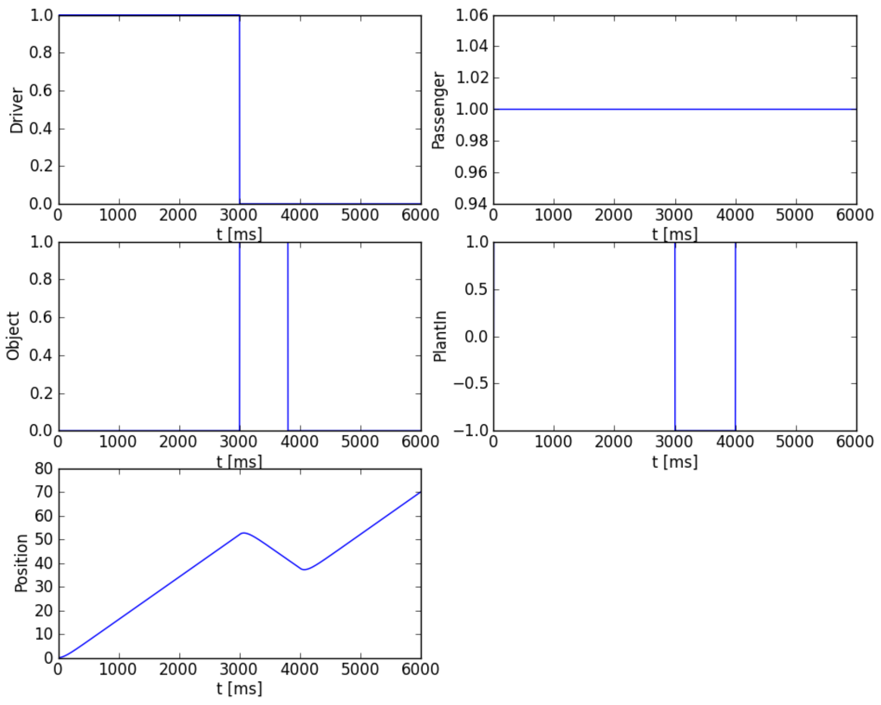


Figure 6.9: Environment and Position of the power window system during co-simulation of the DEVS model and the CBD plant and environment models

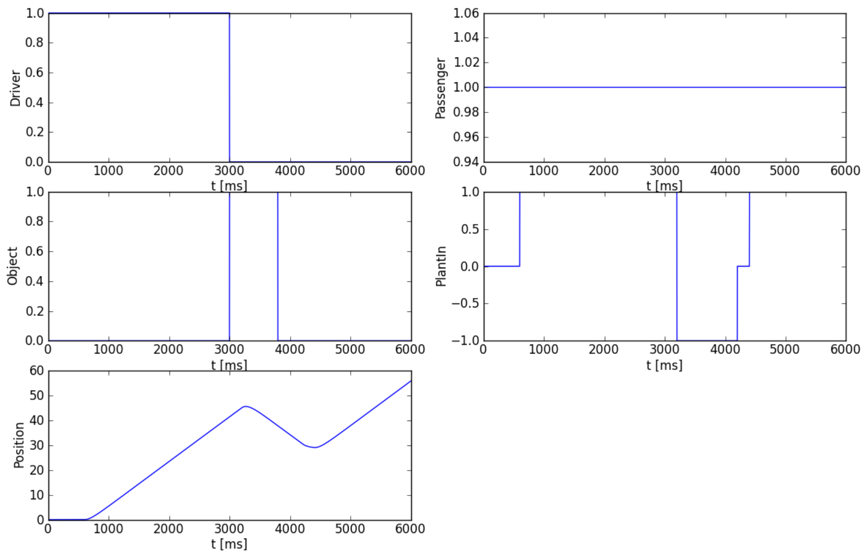


Figure 6.10: Environment and Position of the power window system during co-simulation of the DEVS model and the CBD plant and environment models using another middleware configuration

delays due to the communication hardware and the operating system scheduler. Our approach takes this a step further by simulating not only the application level, scheduler and communication bus level, but also the effects of the *configuration* of the full AUTOSAR platform in combination with the plant and environmental models in causal-block diagrams. A more complete overview of tools that support the deployment of applications on platforms can be found in [Törngren *et al.*, 2006].

6.9 Conclusions

In this chapter we have compared the properties of the DEVS formalism to the required properties for the modelling and simulation of deployment effects. It is shown that DEVS is an appropriate formalism to model the performance behaviour of AUTOSAR based systems. To support this reasoning, we constructed the simulation model of the AUTOSAR basic software and used it to simulate the behaviour of a power window system. Because of the generality of the DEVS formalism, the proposed model can be co-simulated with plant and environment models to look not only at the computational part of the system but at the the whole system behaviour.

The constructed AUTOSAR basic software model and CAN-bus model are a generic model for the performance evaluation of AUTOSAR based systems, since the models can be reused and reconfigured for other models. By using model transformation, the

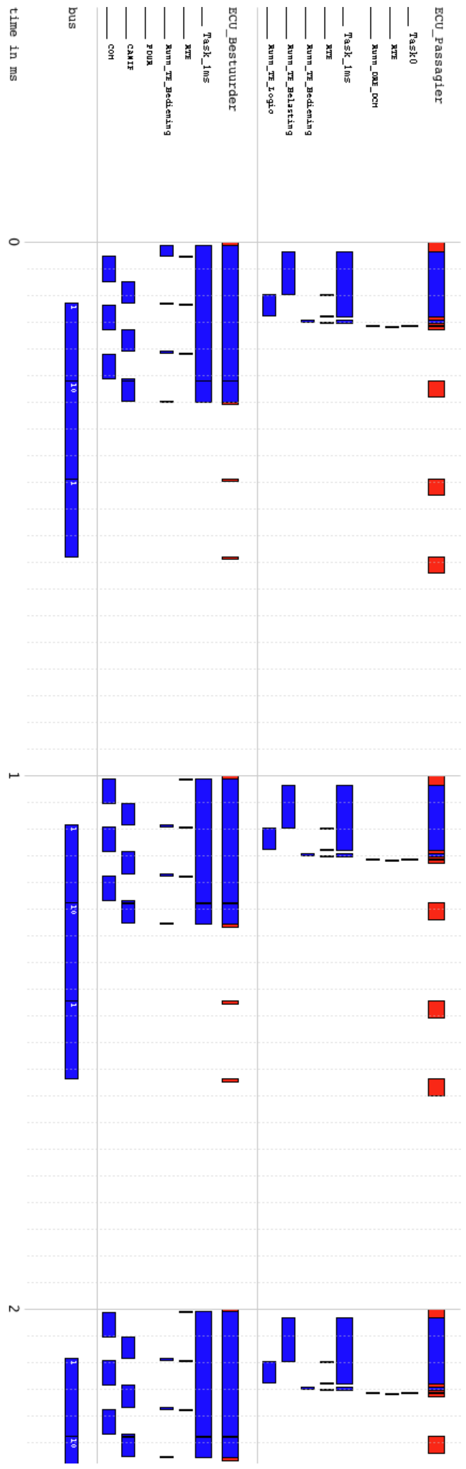


Figure 6.11: Gantt chart of the influence of low-level parameters

simulation model is automatically constructed based on the modelling artefacts of the deployment process.

The results obtained by using this simulation model will help the AUTOSAR developer to analyse the impact of his choices on the (real-time) behaviour of the system. It can be used to explore the various trade-offs while deploying automotive applications to AUTOSAR based ECUs.

Chapter 7

Calibration for System-Level Performance Models

All our knowledge has its origins in our perceptions.
— Leonardo da Vinci

7.1 Introduction

The increasing complexity of embedded systems has led to the emergence of system-level design [Keutzer *et al.*, 2000]. Engineers developing these kind of systems rely heavily on modelling and simulation to produce optimal design solutions. Models capture the behaviour and interactions of the system at higher levels of abstraction and may be used in early stages of development to perform, for example, architectural design exploration or deployment-space exploration. In recent years, system-level simulation models have been proposed to allow these kinds of explorations. Examples are Metropolis [Balarin *et al.*, 2003], Artemis [Pimentel, 2008], Palladio [Becker *et al.*, 2009] and others based on specific modelling formalisms/languages like SystemC [Hastono *et al.*, 2004] and DEVS as we described in Chapter 6.

A crucial step in the Modelling and Simulation Based Design (MSBD) process is model calibration. Setting up experiments to estimate parameters such that the model accurately reflects the implemented system structure and behaviour is technically complex and labour intensive. Parameters to be estimated are for example effective processor speed, memory consumption and network throughput of the hardware platform on which software is deployed. Model calibration requires detailed information about the system under study and its performance. This information may be obtained from datasheets, low-level simulation models, or the actual hardware. Today, the state of the art obtains the calibration parameters by instrumenting application source code to record the execution times of the different components and executing them on either the actual hardware or on a cycle-true simulation of the hardware. Calibration can be done either before or during simulation [Pimentel *et al.*, 2007]. Since a software system is composed of multiple software components, the output value of a component can be propagated to a downstream component, so it does not need to be provided explicitly. Input components however still need an input that reflects the actual operation of the system under different operational conditions.

In this chapter, we address the calibration of system-level performance models of software-intensive systems. These systems feature a tight combination of and coordination between the system's computational and physical components. For calibration of performance models of software intensive systems, the input values of some of the software components originate in the environment of the system and in the feedback loops that exist between the computational and physical components. As a consequence of the feedback, a trace-driven approach to supply these components with input signals is not desirable because of the effects of the software on the physical components and vice versa.

We demonstrate how multi-paradigm modelling allows for the synthesis of a calibration infrastructure. This includes the synthesis, from a model, of a simulator for the "environment" in which a system-to-be built will operate. This infrastructure in turn is used to obtain representative execution time distributions for the performance simulation models. Since all aspects of the system are already modelled during the design and verification phase, the model artefacts can be reused during this process.

7.2 The Power Window Case Revisited

The experiments shown in chapter 6 use a large number of calibration parameters like the worst-case execution time of a single runnable. The simulation model then takes these parameters into account while simulating the behaviour of the power window system. If the software engineer is interested in how the system responds in a normal case, these worst-case execution time parameters can be replaced with a probability distribution of the timing parameters.

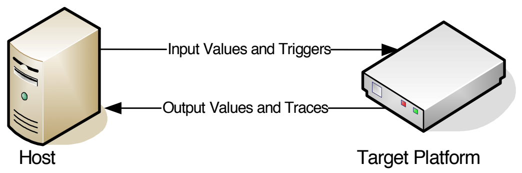


Figure 7.1: The architecture of the calibration infrastructure.

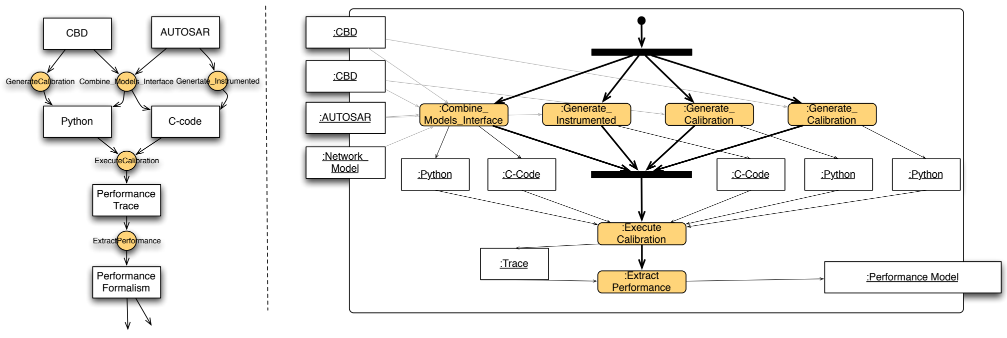


Figure 7.2: Calibration slice of the Power Window

7.3 Approach

By combining the model artefacts of the design of the system we *synthesize* a calibration infrastructure. We first take a look at the hardware components involved in the process. The architecture of the calibration infrastructure, shown in Figure 7.1 consists of:

— Host computer: The computer is responsible for simulating the plant and environment models. It provides the target platform with input values for the input software components, here the Passenger_Control, Driver_Control and the Sensor_Load components. It is also responsible for keeping everything synchronised by sending activation triggers to the target platform. The host also stores the different measurements obtained from the target platform.

— Target platform: This is the actual hardware platform the application will run on after deployment. The target platform runs the instrumented code, making execution time measurements while sending them back to the host computer. It also transmits back the output values that are needed for the plant model, in this case the direction of the motor.

The host and target platform have to communicate with each other. In this case a serial bus is used to communicate. Other types of buses could be used as well, depending on availability on both the target and host platform.

7.3.1 Outline of the synthesis process

Figure 7.2 shows the outline of the calibration infrastructure synthesis process. The transformations involved in the calibration process are:

— Generate instrumented software code: The code from the AUTOSAR application model is augmented with instrumentation code. The basic blocks in the case of AUTOSAR are the runnables.

— Combine the different models: The environment, plant and software model are manually combined into a single model. The step can also be performed automatically using the network model and the traceability links between different model elements. We however used a manual approach for obtaining the combined model. The environment blocks and plant model input and outputs are connected to the application model.

— Generate the target platform code and host interface code: The combined model is used to create the target platform and interface code to glue the host and target together. This contains the triggering of the simulation models on the host as well as on the target platform. It is responsible for sending and receiving the data between host and target platform. It also contains the code to collect the measurements on the host.

— Execute and collect traces: Finally, we execute the infrastructure and collect the measurements. It is important to note that the simulation model and controller do not run in *real-time* but in *virtual (simulated) time*. This is not a problem since the host keeps track of the time and synchronises this with the target platform. The software components under study are executed as in real-time on the target, with stimuli as in real-time, though the time between the execution of the different software components (collecting measurements, simulating a step in the physical model, communicating messages) is not. This setup is also known as processor-in-the-loop (PiL) [Mosterman *et al.*, 2011].

In the following these steps are applied to our example. All models and transformations involved are built using the AToM^3 tool. This tool is made for meta-modelling, modelling, transformation and simulation.

7.3.2 Generate Plant and Environment Simulation Models

The plant model is transformed into an executable simulation model in Python that can be executed stepwise. The simulation of CBDs on digital computers requires a discrete-time approximation. The rate of execution for this model is 1 kHz.

The same transformation was used to generate the simulation model out of the environment model. The rate of execution of this model is 2 Hz since the updating of the push on a button is much slower than the execution of the window movement.

7.3.3 Generate Instrumented Software Code

This transformation generates the instrumented source code for the runnables defined in the application model. All runnables of the model are *instrumented* to capture the execution time. In our example we use a free running timer to capture the behaviour, though other possibilities exist to get the execution time of a basic block. Before and after the function is executed, the time is recorded. The difference of both values is transmitted to the host computer. To make sure that our measurements are correct, the functions are executed in an atomic way. This is done by masking all interrupts while the time is captured.

At the same time, a run-time environment (RTE) as in AUTOSAR is generated. The RTE is used to provide communication buffers for the different signals that are communicated between the software components. It is also used to activate the different runnables in the software components at the right time. For runnables that are triggered on the sending and receiving of signals, the runnable is triggered after writing the signal in the buffer. For the runnables triggered on a timing event, the virtual time of the host computer is used. The order of execution of runnables that must be activated at the same time is determined by using the flow of signals. The topological sort algorithm is used to determine this order.

7.3.4 Combine Models and Generate Platform and Interface Code

Figure 7.3 shows the combined model of the plant, environment and software model. The tool must be able to combine models in different formalisms. The AToM^3 tool is able to combine these models using generic links. These links are used while transforming the model into the platform and interface code.

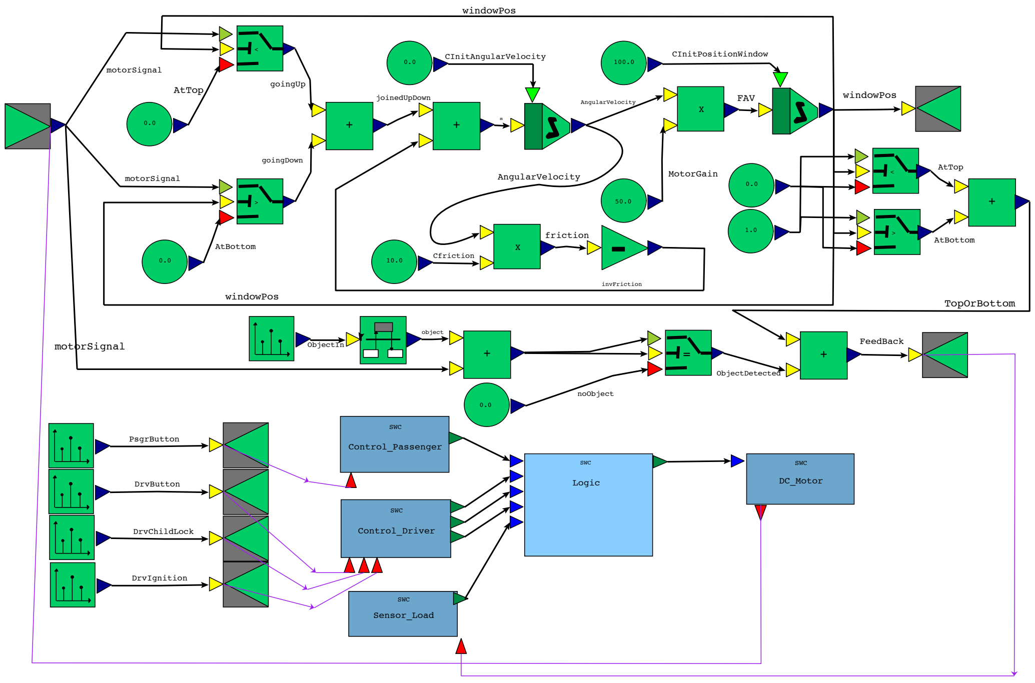


Figure 7.3: The combined model using generic links to connect entities in the different formalisms.

From the combined model we generate the interface code to connect the host and platform. Based on the rate of execution, the environment model is executed first. The values obtained from this simulation are transmitted to the platform. Then, a triggering signal is transmitted to the platform so the virtual time is incremented on the platform. The platform starts the execution of the time-triggered functions based on this virtual time. The RTE is responsible for this and makes sure that the order of execution is correct. The other runnables are activated by the sending and receiving of signals. In between the execution of the functions, the execution time sample is transmitted to the host. When all runnables are executed, the output signal values are transmitted back to the host so they can be used during the simulation of the plant model. This continues until a specified end condition is reached.

For the target platform, a small middleware, based on a template, is generated. This contains the initialisation code of the timer and the communication medium as well as functions to capture the time, send messages and receive messages. This code is very dependent on the target platform and is different for all target platforms where the calibration is needed.

7.3.5 Execute and Collect Traces

Finally the code for the target platform is compiled and loaded onto the target platform. The calibration framework is executed and the results are stored on the host computer. The obtained results can be annotated into the application models for use during the simulation of the behaviour after deployment.

7.4 Results

We generated the calibration infrastructure for deployment of the power window application on an ATMEL AT90CAN128 processor. The middleware consists of a driver for the serial interface and a free running timer of 16 bits. The timer is set at 0.125 μs per tick allowing us to measure between 0 and 8192 μs with a precision of 0.125μs. If the timer overflows, an error message is returned to the host. The timer can then be set to a wider range with less precision. This is an example of the common problem of *scaling*. We defined a small asynchronous protocol that is used to transmit the messages between the host and the platform. The protocol ensures that no messages are lost on the communication channel.

We took 15000 samples of the application running on the target platform. These are shown in Tables 7.1-7.5. The first column gives the execution time in μs. The second column depicts the number of samples with an environment model where the childlock is turned off. The last column shows the number of samples in a model where the lockout is turned on. We verified the obtained results by using hardware instrumentation with more precision. The obtained values from our generated infrastructure match the values obtained by the hardware instrumentation.

Execution Time (μs)	lockout Off	lockout On
20.375	12500	12000
19.875	2500	3000

Table 7.1: Results for the Control_Driver runnable.

Execution Time (μs)	lockout Off	lockout On
11.375	9000	10000
10.875	6000	5000

Table 7.2: Results for the Control_Passenger.

Execution Time (μs)	lockout Off	lockout On
7.625	15000	15000

Table 7.3: Results for the Sensor_Load runnable.

The obtained values from the different runnables can be used as input parameters for the system performance simulation models.

7.5 Discussion

On the tooling side of this approach a problem can occur in combining the different models. Not every tool is able to combine different meta-models as this is prerequisite to automatically devising the interface code. A possible solution to this problem is to create a super-meta-model which contains the elements of the different formalisms [Vangheluwe *et al.*, 2002].

In this chapter we focused on the calibration of a single application to be deployed on a single hardware platform type. In realistic situations multiple types of target platforms can be used when deploying such distributed applications. During the deployment space exploration the choice of platforms is made. When mixing multiple platforms during deployment multiple calibration steps are necessary. A calibration for another platform can be done by using another “middleware template”, used in Section 4 to generate the platform and interface code, that contains the low-level code to transmit and receive messages over the serial bus and perform the measurement of the execution time. The other transformations remain unchanged. This applies as well for other applications. All transformations are generic and can be used for all applications based on this workflow.

We did not discuss calibration of the middleware part in this paper. This middleware executes the runnables in tasks that are scheduled by the operating system and send

Execution Time (μs)	lockout Off	lockout On
20.000	7500	4999
20.500	0	10001
20.875	7499	0
21.375	1	0

Table 7.4: Results for the Logic runnable. The strange result of the last row is because of a special condition that can only occur in the first execution round.

Execution Time (μs)	lockout Off	lockout On
8.00	6000	3000
8.250	9000	12000

Table 7.5: Results for the DC_Motor runnable.

and receive messages. The calibration of this part can also be done by instrumenting the source code. The basic blocks of the middleware contain the system calls (like activating a task, sending a message or handling an interrupt). The functions of the middleware have to be measured for a given target platform and can then be reused for all applications running on this middleware/platform.

The described method can also be used to calibrate simulation models for other dynamic behaviour including energy consumption. For another type of calibration the synthesis transformation to generate the instrumented software has to be changed to measure another type of parameter. A sensor has to be present on the target platform to measure this parameter. This has to be accompanied with another template for the middleware that can do this type of measurements.

This chapter is just an introduction to the problem of model calibration therefore a lot of simplifications were made. We will give an overview of the different simplifications and some possible solutions that can be used to resolve these simplifications.

7.5.1 Simplifications

For simple processors, like the one used in the example, the results are deterministic, since there is a discrete number of paths in the software that can be executed. One drawback of the method is the calibration of platforms that have unpredictable components in them like such as:

— Pipelines and Branch Prediction

— Caches

— Virtual Memory Translation Buffers

— Variable Latency Instructions

— Statistic Execution Interference

All these components have an impact on the timing behaviour of the application. The most prominent effect is because of the caching mechanisms in the processors [Colin, 2003]. Empirical research has been done to identify the effects of preemption on the cache affinity of a system [Bastoni *et al.*, 2010]. While the simulation models use the results of the calibration framework, the effects should be assessed during either the analysis of the calibration results or during the simulation of the system.

In this work we assumed that the results of the calibration are valid. This is true with respect to the used environment model. Though these distributions cannot be generalised to other scenarios. For this, other environmental models need to be used and the model needs to be recalibrated with respect to the new proposed scenario. For determining the execution time bounds, the problem is more aggravated. In a simplified manner, the worst-case execution time of a software function is determined by the longest path that can be taken during execution of the function. To gain confidence whether the proposed environmental model has any chance of taking one of the longest-paths, some coverage criteria can be used. The coverage criteria states how much a function is exercised by a test suite. For coverage at the source code level, a lot of criteria exist for example, Modified Condition/ Decision Coverage and its variants [Chilenski, 2001] are used in aerospace. These coverage metrics are extended to the model level by the model-driven testing community for example in [Mcquillan and Power, 2005; de Souza *et al.*, 2000]. A lot of work has been done for generating adequate testing infrastructure based on models, for example in [KansomKeat *et al.*, 2008; Offutt, 1999; Pasareanu *et al.*, 2009; Chevalley and Thévenod-Fosse, 2001]. These techniques could improve the bounds but are not suitable since these criteria are developed for functional testing. Other coverage criteria, designed for the purpose of Measurement-Based Timing Analysis (MBTA), like Balanced Path Generation I and II [Bunte *et al.*, 2011], could be used instead. These criteria should be translated to the modelling level for generation of environment models.

Finally, in this work it is assumed that all components and models are available as white-boxes to the developers. In organisations this is usually not the case where components are bought as *Commercial Of-The-Shelf* (COTS) components that are black-box. This has been investigated in [Perrone *et al.*, 2008].

7.6 Related Work

Calibration of system-level performance simulation models has not been widely addressed yet. The work of Pimentel et al. focusses on calibration for embedded multimedia applications [Pimentel *et al.*, 2007]. However, it does not take input values of the input components into account nor does it have any feedback from a physical component. Other related work is found in two areas: (a) determining the worst-case execution time of a function and (b) software performance modelling and model-based testing.

In the real-time systems community, a lot of research effort has been devoted to the problem of worst-case execution times (WCET). This problem can be described as determining the upper bounds on execution times of the software components running on a hardware platform. This is not a trivial problem since modern processor architectures contain interacting components such as caches, pipelines and branch prediction. Two approaches are currently used to obtain this bound: (a) static analysis and (b) measurement based.

— Static methods: These methods do not rely on the real hardware but use models of the hardware platform combined with the source code. The source code of the software is parsed and combined with a model of the processor to determine the upper bounds of the execution time. Analysis methods contain different steps for control flow analysis, processor behaviour and bound calculation. A lot of commercial and academic tools are available to do the static analysis, for example: `Absint aiT` [Ferdinand, 2004] and `SWEET` [Engblom *et al.*, 2003].

— Measurement based methods: These kind of methods execute the task or part of the task on the real hardware or an a low-level simulation model with specific test programs. The measurement can be performed by collecting timestamps from the processor by instrumenting the code. Mixed HW/SW solutions can also be used to collect the timings from the lightweight instrumentation code. HW tracing mechanisms are available but do not always produce correct timings. For example, NEXUS buffers its output while a timestamp is given when the event leaves the buffer. Finally one could use a cycle accurate simulator to replace the hardware. An academic tool, `pWCET` [Bernat *et al.*, 2003] is available to instrument the code and calculate a bound.

Kirner et al. derive a bound from Matlab®/Simulink® models by instrumenting the Simulink blocks to produce timing information [Kirner *et al.*, 2002]. This timing information is back-annotated in the model. A more detailed overview of the tools and methods involved in obtaining the worst-case execution time can be found in [Wilhelm *et al.*, 2008]. We are not only interested in the worst-case execution time but also in the distribution of the execution time. The `pWCET` tool can provide us with this, though the input values for the program have to be specified at the source code level. Our work is closely related to the measurement-based methods for obtaining this execution bound. Though in none of these cases, the physical part is taken into account since they are only interested in the worst-case bounds of the application. Our work is interested in obtaining distributions together with a reasonable bound for the execution times. The techniques proposed by the measurement-based timing analysis community to increase the confidence in the obtained bounds can also be applied to this work. Examples of these techniques can be found in [Deverge and Puaut, 2005; Bunte *et al.*, 2011]. Our technique however can be extended to include measurements for other types of metrics that can be derived from the dynamic behaviour of the application.

Measurement is often used by performance engineers to calibrate, validate and optimize systems. Instrumentation is key to obtaining these results. Our approach instruments the software functions when generating the source code. Other research has been devoted to the correct and lightweight instrumentation of existing code. Fishmeister and Lam for example developed an instrumentation method for real-time embedded systems that ensures that no deadlines are violated during the execution of the instrumented software [Fischmeister and Lam, 2010]. Others have focused on the automatic instrumentation of software using for example aspect-oriented programming [Debusmann and Geihs, 2003]. Other research in this area, close to our contribution, is to determine the workload under which the performance tests run. Garousi, Briand and Labiche used UML models to generate stress tests for networks in distributed systems [Garousi *et al.*, 2006].

The calibration framework presented here resembles the processor-in-the-loop (PiL) approach for testing systems. For example, the PiL-approaches by MathWorks®[MathWorks, 2013] and dSpace [dSpace, 2013] can also be used to measure execution times of the different runnables. Our approach is however more flexible, because the transformations are open to generating a calibration infrastructure of different parameters. The transformations can also be changed so other formalisms can be used. While the infrastructure resembles our setup, the purpose is very different. PiL is used to test the developed control system or calibrate the parameters of the control model. The performance simulation model on the other hand is used to explore the deployment space before the actual deployment. Still, the infrastructure used in the testing procedure could be adapted for the purpose of calibration.

7.7 Conclusion

In this chapter we addressed the calibration of system-level performance models of software intensive systems. To obtain these calibration parameters we presented a method to automatically generate a calibration infrastructure. The infrastructure is generated using a multi-paradigm modelling approach based on the model artefacts developed during the development and analysis of the system. The plant and environment models are simulated on a host computer that gives the inputs for the instrumented application software running on the target platform. The target feeds back information to the plant model to close the control loop, while returning the measurements. The contribution is demonstrated by applying it to obtaining calibration parameters for a system-level performance model of the automotive power window.

Chapter 8

Automatic Design Space Exploration

Exploration is really the essence of the human spirit.
— Frank Borman

8.1 Introduction

Software intensive systems often need to comply with particular requirements such as real-time execution deadlines, reliability and low power consumption. This, while hardware and software platform resources are often limited in performance, memory, number of hardware interfaces, communication bandwidth, time to market, and so on, mainly for cost reasons. Building software applications taking these restricted resources into account is challenging. The software development process for such systems must consider these platform restrictions. During application design, the developer needs to make deployment choices respecting the available platform resources, even optimizing their usage.

This chapter explores the use of model transformation during the design process, with the goal of (semi-)automatic deployment space exploration. During the deployment process, models are transformed from a high abstraction level down to a complete realization. For deployment optimization, intermediate transformations are often necessary. To guide the search for an optimal deployment, models are transformed into simulation or analysis models that allow estimation of the impact of certain resource limitations on a candidate deployment.

The use of the MPM-approach to design space exploration allows for explicit optimisation models tailored to the problem at hand. Model transformations can also encode domain knowledge by adding extra constraints in the transformation rules. The use of this MPM-approach integrates design space exploration in the MPM development cycle. This results in a uniform process where the transformation models can be reused for similar problems, documentation, etc.

8.2 The Power Window Case Revisited

In the power window case study, we are interested in feasible solutions that meet the required end-to-end latency. This means that the window has to react within 200 ms. As an example optimisation step, we also want to optimise this end-to-end latency, so an optimal solution is found (that is, the fastest solution possible, so the control algorithm reacts as fast as possible). We need to make choices of how the software components are distributed over the network as well as how to configure the real-time operating system and communication stack.

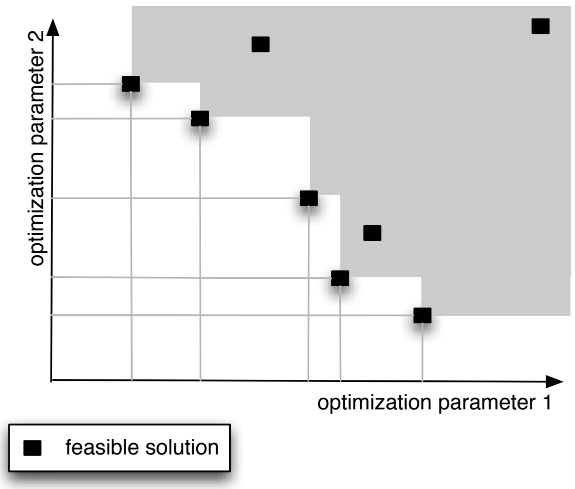


Figure 8.1: An example Pareto front

In realistic systems this is of course an oversimplification. Besides the feasibility of the solution, system integrators make extreme efforts to reduce the memory consumption of the system, the required number of ECUs, the power consumption, etc. This results in trade-off situations where choices that favour a particular parameter, have an adverse effect on another parameter. This results in a set of solutions whereby a set of these solution points are *Pareto efficient*. A Pareto efficient solution point is where one parameter can not be made better, without making another parameter worse. In optimisation usually, this comes down to minimising a set of parameters resulting in a Pareto-front, shown in

Figure 8.1.

8.3 Introducing Multiple Levels of Abstraction/Approximation

Our approach, shown in Figure 8.2, is based on the key concepts of Multi-Paradigm Modelling (MPM): Model everything at the most appropriate level(s) of abstraction, using the most appropriate formalism(s) [Vangheluwe *et al.*, 2002].

Figure 8.2 shows the introduction of different abstraction/approximation levels. The start model at the top is refined. This results in a set of solutions (the set cardinality is depending on the design choices that are available). These solutions are transformed to a simulation or analysis model where the relevant properties can be checked. Bad solutions are pruned out, shown in red. The other, shown in green, are further explored.

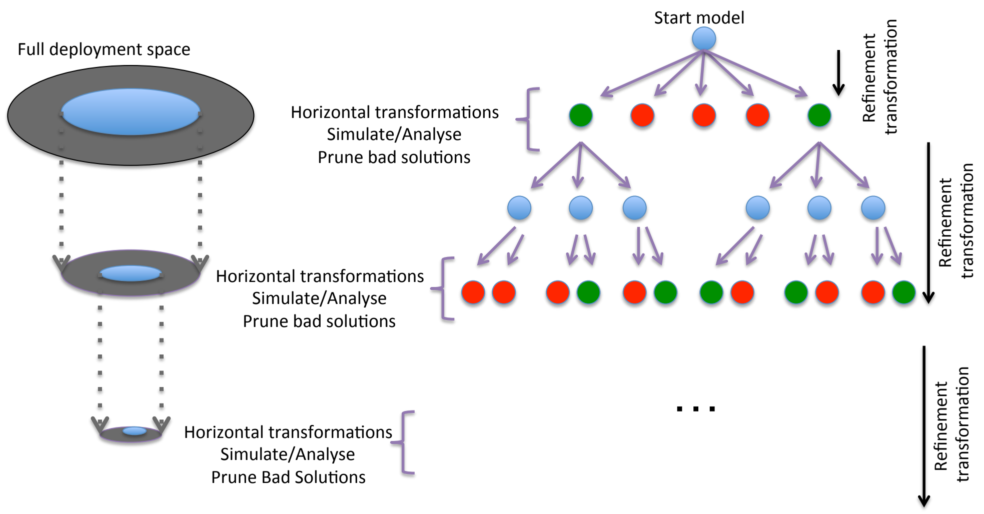

Full deployment space
Start model
Horizontal transformations
Simulate/Analyse
Prune bad solutions
Refinement transformation
Horizontal transformations
Simulate/Analyse
Prune bad solutions
Refinement transformation
Horizontal transformations
Simulate/Analyse
Prune Bad Solutions
...
Refinement transformation

Figure 8.2: Automatic Deployment Space Exploration Approach.

Two types of transformations are used in our approach:

— Refinement Transformations: Deployment can be viewed as a set of transformations since we iteratively add knowledge about the platform to the model. By using these transformations we can generate solutions allowed within the syntactic constraints imposed by the meta-model. Other constraints on the solutions can be encoded in the transformations as well, using the negative application condition (NAC) and other constraints. Every transformation may yield multiple solutions, which causes an explosion of the search space. Therefore, it is important that non-conforming solutions are pruned as early as possible. To achieve this, multiple layers of abstraction are defined where high-level estimators can be found to check the generated partial solutions.

— Horizontal Transformations: The model is transformed to another formalism at the same abstraction level making it amenable to evaluation of its suitability using simulation or analytical techniques.

The solution space is reduced with every abstraction layer since the pruned branches are not further explored. The choice of analysis method needs to match the system properties that are optimized.

It is important to note that the evaluation cost increases with every refinement step. Simple analysis models usually use less computational power than detailed analysis. Even worse, very detailed, low level simulations need a lot of computational power.

8.4 Introducing Search Techniques

In this section we introduce model transformation techniques and discuss how general search techniques can be used within model transformations. While it is out of scope of this dissertation to give a complete overview of these techniques, we show that by using an expressive transformation language, meta-heuristics can be implemented in the transformation models.

8.4.1 Model Transformation Languages and T-Core

The developed transformation language is based on the T-core transformation framework that allows the construction of custom transformation languages [Syriani and Vangheluwe, 2010]. Figure 8.3 shows some of the components of a transformation language. We briefly discuss the components used in this work. More information can be found in [Syriani and Vangheluwe, 2010].

— Operators:

  — Matcher: The matcher finds the matches of the LHS condition in the model and stores them in a match-set.

  — Iterator: The iterator is used to select one match to rewrite.

  — Rewriter: The rewriter rewrites the model using the RHS pattern.

  — Rollbacker: The rollbacker enables backtracking in the transformation language by creating a checkpoint.

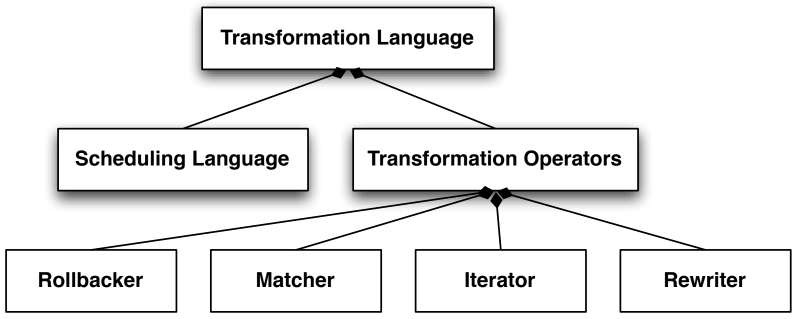


Figure 8.3: Composition of a Transformation Language

— Scheduling Language: The scheduling language is used to schedule the different rules one after another. Different kinds of scheduling languages can be used. In this work we use the Python programming language as our scheduling language.

In the following subsections we show how to implement, using the T-Core Transformation Framework, three well known search techniques that are used in optimization. The first two: *exhaustive* and *random* search create a number of solution points in the search space. The latter: *Hill Climbing* starts optimizing a single solution.

8.4.2 Exhaustive Search

While the exhaustive search is infeasible in most problems, it can be used for the optimisation of small problems. Exhaustive search will generate all solutions of the design space. Figure 8.4 shows an activity diagram of the implementation of the exhaustive search method. The transformation starts by matching all the occurrences in the start model. The iterator chooses the first match in the match-set. At this point a checkpoint is made. This checkpoint contains (a) the model, (b) the match-set, without the chosen match and (c) the selected match. The selected match is rewritten in the model. If more rules are available or the same rule has to be executed multiple times, this is done by using the same method (match, select match, checkpoint and rewrite). The complete solution is archived for further analysis. Then backtracking can start. Since the backtracker can contain multiple checkpoints (from multiple rule applications), the last one is selected. This restores (a) the model, (b) the match-set and (c) the match that was selected at the time instant the checkpoint was made. The iterator is used to replace the previously chosen match with another one from the match-set, again this is check-pointed and rewritten. The process continues as described above until all matches in the match-sets of all checkpoints have been applied.

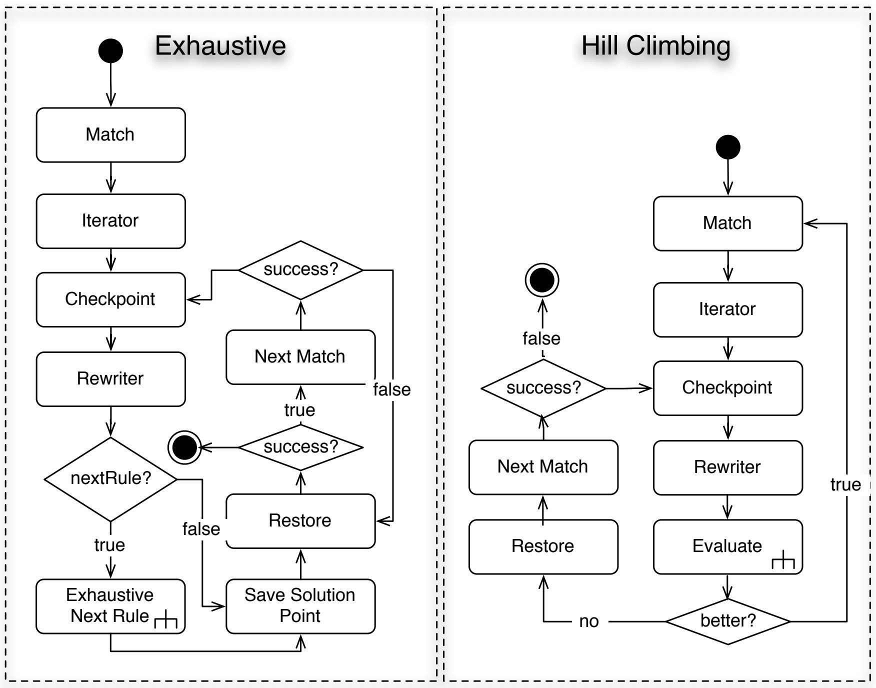


Figure 8.4: Exhaustive Search and Hill climbing

8.4.3 Random Search

In random search a set of solutions is created in a random way. Random search uses only the matcher, iterator and rewriter. After matching all occurrences of the pattern in the model, a random match is selected for rewrite. This requires a different iterator than in the exhaustive case. This is because the normal iterator is deterministic and always chooses the same match. The rewriter applies the randomly chosen match on the model. Another rule, or the same rule can be executed after that until a solution point is obtained. A loop is used to create a set of solutions.

8.4.4 Hill Climbing

Hill climbing is a local search technique that uses an incremental method to optimize a single solution. It examines neighbouring states and accepts the change if it is a better solution with respect to the goal functions. Figure 8.4 shows the building blocks of the Hill Climbing transformation. After matching a (set of) rule(s), the iterator picks one match at random and rewrites this in the model. The solution is evaluated and compared with the original solution. In case the solution is not better, the original solution (with the matches) is restored and another match is randomly selected and evaluated. If the solution is a better one, it is accepted. The evaluator contains a set of transformation rules to calculate the metrics of the solution or to generate an analysis or simulation model that can be executed. The metrics obtained are given back to the hill climbing solution so a decision can be made. When a better solution has been found, the process is restarted until no more improvements can be found.

8.5 Combining Transformations

The solution has to combine different transformations, search-based transformations and Model-to-Model transformations, in sequence or in parallel to optimize a system. We leverage a number of techniques to alleviate the state space explosion problem during the optimization of a system:

— *Different levels of abstraction or approximation:* As already discussed in section 8.3, different levels of abstraction/approximation are needed to reduce the search space.

— *Other optimization techniques:* When a general solution method is already available, for example through the capabilities of standard tool, we transform the model to this representation. The results are transformed back to the original representation for further exploration or synthesis activities.

— *Manual activities:* When the designer has a solution (manually or using external tools) without an available automatic transformation, the designer can manually change the model. The exploration activity resumes from this point.

In this work the Formalism Transformation Graph and Process Model (FTG+PM), proposed in chapter 3, is used as the base for chaining different transformations. The framework can be used to incorporate all the different steps of the MDE lifecycle and thus allows the embedding of the exploration activity in the design and verification of complex systems.

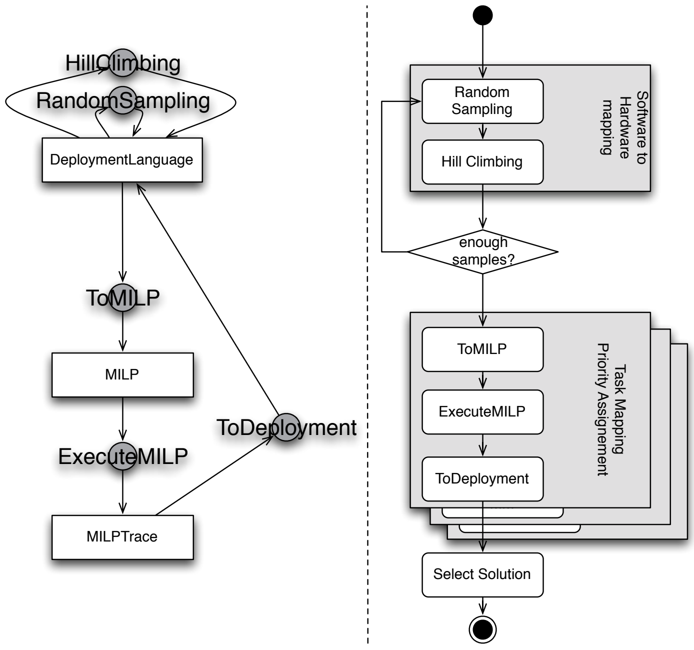


Figure 8.5: An example FTG+PM for design space exploration

Figure 8.5 shows an example FTG+PM for the purpose of design space exploration. On the left side of the figure, all languages involved in the optimization chain are displayed. We only present three languages: (a) The deployment language where the design space should be explored, (b) The MILP language, representing the model in a Mixed Integer Linear Program, and (c) The MILPTrace language, representing the results produced by the MILP solver. At the FTG level, five transformations are defined: (a) Random Sampling, (b) Hill Climbing, (c) ToMILP, (d) Execute MILP and (e) ToDeployment. For each of these transformations the input and output languages are defined. On the right side of the figure these transformations are scheduled one after another in the process. The optimization chain starts with the Random sampling of a number of solutions. These solutions are further explored using an instance of the hill climbing transformation. Later, the intermediate results are stored and this process repeats until a fixed number of solutions are found. At another approximation level, a set of these solutions are transformed to a MILP representation. This MILP representation is solved using standard solvers such as CPLEX. Finally, the traces are translated to the original representation.

8.6 The Power Window Case Study

The power window however is not the ideal case study to test this approach. The model is too small to evaluate the benefits of a transformation based approach. However, we will use the power window exemplar for validation purpose. Because of the small size of the exemplar, it can be exhaustively searched for the optimal solutions. Other optimisation techniques, can be applied as well and compared with the optimal solution.

In the power window exemplar, we are interested whether the system can meet the performance requirements. For the deployment exploration in the context of real-time behaviour we identified three approximation levels. The levels are pragmatically chosen so they correspond to the analytical and simulation methods currently available in the automotive industry. For each of these levels we describe what information has to be available and what method is used to obtain the high-level estimator.

The first defined level approximates the *system architecture*. Here, the software components of the application are mapped to a specific hardware component. In case of multiple components, a bus connects these components. Since all information about timing of the triggers and execution time is known at this point, we can use a simple bin packing check to ensure that no single component or bus is overused at this level of approximation. This means that no single hardware component has a utility of 100%.

The next approximation level concentrates on the *services* provided by the communication stack and operating system. The runnables are grouped into tasks and they are given a priority. The same is done for the signals that are to be transmitted on the bus. These are grouped into messages and given a priority for the arbitration on the bus. Furthermore, signals and messages are given their transmission mode and property. At this abstraction level all information for schedulability analysis is present. This means that solutions that cannot meet their end-to-end deadlines are automatically pruned.

Finally, the application is fully mapped to the hardware platform by defining *hardware buffers* for the reception and transmission of messages. The drivers and interfaces of the communication stack are configured and software buffers are defined. Some hardware-specific options are also configured. The solutions are checked for their real-time behaviour under the influence of buffering. We use a DEVS simulation model presented in Chapter 6 for this purpose. Solutions that have lost messages and/or that cannot meet their end-to-end deadlines under the influence of buffering are pruned in this step.

The start model is shown in Figure 8.6. The model contains five software components (blue boxes), each contain a single runnable (green box with F(x)). The components communicate a signal (red circle) using ports (blue and green triangles).The software components that use hardware, like sensors and actuators, are pre-mapped to an ECU (green box) before exploration. This is due to spatial constraints of the system. The source components read out sensor values at a rate of 1ms. The logic component starts execution when the sensor value of the object is received. The motor actuator executes when a signal is received from the logic component. The application can be deployed on two

ECUs that communicate via a CAN-bus at 500 kbit/s. Both ECUs already have empty configuration modules for the RTOS, COM and CANIF module (grey boxes).

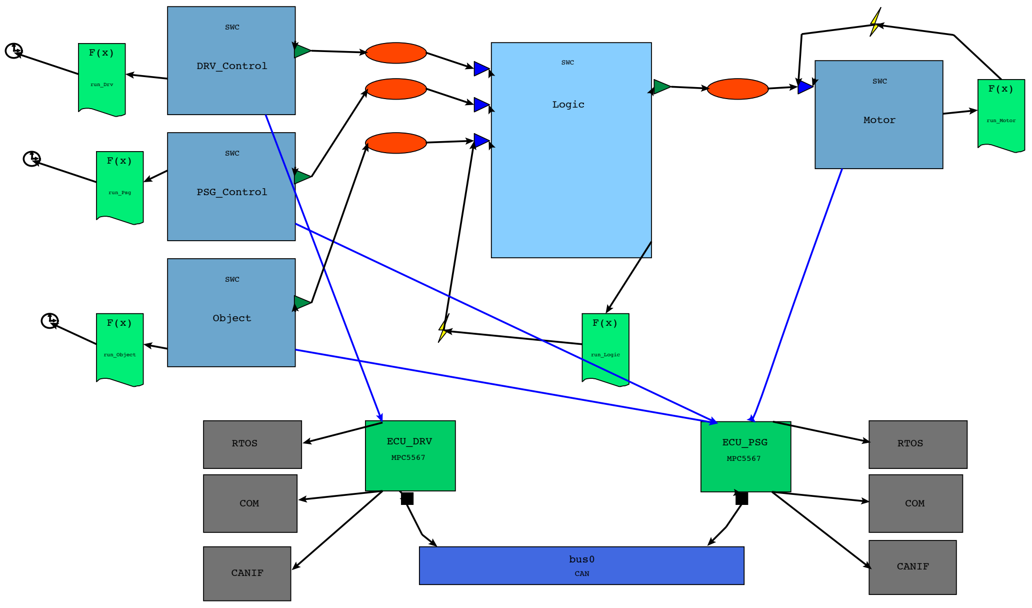


Figure 8.6: The power window model for exploration

We give a small overview of the different transformations and results involved in the exhaustive search of the deployment space of the power window exemplar.

8.6.1 Architecture level

— SWC2ECU: This transformation assigns an unmapped software component to an ECU. The transformation is composed of two rules. The first rule assigns a pivot to an unmapped software component. This pivot is used by the second rule to map the component. Intuitively, we select a component and match the ECUs where we can deploy it. From the match set, a match is selected and a checkpoint is made. This process continues until all software components are mapped. Then backtracking starts. The checkpoint takes the next ECU to deploy the component. A checkpoint is again made. The first process of matching an empty SWC, mapping and checkpointing starts again until again all software components are mapped. The whole process of backtracking, rematching and checkpointing continues until all solutions are generated. If the pivot is not used, similar solutions are created. The rule is shown in Figure 8.7.

— Mapping2BinPack: A bin packing function is created that can be used to check the feasibility of the solution. If the solution is not feasible, the full branch can be pruned.

Two solutions are created after this approximation level, since only a single software component can be mapped freely. Listing 8.1 shows the bin packing check of a single

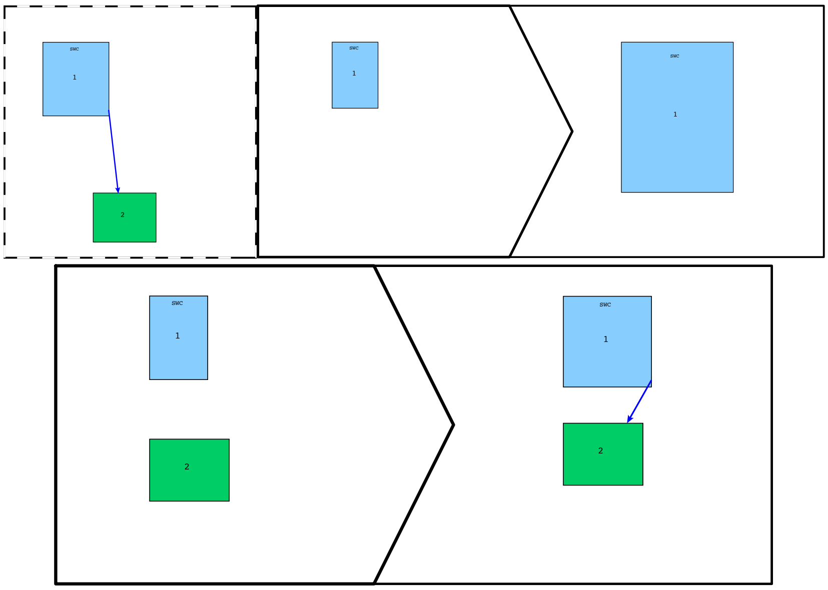


Figure 8.7: Transformation to map a software component to an ECU

solution. Both solutions are feasible and can be further explored.

Code Listing 8.1: "The bin packing check of a single solution"

```
def calc():
    ECU_PSG = 0
    ECU_DRV = 0
    ECU_PSG += (0.011375 /  1)
    ECU_PSG += (0.007625 /  1)
    ECU_PSG += (0.00825 /  1)
    ECU_DRV += (0.020375 /  1)
    ECU_DRV += (0.021375 /  1)
    if (ECU_PSG < 1 and ECU_DRV < 1):
        return True
    return False
```

8.6.2 Service level

— Run2Task: We defined two extra constraints on the mapping of runnables to tasks. In our case, each runnable is mapped to a single task and each task has a different priority. This is an added constraint with respect to the AUTOSAR standard, where multiple runnables can be mapped to a single task and tasks can have the same priority. The rules should be adapted to be conform with the AUTOSAR standard. The transformation consists of four rules. The first rules creates an amount of tasks (purple boxes) equal to the number of runnables in the system. Each task has a unique priority. From this point on, the mapping is done per ECU by first selecting an ECU (assign a pivot), where the mapping is not yet done. Afterwards the runnables of that ECU are mapped to the tasks on that ECU. The same principle as with the SWC2ECU transformation applies, by using a pivot on the runnable. The set of models from a single ECU are used to map the next ECU since the combinations are needed for the exhaustive case. Figure 8.8 shows the transformations involved.

— signal2IPDU: This transformation packs signals to an IPDU that can be transmitted on the CAN-bus. Each IPDU is assigned a unique priority. Signals transmitted from the same ECU can be packed together in a single IPDU. We only take the *Direct* transmission mode of an IPDU into consideration. The transformations are very similar to the run2task transformation, with the exception that IPDUs can be shared. For this, two extra rules are used. The negative application condition, that makes sure that an IPDU has only a single signal, is removed. The first extra rule, deletes an IPDU, if more than one IPDU is present on the ECU. The second rule removes solutions where an empty IPDU is present in the model. These rules are combined to make sure that no redundant solutions are present after the deployment of the second approximation level.

— Mapping2sched: This transformation creates a csv-file that can be interpreted by a timing analysis tool from the aXbench project[1]. The tool calculates the worst-case response time of the task.

Code Listing 8.2: "Output of the Mapping2WCDOPS transformation"

```
Task;TE_run_DRV_Control;;0;0;0;0;0;10;101;0;false;TT
Task;run_DRV_CONTROL;;0,020375;0;0;0;0;10;101;1;false;PRED
Task;IPDU_10;;0,124;0;0;0;0;10;201;2;false;PRED
Task;TE_run_PSG_CONTROL;;0;0;0;0;0;20;102;0;false;TT
Task;run_PSG_CONTROL;;0,011375;0;0;0;0;20;102;4;false;PRED
Task;TE_run_Object;;0;0;0;0;0;30;102;0;false;TT
Task;run_objectL;;0,007625;0;0;0;0;40;102;6;false;PRED
Task;run_Logic;;0,21375;0;0;0;0;50;102;7;false;PRED
Task;run_Bediening;;0,00825;0;0;0;0;60;102;8;false;PRED
```

The task creation transformation generates 36 unique solutions from the two previous solution. In 24 cases, only a single signal has to be transmitted, resulting in only 24 solutions after the signal packing transformation. The other 12 solutions each generate 2 different solutions, one with a single transmission IPDU on each ECU, and one with two transmission IPDUs on the ECU_PSG and a single IPDU on the ECU_DRV. The definition of the signal transmission property is applied on the solutions where two signals are packed together. This results in a total of 60 solutions at the second approximation level. An example of a csv-file can be seen in Listing 8.2. All solutions are schedulable within the bounds of the application.

[1]http://www.axbench.de

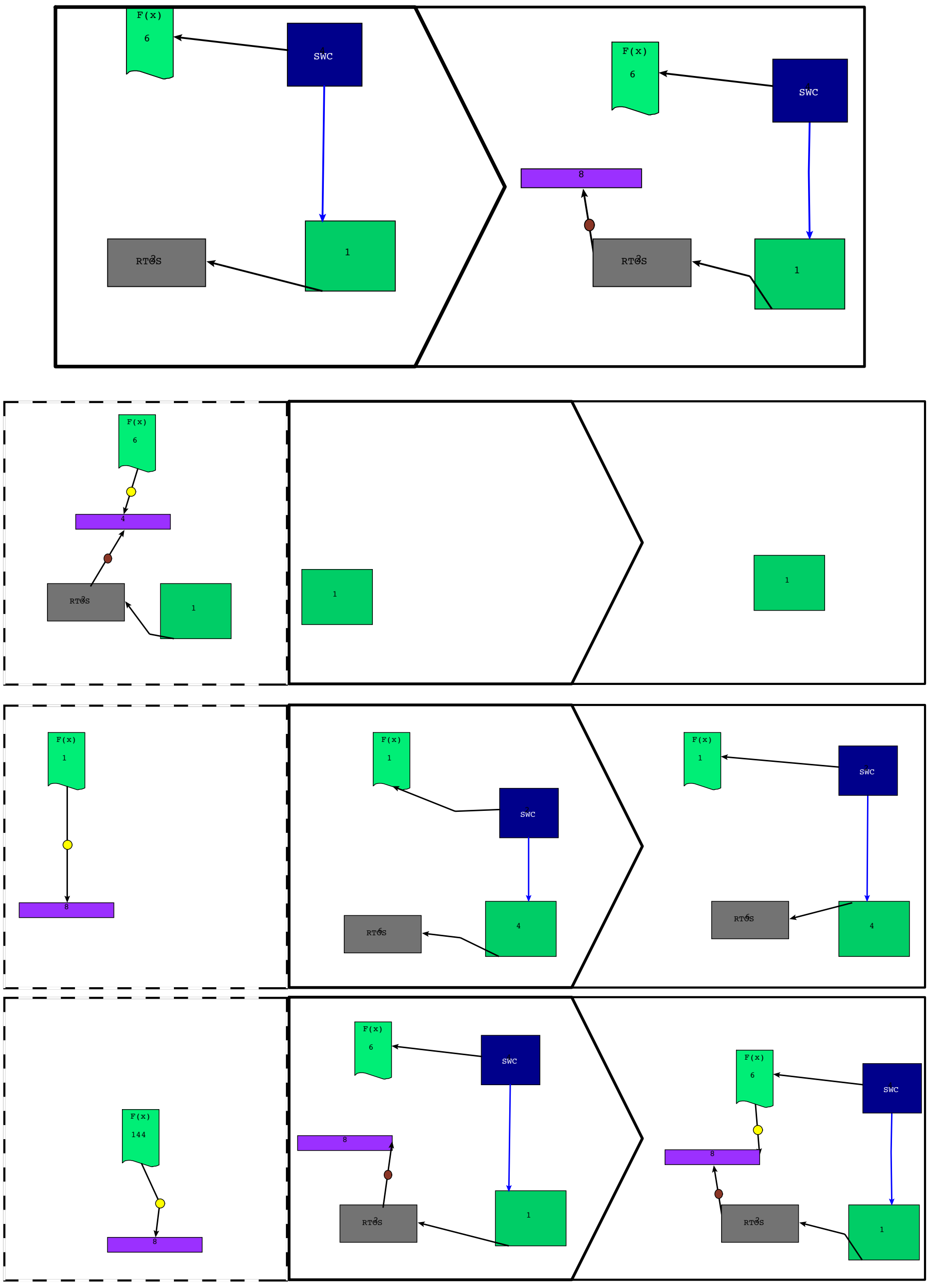


Figure 8.8: Mapping of a runnables to tasks

8.6.3 Full deployment level

The last approximation level is generated using another transformation language, namely the *Janus Transformation Language* [Cicchetti *et al.*, 2011]. JTL is a bidirectional model transformation language based on Answer-Set Programming (ASP) [Gelfond and Lifschitz, 1988]. The transformation engine is based on a generation mechanism that first expands the set of possible solutions based on mapping rules and then refines such a set by applying constraints on the derived target models, such as meta-model conformance rules and additional desired characteristics

The transformation in JTL creates all the solutions with the following rules:

— CreateBuffers: The number of hardware buffers in the CAN controller are bounded. Therefore they must be distributed between transmission and reception buffers. Once they are partitioned, the frames need to be assigned to a hardware buffer. Hardware buffers can be overwritten if a message has not been transmitted yet.

— EnableSWBuffers: This transformation rule enables or disables the use of software buffers in the CAN interface basic software module. If software buffering is allowed, the message can be stored in software when the hardware buffer is full.

— EnableCancellation: The transformation rule enables or disables the transmit cancellation flag. This controls the cancellation of a message inside the CAN hardware buffer when a higher priority message is available.

— EnableMultiplexing: The CAN hardware normally checks the buffers in a linear fashion, transmitting the first buffer that is not empty. By allowing multiplexing, the buffers of CAN become a priority queue. It will always transmit the highest priority message first. The transformation rule enables or disables this flag.

— Sol2Simulation: This transforms a single solution point to the simulation model described in Chapter 6.

Given the number of variables and the corresponding possible evaluations, the execution yields between 64 (only transmitting a single message or two messages) and 192 solutions (transmitting three messages) for a single solution point from the previous approximation level. However, in 24 cases of the previous approximation level, there is only one ECU transmitting IPDUs. Since nothing is transmitted on the other ECU, some of the flags become obsolete, such as, the CAN cancellation, multiplexing, and software buffering. Therefore, the transformation specification has been enriched with an extra constraint. When the source model contains an ECU transmitting no frames (i.e., the Passenger), the solution space has been reduced from 24 to 8 possible solutions.

The total amount of solutions for the power window exemplar is 6240 solutions. Since simulation of all these solutions is impractical, 24 selected solutions are simulated. Simulation of these solutions yield that most are feasible. In some cases though, messages are lost because of a shared buffer and no software buffering, resulting in non-feasible solutions that do not adhere to the proposed requirements.

8.7 Industrial Size Model

We show the use of search techniques and heterogeneous optimisation techniques using another automotive case study, based on [Zheng *et al.*, 2007], where a set of software functions need to be assigned to a set of electronic control units (ECU). The designer can select the types of the ECUs. The software functions are further executed on a real-time operating system with a fixed priority preemptive scheduler, where the priorities of the tasks need to be selected. The communication signals need to be packed into communication frames on a communication bus, with fixed-priority non-preemptive scheduler (like CAN). The solutions are constrained by typical end-to-end deadlines. A part of the meta-model for this deployment language is shown in Figure 8.9.

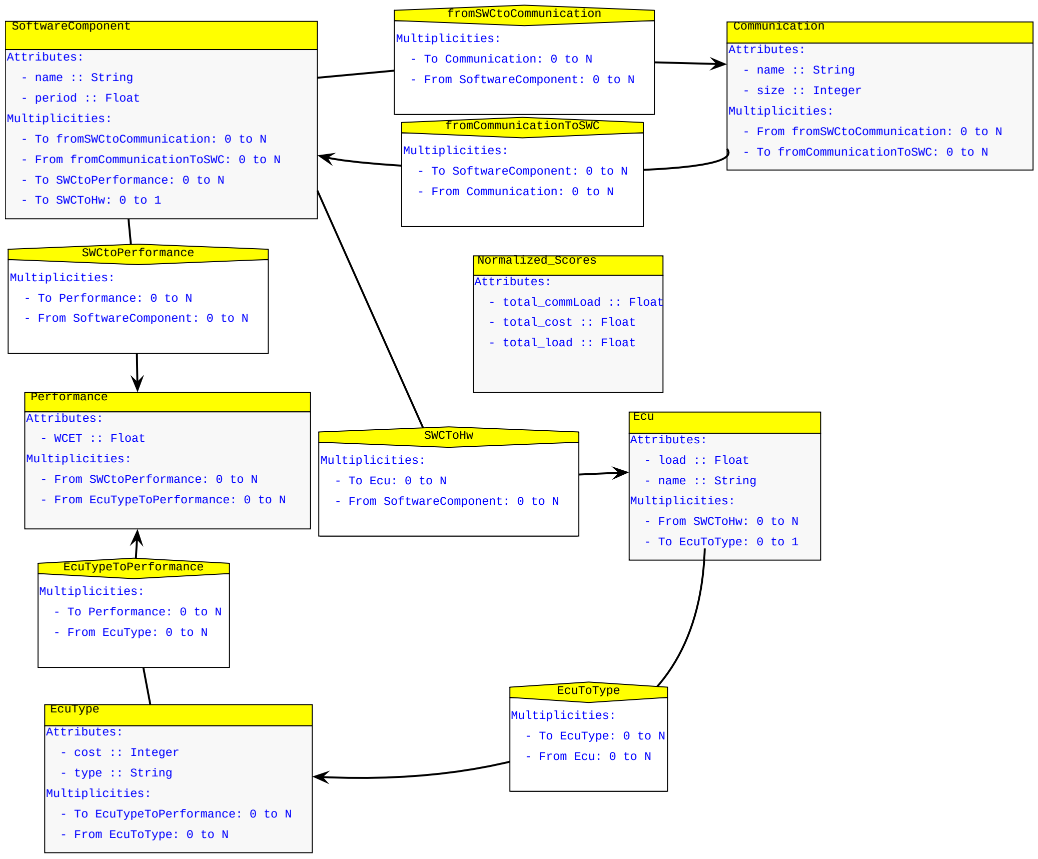


Figure 8.9: Deployment metamodel of the industrial size case study (partially)

The industrial size design consists of 40 software functions, 81 signals, and 9 ECUs (each has two types to choose from). Because of the size of the model, it is not useful to show the full model. A small part is shown in Figure 8.10. The blue rectangles represent the software functions. Each function has a period, defining the rate of execution of the software function. They are connected with each other using a purple circle, representing the data transferred between the software functions. An ECU is represented by a green box. Types of ECUs are white boxes with a capital *T* inside. The performance properties of a software functions are shown using a scope like icon. They connect to both an ECU type and a software function.

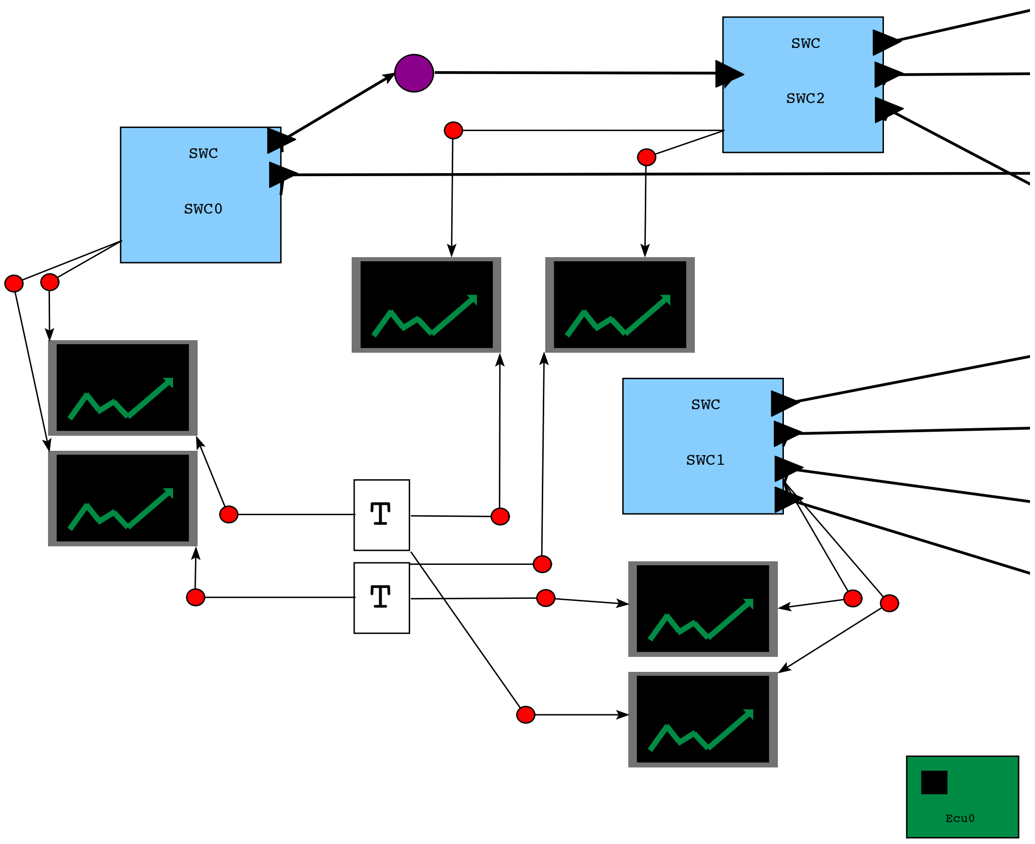


Figure 8.10: Part of the model used in the industrial case study

— **Architecture Level:** At the architecture level, we assign the different software functions to an ECU and assign a type to the ECUs. We can calculate the classical schedulability test, defined by Liu and Layland [Liu and Layland, 1973], to prune infeasible solutions. The schedulability test however also prunes branches that could produce an optimal solution. Because of the size of the model, the schedulability test is used as a pre-processing heuristic. Optimality is defined in terms of cost (based on the hardware cost), the extensibility of the solution (based on computation power left with a type penalty for the slow processor), and communication cost.

— **Scheduling Level:** At this approximation level the priorities are assigned to the tasks, signals are packed to messages and messages are assigned a priority. Optimality is defined by minimizing the end-to-end latencies of the sum of all the paths within the application.

The FTG+PM of the case study is shown in Figure 8.5. On the first approximation level, 1000 random solutions are created. These solution points are hill-climbed. The hill climber only accepts feasible solutions that are equal on all goal functions and at least better on one goal function. From the solutions found after the first approximation level, the Pareto optimal solutions are selected and transformed to a MILP representation. This MILP representation is executed and the results are transformed back to the deployment model representation. The MILP obtains the optimal solution for the second approximation level.

8.7.1 Transformations

We give a brief overview of the transformation rules involved in this case study. The transformation rules for randomly searching the design space consist of:

— Ecu2Type: This rule maps an ECU to a hardware type. The first part of the rule selects an unmapped ECU and assigns it a pivot. The second part of the rule assigns the selected ECU (using the pivot) to a type. The rule is executed until all ECUs have type information.

— Task2Ecu: This rule, shown in Figure 8.11, maps an unmapped software function to an ECU. As with the previous rule it is executed until all software functions are mapped to an ECU. A variant rule is created that, by construction, only creates feasible solutions. Here an extra constraint is imposed on the transformation rule. The load of the ECU is calculated in the constraint code of the rule. If the load exceeds 69%, the rule cannot be applied. If used, a part of the next rule becomes obsolete.

— CheckFeasible: This transformation rule checks whether all software functions are mapped to an ECU and that no ECU has a load greater than 69%. The 69% stems from rate-monotonic analysis, we are thus sure that a rate-monotonic schedule can be constructed that will always meet its deadline from these solutions.

Hill climbing is done with two rules that can move a software function from one ECU to another ECU (Figure 8.12). For the evaluation of a single solution point, we use transformations as well. One rule calculates the total cost and the total load of the system. The second rule calculates the total communication cost. Note that this could also be done by transforming the model to another language where analysis or simulation can be done.

Finally, three rules create a Mixed Integer Linear Program from a model after the architecture deployment step. We regard the execution of the MILP as a transformation as well since it manipulates the MILP-model and returns a set of traces containing the solution of the problem.

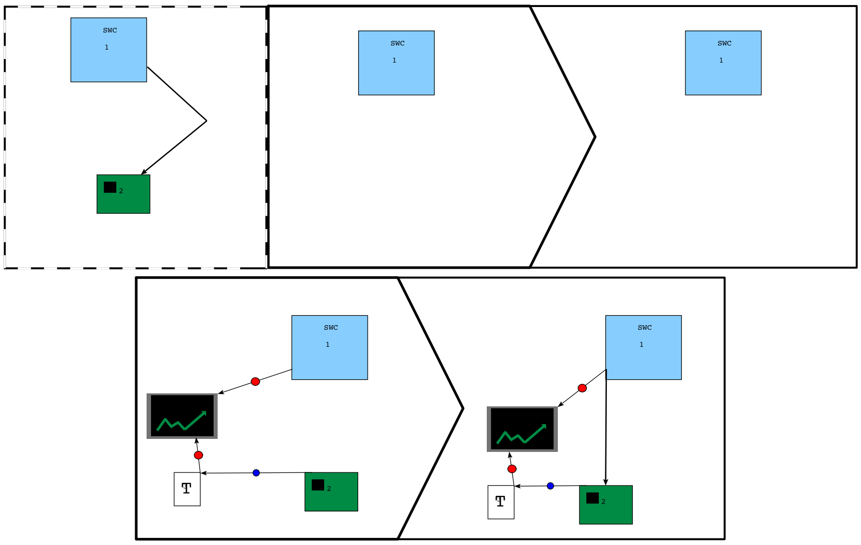


Figure 8.11: An example rule to match a software function to an ECU

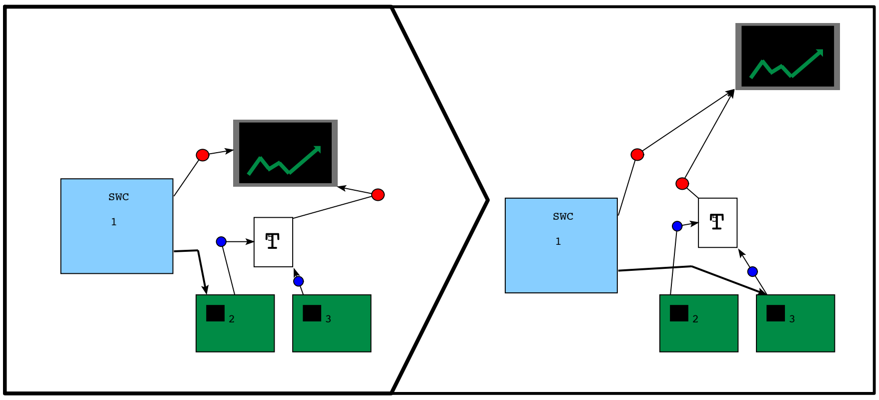


Figure 8.12: An example rule to move a software function to another ECU

8.7.2 Results

The industrial size model has $2^9 \times 9^{40} = 7.57 \times 10^{40}$ possible solutions at the first approximation level. The 1000 created solutions before hill climbing are shown in Figure 8.13. After the hill-climbing, 254 unique local optima remained. These are shown in Figure 8.14. As can be seen, the search is maximizing the extensibility while reducing the communication cost. This process used 125 hours of computation time on an 8 core Intel Xeon processor running at 2.66 GHz. The long computation time is because of the size of the model, though our focus was not on the performance of the approach.

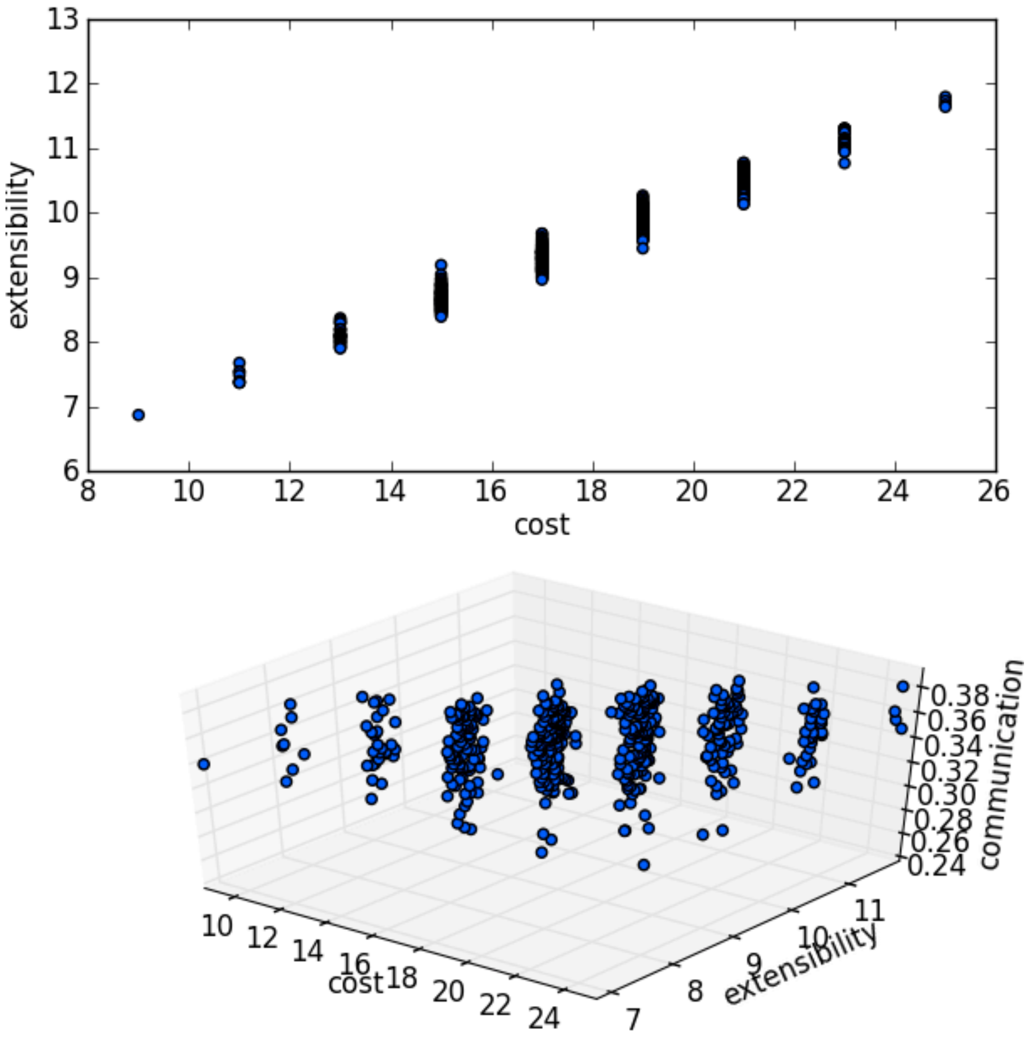


Figure 8.13: The solutions before hill climbing

At the second level of approximation, the number of possible solutions depends on the configuration after the first step. Nine Pareto front solutions got selected for the second

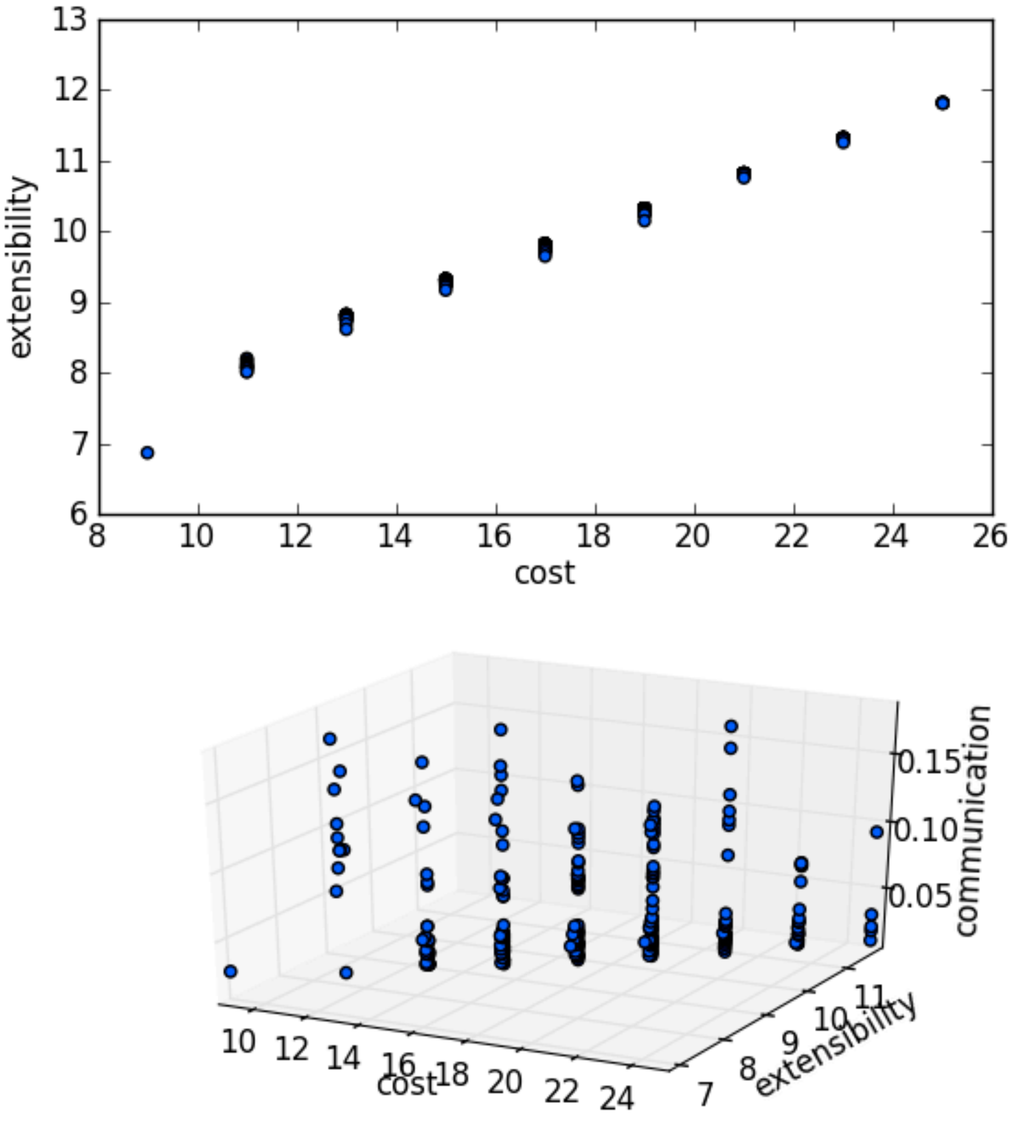


Figure 8.14: The local optima after hill climbing

part. The transformations and execution of the second approximation level used 10 minutes, on the same machine, to produce the optimal solution. The nine solutions are scheduled all within the deadlines of the case-study. The full MILP implementation can be seen in Appendix B.

8.8 Discussion

In this section we discuss some of the issues and opportunities of having a transformation based approach to design space exploration. The limiting factor in our prototype search transformation model is the execution time. The most expensive operation in the transformation language is the matching algorithm. The complexity of this operation is $\Theta(V!.V)$ [Syriani, 2011]. When using algorithms like hill climbing, a lot of matching restarts from the beginning while only a small portion of the model has been changed. To increase the speed of the matching operation, an incremental approach to model transformation can be used. An example incremental approach is proposed in [Bergmann *et al.*, 2008] where the 'rete' algorithm is implemented for speeding up the matching process. Using the 'rete' algorithm, a network of nodes is constructed, where each node (except the root) corresponds to a pattern occurring in the left-hand-side of a rule. The path from the root node to a leaf node defines a complete rule left-hand-side. Each node in the LHS has a memory that satisfy the constructed pattern. The 'rete' algorithm provides a speed-up by sacrificing memory. Parallelism can also be used to speed up the process, since the branches in the approach can in fact be executed in parallel. Other techniques like pivots and scoping can be used to define regions where to start the search for matches or to define a specific region where to optimize, reducing the load on the matching process.

Different opportunities exist besides those mentioned in the introduction. The transformations make domain knowledge explicit but they can also encode domain knowledge already known by the domain and/or integration experts. For example, when it is known that certain software functions have to be mapped together, a rule can be written that encodes this knowledge. Other domain knowledge can be discovered by mining the traces of the transformations. This can uncover the sensitivity of parameters, where the change of certain parameters has more effect then others. These are the choices that should be focussed on during design space exploration. The mining of the traces can also be used to uncover domain knowledge, for example when certain choices always lead to good or bad solutions.

8.9 Related Work

Performance analysis is crucial for the deployment of safe and cost-effective software-intensive systems. Balsamo et.al. [Balsamo *et al.*, 2004] present a review of research in the field of model-based performance prediction. The techniques are based on simulation models, process algebra, Petri nets and stochastic processes. An example of methods close to our research is architecture-based performance analysis [Spitznagel and Garlan, 1998] where a performance model, based on queuing networks, is derived from a system described in UML. For the design and deployment of software components, the Palladio component model [Becker *et al.*, 2009] offers a meta-model with annotations to describe extra-functional properties. The model can be transformed into both an analytical and a simulation model. Kugele et al. [Kugele *et al.*, 2009] use a similar approach by annotating a component-based meta-model with extra-functional properties. Part of the deployment is automated using an integer linear programming approach.

Platform-based design [Keutzer *et al.*, 2000] introduces clear abstraction levels and allows for separation of concerns between the refinement of the functional architecture specification and the abstractions of possible implementations. Di Natale and Sangiovanni-Vincentelli [Sangiovanni-Vincentelli and Di Natale, 2007] adapted the platform-based technique to the development of automotive embedded systems. The definitions of the models and architecture solutions, involved in the AUTOSAR process, are isolated from the details while still allowing enough information for the accurate prediction of the implementation's properties. The process is driven by a what-if analysis.

Popovici et al. developed an exploration technique based on platform-based design for the deployment of multimedia applications on MPSoC architectures [Popovici *et al.*, 2008]. The technique allows software code generation, software development platform generation and simulation model generation. This allows easy experimentation with different mappings of the software on the architecture. Different levels of abstraction are defined at which the generation, simulation and validation of the software components can take place.

Transformation based approaches to Design Space Exploration are relatively new topics in the field.

The DESERT tool-suite [Neema *et al.*, 2003] provides a framework for design space exploration. It allows an automated search for designs that meet structural requirements. Possible solutions are represented in a binary encoding that can generate all possibilities. A pruning tool is used to allow the user to select the designs that meet the requirements. These can then be reconstructed by decoding the selected design.

In [Saxena and Karsai, 2010], Saxena and Karsai present an MDE framework for general design space exploration. It comprises of an abstract design space exploration language and constraint specification language. Model transformation is used to transform the models and constraints to an intermediate language. This intermediate language can then be transformed to a representation that is used by a solver. As in our approach, a set of solvers can be supported by using model transformations. Though our approach combines different optimization steps where the process of exploration is defined explicitly. The constraints in our approach are made explicit in the transformation rules and not in a separate language.

Schätz et al. developed a declarative, rule-based transformation technique [Schätz *et al.*, 2010] to generate the constrained solutions of an embedded system. The rules are modified interactively to guide the exploration activity.

In [Hegedus and Horváth, 2011], a framework for guided design space exploration using graph transformations is proposed. The approach uses hints, provided by analysis, to reduce the traversal of states.

The OCTOPUS toolchain [Basten and Benthum, 2010] is a domain specific tool for the design space exploration of embedded systems. The tool is organised around an intermediate language used for connecting different tools together. The FTG+PM is used for the same purpose but uses an alternative approach without the need for an intermediate language.

The DECOS (Dependable Embedded Components and Systems) [Herzner *et al.*, 2007] approach uses model-based development techniques to build complex distributed embedded systems. The DECOS architecture enables the transition from a federated to a more integrated distributed architecture, integrating multiple subsystems on a single platform. As a consequence, the platform choices are less open and depend on a time-triggered communication bus. The developer is assisted during the deployment by a commercial tool, TTTech toolsuite, for schedulability analysis.

In addition to the above performance analysis methods and deployment space exploration techniques there are some other automatic methods for local and global optimization. Some use heuristic search methods, such as simulated annealing [Pop *et al.*, 2002], genetic algorithms [Sinnen, 2007], or use linear programming [Zheng *et al.*, 2007] or SAT-solving [Metzner and Herde, 2006]. Jackson et al. use a logic representation of the deployment problem and solve this model using the FORMULA tool to create feasible solutions [Jackson *et al.*, 2010].

8.10 Conclusions

In this chapter, we have shown that it is feasible to implement design space exploration through the usage of model transformations. Our solution embraces the different aspects of Multi-paradigm modelling by defining different levels of abstraction/approximation and using appropriate formalisms (in this case analysis, simulation and optimisation techniques). The enabler for this approach is model transformation combined with the explicit modelling of the process.

It is shown that an expressive transformation language can be used to implement meta-heuristics in the transformation models. This is complimented with the FTG+PM to combine different heterogeneous techniques at different levels of abstraction or approximation.

In the approach, the design space exploration step is integrated in the MPM development cycle with the benefit of having a uniform process. This results in documented, explicit design space exploration models that can be reused for later design variants, documentation, etc. The approach was applied to the power window and an industrial size automotive case study, yielding a set of Pareto-optimal solutions.

Chapter 9

Conclusions

All things are difficult before they are easy.
— Thomas Fuller

9.1 Summary

In this thesis we attempted to close some of the remaining gaps in the use of a multi-paradigm modelling approach for the design, verification and deployment of software-intensive systems. The principles of multi-paradigm modelling are used throughout the different phases of the design process namely different levels of abstraction/approximation and using the most appropriate formalisms for each of the design artefacts. A complete tool chain was created encompassing requirements engineering, domain specific modelling, verification, hybrid simulation, deployment (including the creation and calibration of the simulation models) and deployment space exploration. The process is modelled explicitly using the Formalism Transformation Graph and Process Model (FTG+PM). The approach was validated using the exemplar, the power window system (chapter 4).

We gave some anecdotal evidence that systematically and automatically deriving models of different complexity significantly increases productivity and quality of the models. This can be seen in the deployment space exploration part. Deployment solutions at different approximations levels are automatically generated using model transformations and evaluated. Infeasible solutions are pruned out as early as possible at the different levels of approximation. The quality of the models is further improved by adding search methods and domain knowledge in the transformation models. This results in a set of Pareto-optimal deployment models that can be used for implementation.

The remainder of the thesis focussed on some of the pending issues to complete the full development of software-intensive systems using MPM.

9.1.1 Process Modelling for MPM (Chapter 3)

In Chapter 3, process modelling aspects for multi-paradigm modelling are presented. Using the lessons learned from the megamodelling and process modelling communities, a platform, the FTG+PM, is presented to carry out formalism transformations to guide the design, verification and deployment of software-intensive systems. Megamodelling aspects are introduced in the Formalism Transformation Graph (FTG), where the relations, using transformation definitions, between different formalisms are represented. Transformations and models (typed in the FTG) are used in the Process Model (PM) to describe and prescribe a process. A sub-class of UML 2.0 activities are used for the process modelling part because of the focus on control- and dataflow. Some execution support for the FTG+PM is already available by using the lessons learned from the available transformation chaining frameworks.

9.1.2 The Power Window Exemplar (Chapter 4)

In Chapter 4 a complete specification and design of an accepted exemplar, the power window, is shown. The various phases of the development include: requirements engineering, verification, hybrid simulation and deployment. The process is explicitly modelled using the FTG+PM. For each of the different phases, appropriate formalisms were defined and the relations between them using transformations.

9.1.3 Explicit modelling of hybrid simulation models (Chapter 5)

While the novelty of the hybrid simulation is quite low, a hybrid simulation model and simulator of Causal Block Diagrams and Statecharts have been realised. The interface between the continuous and hybrid formalism, namely the State Event Locator and Transducer, is automatically generated using a transformation from a hybrid statechart formalism. The interface is completely modelled.

9.1.4 DEVS for platform modelling and simulation (Chapter 6)

For the evaluation of low-level deployment choices, DEVS was evaluated as a very appropriate formalism. DEVS allows the modelling of a complete ECU including the middleware and application layer. Also bus models can be constructed and connected with the ECUs using buffer models. The advantage of DEVS is that also plant and environment models in different formalisms can be expressed as DEVS atomic blocks. This way, the whole system can be modelled and simulated. Using the model, integration engineers can estimate the impact and evaluate the choices they make on the behaviour of the system.

9.1.5 Calibration of simulation models (Chapter 7)

In Chapter 7, guided by the research question *How can MPM leverage the calibration of simulation models of deployed software-intensive systems?*, we used a multi-paradigm approach for inferring calibration parameters for deployment simulation. The approach combines the created artefacts from the design process of the plant and environment together with instrumented code for the computational part. A hardware-in-the-loop like approach is taken where the environment and plant are simulated on a server and the instrumented code is executed on the target platform. A small middleware and communication infrastructure is automatically generated from the combined model. During execution, the environment and plant models transmit input values to the target board. The target board responds to these events by executing the instrumented source code. Output values are transmitted back to the plant model that can respond to changes in the dynamics of the system. At the same time, the measurements are transmitted back to the host so it can build a performance model of the software functions.

9.1.6 Automatic design space exploration (Chapter 8)

Chapter 8 investigated how multi-paradigm modelling can be used for the optimisation of the design of software-intensive systems. The approach uses the main philosophy of multi-paradigm modelling for guiding the exploration of the design space: using multiple abstraction/approximation levels and using the most appropriate formalisms (in this case optimisation methods). The rationale for using multiple abstraction/approximation levels is that bad solutions (that will never be feasible or optimal) are pruned earlier in the optimisation process. The main enabler for this is model transformation. The design or deployment model is transformed to another (most appropriate) optimisation model, like MILP. The results can be transformed back to the design model. If no such optimisation method is available, general purpose optimisation techniques, like hill-climbing, random search, exhaustive search, etc. are used in the transformation models by using an expressive transformation language. These transformations can also contain domain knowledge and heuristics to reduce an otherwise prohibitively extensive search for feasible and/or optimal solutions. Process modelling (using the FTG+PM) is used to combine the different techniques so a full optimisation is achieved. This results in a uniform process where the design space exploration knowledge is made explicit within the transformation models. These can later be reused for similar problems.

9.2 Future Work

In this thesis the feasibility for the design, verification and deployment of software-intensive systems was examined. We made a number of contributions to this domain but nevertheless, many challenges still remain. Some relevant directions are stated below.

9.2.1 Techniques

For the different techniques presented in this dissertation, a lot of work still has to be done to increase usability and robustness.

Calibration

As mentioned in Chapter 7, a number of simplifications were applied to generate the calibration parameters. These shortcomings should be remedied so the calibration can be applied to a wide range of processor architectures.

An extension to this work is the evaluation of how much information should be available in the models before calibration can start. In this work, the models of the plant, environment and control are completely defined before calibration can start. If trade-off analysis is further extended to higher abstraction levels, high-level estimation metrics are needed. For this, calibration is needed at higher levels where less information is present.

Design space Exploration

Different general search techniques, like exhaustive and random search were included in this feasibility study to design space exploration. Other general search techniques, for exploring the design space still need to be included. Examples are heuristic searches and other meta-heuristic searches, like simulated annealing and evolutionary techniques.

In our study, the inclusion of different heterogeneous techniques was used in an ad-hoc way. A solution for the problem was at hand and a transformation to this implementation was created. The use of these other optimisation methods should be made more formal by explicitly modelling constraints and goal functions. This will aid the transformation designer in creating these transformations or even generating them automatically.

Finally, incremental matching should be further developed for the purpose of design space exploration. The 'rete' algorithm can be used as a starting point for an implementation. Though other incremental techniques should be used as well, an example of a possible direction is matching a set of models with only a small difference.

9.2.2 Empirical Research on the MPM approach

While we cannot give any proof about the optimality of the approach, some anecdotal evidence is provided. A next step is to evaluate the proposed approach to a set of problems and compare it to different approaches. While empirical evaluation of MDE is currently a hot topic in the modelling community [Hutchinson *et al.*, 2011; Kuhn *et al.*, 2012], there has not been a lot of research on this topic. The main hurdle is that MDE is not yet widely adopted in the industry.

9.2.3 Feature Modelling

Product families are very important in the current software-intensive systems design chains. In this thesis we completely excluded this part in the development of the proposed tool-chain and involved techniques. Feature modelling is currently a popular approach for modelling product families. While, these reside at the top of the tool-chain, the techniques below should be adapted for this as well. Transformation based approaches could help alleviate the design of product families throughout the tool-chain. This would also require extensions to the techniques for the matching algorithms involved in the transformation languages to match product families.

9.2.4 Consistency Management

In the development of the examplar, we used a top-down approach. While this is a good start, normal business processes do not use such a vertical approach and legacy has to be considered more often than not. In large organisations, all this design information should be kept consistent on all the different abstraction and approximation levels. Large companies also work with suppliers that supply off-the-shelf software and system parts. All this information should be kept consistent on all the different views on the system, this also includes product families.

Appendix A

Power Window Use Case Descriptions

Use Case 1	**Open Driver Window**
Scope:	System-wide
Level:	The Driver wants to lower the driver window by pushing the lower window button
Actors:	Driver (A0), Driver Window Control Buttons(A1), Driver Window (A2)
Preconditions:	The Ignition System is turned to the start, on, accessory position
Postconditions:	The window stopped

Main Success Scenario:

1. Driver A0 pushes the lower button A1.
2. Driver Window A2 goes down.
3. Driver A0 releases the lower button A1.
4. Driver Window A2 stops.

Alternate Scenario

1. Driver A0 pushes the lower button A1.
2. Driver Window A2 goes down.
3. Driver Window A2 is fully opened and stops.
4. Driver A0 releases the lower button A1.

Special Requirements:

— The window reacts within 200 ms.

— The window is fully opened within 4 s.

Use Case 2	**Close Driver Window**
Scope:	System-wide
Level:	The Driver wants to raise the driver window by pushing the lower window button
Actors:	Driver (A0), Driver Window Control Buttons(A1), Driver Window (A2)
Preconditions:	The Ignition System is turned to the start, on, accessory position
Postconditions:	The window stopped

Main Success Scenario:

1. Driver A0 pushes the raise button A1.
2. Driver Window A2 goes up.
3. Driver A0 releases the raise button A1.
4. Driver Window A2 stops.

Alternate Scenario

1. Driver A0 pushes the raise button A1.
2. Driver Window A2 goes up.
3. Driver Window A2 is fully closed and stops.
4. Driver A0 releases the raise button A1.

Special Requirements:

— The window reacts within 200 ms.

— The window is fully opened within 4 s.

Use Case 3	**Lockout Passenger Window**
Scope:	System-wide
Level:	The Driver wants to lockout the passenger window by pushing the lockout window button
Actors:	Driver (A0), Passenger Window Control Buttons(A1), Driver Window (A2)
Preconditions:	The Ignition System is turned to the start, on, accessory position
Postconditions:	The passenger window is locked out

Main Success Scenario:

1. Driver A0 pushes the lockout button A1.
2. Passenger window (A2) is locked out.

Exception, window is already locked out:

1. Driver A0 pushes the lockout button A1.
2. Passenger window (A2) is no longer locked out

Use Case 5	**Open Passenger Window**
Scope:	System-wide
Level:	The Passenger or Driver wants to lower the passenger window by pushing the lower window button
Actors:	Driver (A0), Passenger (A1), Driver Window Control Buttons(A2), Passenger Window Control Buttons (A3), Passenger Window (A4)
Preconditions:	— The Ignition System is turned to the start, on, accessory position — the window is not locked out
Postconditions:	The window stopped

Main Success Scenario:

1. Passenger A1 pushes the lower button using A3 or Driver A0 pushes the lower button using A1.
2. Passenger Window A4 goes down.
3. Passenger A1 releases the lower button A3 or Driver A0 releases the lower button using A1.
4. Passenger Window A4 stops.

Alternate Scenario

1. Passenger A1 pushes the lower button using A3 or Driver A0 pushes the lower button using A1.
2. Passenger Window A4 goes down.
3. Passenger Window A4 is fully opened and stops.
4. Passenger A1 releases the lower button A3 or Driver A0 releases the lower button using A1.

Special Requirements:

— The window reacts within 200 ms.

— The window is fully opened within 4 s.

— Driver commands have priority over Passenger commands.

Appendix B

MILP implementation

The full implementation of the Mixed Integer Linear Program using the CPLEX solver, as implemented by Gang Han (McGill University) based on the model of Haibo Zeng (McGill University).

```
#a_t[i][j]:whether task i is allocated on ECU j
# (1: yes, 0: no)
#C_t[i]: the WCET of task i on its allocated ECU
#g_m[i]:whether message m is global, i.e, the source and sink of m are
# on different ECUs(1: yes, 0: no)
#h_t[i][j]: whether task i and task j are on the same ECU (1: yes,0: no)
#h_t_e[i][j][e]: whether task i and task j are both on ECU e
# (1: yes, 0: no)


def mapping(a_t, C_t, g_m, h_t, h_t_e):
    r_t = []# {TASKS} >= 0;
    p_t = []# p_t[i][j]=1: the priority of task j is higher than task i
    w_t = []# {TASKS, TASKS} >= 0;
    y_t = []# {TASKS, TASKS} >= 0;
    x_t = []# {TASKS, TASKS} integer >= 0;
    r_m = []# {MESSAGES} >= 0;
    p_m = []# p_m[i][j]=1:the priority of msg j is higher than msg i
    w_m = []# {MESSAGES, MESSAGES} >= 0;
    y_m = []# {MESSAGES, MESSAGES} >= 0;
    x_m = []# {MESSAGES, MESSAGES} integer >= 0;
    L = []# {PATHS} > 0

    model = cplex.Cplex()
    model.objective.set_sense(model.objective.sense.minimize)
    #subject to Task_Alloc_3 {i in TASKS, j in TASKS, e in ECUS}:
    #         h_t_e[i, j, e] >= a_t[i, e] +  a_t[j, e] - 1;
    #subject to Task_Alloc_4 {i in TASKS, j in TASKS, e in ECUS}:
    #         h_t_e[i, j, e] <= a_t[i, e];
    #subject to Task_Alloc_5 {i in TASKS, j in TASKS, e in ECUS}:
```

```
    #        h_t_e[i, j, e] <= a_t[j, e];
    #subject to Task_Alloc_6 {i in TASKS, j in TASKS}:
    #        h_t[i, j] = sum{e in ECUS} h_t_e[i, j, e];
    #### Global Signal(Message) Constraints
    #subject to Global_Signal {i in MESSAGES}:
    #   g_m[i] = 1- h_t[SRC_m[i], DET_m[i]];

    ###### Task WCRT
    #subject to Task_Response_Time1 {i in TASKS}:
    #   r_t[i] = C_t[i] + sum{k in TASKS: i <> k} (w_t[i, k] * C_t[k]);

    index_i = 0
    for i in TASKS:
        index_k = 0
        varname_i = "r_t[" + i + "]"
        r_t.append(varname_i)
        w_t.append([])
        coef = [1]
        convars = [varname_i]
        model.variables.add(names=[varname_i], lb=[0],
            ub=[cplex.infinity],
            types=["C"])

        for k in TASKS:
            varname_i_k = "w_t[" + i + "][" + k + "]"
            w_t[index_i].append(varname_i_k)
            if i != k:
                coef.append(-C_t[index_k])
                convars.append(varname_i_k)
                model.variables.add(names=[varname_i_k], lb=[0],
                    ub=[cplex.infinity], types=["C"])
            index_k += 1
        model.linear_constraints.add(lin_expr=[
            cplex.SparsePair(convars, coef)],
            senses=["E"], rhs=[C_t[index_i]])
        index_i += 1


    #subject to Task_Response_Time2_1 {i in TASKS, k in TASKS: i<>k}:
    #   y_t[i, k] - CONSTANT_M * (1 - p_t[i, k]) <= w_t[i, k];

    #subject to Task_Response_Time2_2 {i in TASKS, k in TASKS: i<>k}:
    #   w_t[i, k] <= y_t[i, k];

    #subject to Task_Response_Time2_3 {i in TASKS, k in TASKS: i<>k}:
    # w_t[i, k] <= CONSTANT_M * p_t[i, k];

    #subject to Task_Response_Time2_4 {i in TASKS, k in TASKS: i<>k}:
    #   x_t[i, k] - CONSTANT_M * (1 - h_t[i, k]) <= y_t[i, k];

    #subject to Task_Response_Time2_5 {i in TASKS, k in TASKS: i<>k}:
    #   y_t[i, k] <= x_t[i, k];
```

```
#subject to Task_Response_Time2_6 {i in TASKS, k in TASKS: i<>k}:
#   y_t[i, k] <= CONSTANT_M * h_t[i, k];

#subject to Task_Response_Time3 {i in TASKS, k in TASKS: i<>k}:
#   0 <= x_t[i, k] - r_t[i] / T_t[k] <= 0.9999999;

index_i = 0
for i in TASKS:
    index_k = 0
    y_t.append([])
    x_t.append([])
    p_t.append([])
    for k in TASKS:
        y_varname_i_k = "y_t[" + i + "][" + k + "]"
        x_varname_i_k = "x_t[" + i + "][" + k + "]"
        p_varname_i_k = "p_t[" + i + "][" + k + "]"
        y_t[index_i].append(y_varname_i_k)
        x_t[index_i].append(x_varname_i_k)
        p_t[index_i].append(p_varname_i_k)
        if i != k:
            model.variables.add(names=[y_varname_i_k], lb=[0],
                ub=[cplex.infinity], types=["C"])
            model.variables.add(names=[x_varname_i_k], lb=[0],
                ub=[cplex.infinity], types=["I"])
            model.variables.add(names=[p_varname_i_k], lb=[0],
                ub=[1], types=["B"])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k, p_varname_i_k,
                     w_t[index_i][index_k]],
                    [1, CONSTANT_M, -1])],
                senses=["L"], rhs=[CONSTANT_M])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [w_t[index_i][index_k], y_varname_i_k],
                    [1, -1])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [w_t[index_i][index_k], p_varname_i_k],
                    [1, -CONSTANT_M])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [x_varname_i_k, y_varname_i_k],
                    [1, -1])], senses=["L"],
                rhs=[CONSTANT_M *(1 - h_t[index_i][index_k])])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k, x_varname_i_k],
                    [1, -1])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k],
                    [1])], senses=["L"],
                rhs=[CONSTANT_M * h_t[index_i][index_k]])
            model.linear_constraints.add(lin_expr=[
                cplex.SparsePair([x_varname_i_k, r_t[index_i]],
                    [1, -1.0 / T_t[index_k]])], senses=["L"],
                rhs=[0.9999999])
            model.linear_constraints.add(lin_expr=[
                cplex.SparsePair([r_t[index_i], x_varname_i_k],
                    [1.0 / T_t[index_k], -1])],
                    senses=["L"], rhs=[0])
        index_k += 1
    index_i += 1

#i!=j,p_t[i,j]+p_t[j,i]=1
for i in range(len(TASKS)):
    for j in range(len(TASKS) - i - 1):
        #p_t[i][i+j+1]+p_t[i+j+1][i]=1
        model.linear_constraints.add(lin_expr=[
            cplex.SparsePair([p_t[i][i + j + 1], p_t[i + j + 1][i]],
                [1, 1])], senses=["E"], rhs=[1])

###### Message WCRT
#subject to Message_Response_Time1 {i in MESSAGES}:
# r_m[i] >= (g_m[i] - 1) * CONSTANT_M + C_m[i] +
# Bmax + sum{k in MESSAGES: i <> k} (w_m[i, k] * C_m[k]);

#subject to Message_Response_Time1_1 {i in MESSAGES}:
# r_m[i] <= g_m[i] * CONSTANT_M;

index_i = 0
for i in MESSAGES:
    index_k = 0
    varname_i = "r_m[" + i + "]"
    r_m.append(varname_i)
    w_m.append([])
    coef = [-1]
    convars = [varname_i]
    model.variables.add(names=[varname_i], lb=[0],
        ub=[cplex.infinity], types=["C"])
    model.linear_constraints.add(lin_expr=
    [cplex.SparsePair([varname_i], [1])],
        senses=["L"], rhs=[g_m[index_i] * CONSTANT_M])

    for k in MESSAGES:
        varname_i_k = "w_m[" + i + "][" + k + "]"
        w_m[index_i].append(varname_i_k)
        if i != k:
            coef.append(C_m[index_k])
            convars.append(varname_i_k)
            model.variables.add(names=[varname_i_k],
                lb=[0],
                ub=[cplex.infinity], types=["C"])
        index_k += 1
    model.linear_constraints.add(
        lin_expr=[
            cplex.SparsePair(convars, coef)], senses=["L"],
        rhs=[CONSTANT_M * (1 - g_m[index_i])
             - C_m[index_i] - Bmax])
    index_i += 1

#subject to Message_Response_Time2_1
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    y_m[i, k] - CONSTANT_M * (1 - p_m[i, k]) <= w_m[i, k];

#subject to Message_Response_Time2_2
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    w_m[i, k] <= y_m[i, k];

#subject to Message_Response_Time2_3
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    w_m[i, k] <= CONSTANT_M * p_m[i, k];

#subject to Message_Response_Time3_1
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    x_m[i, k] - CONSTANT_M * (1 - g_m[k]) <= y_m[i, k];

#subject to Message_Response_Time3_2
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    y_m[i, k] <= x_m[i, k];

#subject to Message_Response_Time3_3
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    y_m[i, k] <= CONSTANT_M * g_m[k];

#subject to Message_Response_Time4
# {i in MESSAGES, k in MESSAGES: i<>k}:
#    0 <= x_m[i, k] - (r_m[i] - C_m[i] * g_m[i])
# / T_m[k] <= 0.9999999;

index_i = 0
for i in MESSAGES:
    index_k = 0
    y_m.append([])
    x_m.append([])
    p_m.append([])

    for k in MESSAGES:
        y_varname_i_k = "y_m[" + i + "][" + k + "]"
        x_varname_i_k = "x_m[" + i + "][" + k + "]"
        p_varname_i_k = "p_m[" + i + "][" + k + "]"
        y_m[index_i].append(y_varname_i_k)
        x_m[index_i].append(x_varname_i_k)
        p_m[index_i].append(p_varname_i_k)

        if i != k:
            model.variables.add(names=[y_varname_i_k],
                lb=[0], ub=[cplex.infinity], types=["C"])
            model.variables.add(names=[x_varname_i_k],
                lb=[0], ub=[cplex.infinity], types=["I"])
            model.variables.add(names=[p_varname_i_k],
                lb=[0], ub=[1], types=["B"])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k, p_varname_i_k,
                     w_m[index_i][index_k]],
                    [1, CONSTANT_M, -1])],
                senses=["L"], rhs=[CONSTANT_M])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [w_m[index_i][index_k], y_varname_i_k],
                    [1, -1])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [w_m[index_i][index_k], p_varname_i_k],
                    [1, -CONSTANT_M])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [x_varname_i_k, y_varname_i_k],
                    [1, -1])], senses=["L"],
                rhs=[CONSTANT_M *(1 - g_m[index_k])])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k, x_varname_i_k],
                    [1, -1])], senses=["L"], rhs=[0])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [y_varname_i_k],
                    [1])], senses=["L"],
                rhs=[CONSTANT_M * g_m[index_k]])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [x_varname_i_k, r_m[index_i]],
                    [1, -1.0 / T_m[index_k]])], senses=["L"],
                rhs=[0.9999999 - 1.0 * g_m[index_i] *
                                 C_m[index_i] / T_m[index_k]])
            model.linear_constraints.add(
                lin_expr=[cplex.SparsePair(
                    [x_varname_i_k, r_m[index_i]],
                    [-1, 1.0 / T_m[index_k]])], senses=["L"],
                rhs=[1.0 * g_m[index_i]
                     * C_m[index_i] / T_m[index_k]])
        index_k += 1
    index_i += 1

#i!=j,p_m[i,j]+p_m[j,i]=1

for i in range(len(MESSAGES)):
    for j in range(len(MESSAGES) - i - 1):
        #p_m[i][i+j+1]+p_m[i+j+1][i]=1
        model.linear_constraints.add(
            lin_expr=[cplex.SparsePair([p_m[i][i + j + 1],
                                        p_m[i + j + 1][i]],
                [1, 1])], senses=["E"], rhs=[1])

        #minimize Total_Latency:
        #   sum {p in PATHS} L[p];

for p in PATHS:
    varname = "L[" + p + "]"
    L.append(varname)

model.variables.add(names=L, lb=[0] * len(L),
    ub=[cplex.infinity] * len(L), types=["C"] * len(L),
    obj=[1] * len(L))

index_p = 0
for p in PATHS:
    right_side = 0
    model.linear_constraints.add(
        lin_expr=[cplex.SparsePair([L[index_p]], [1])],
        senses=["L"], rhs=[deadline[index_p]])
    convars = [L[index_p]]
    coef = [-1]

    for i in TASKS_ON_PATH[index_p]:
        convars.append(r_t[TASKS.index(i)])
        coef.append(1)

    for i in MESSAGES_ON_PATH[index_p]:
        index_i = MESSAGES.index(i)
        convars.append(r_m[index_i])
        coef.append(1)
        right_side += -(T_m[index_i] +
                        T_t[TASKS.index(DET_m[index_i])]) *\
                      g_m[index_i]

    model.linear_constraints.add(
        lin_expr=[cplex.SparsePair(convars, coef)],
        senses=["E"], rhs=[right_side])
    index_p += 1

#print convars,coef

try:
    model.solve()
except CplexError, exc:
    print exc
    return
print
# solution.get_status() returns an integer code
print "Solution status = ", model.solution.get_status(), ":",
# the following line prints the corresponding string
print model.solution.status[model.solution.get_status()]
print "Solution value  = ", model.solution.get_objective_value()


#numcols = model.variables.get_num()
#for j in range(numcols):
#   print model.variables.get_names(j),model.solution.get_values(j)

index = 0
index_i = 0
for i in TASKS:
    if r_t[index_i] != model.variables.get_names(index):
        print "0000"
    r_t[index_i] = model.solution.get_values(index)
    index += 1

    index_k = 0

    for k in TASKS:
        if i != k:
            if w_t[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            w_t[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1

        index_k += 1

    index_i += 1

index_i = 0

for i in TASKS:
    index_k = 0
    for k in TASKS:
        if i != k:
            if y_t[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            y_t[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
            if x_t[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            x_t[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
            if p_t[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            p_t[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
        index_k += 1
    index_i += 1
index_i = 0

for i in MESSAGES:
    if r_m[index_i] != model.variables.get_names(index):
        print "0000"
    r_m[index_i] = model.solution.get_values(index)
    index += 1
    index_k = 0
    for k in MESSAGES:
        if i != k:
            if w_m[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            w_m[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
        index_k += 1
    index_i += 1
index_i = 0

for i in MESSAGES:
    index_k = 0
    for k in MESSAGES:
        if i != k:
            if y_m[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            y_m[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
            if x_m[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            x_m[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
            if p_m[index_i][index_k] !=\
               model.variables.get_names(index):
                print "0000"
            p_m[index_i][index_k] =\
            model.solution.get_values(index)
            index += 1
        index_k += 1
    index_i += 1
index_p = 0

for p in PATHS:
    if L[index_p] != model.variables.get_names(index):
        print "0000"
    L[index_p] = model.solution.get_values(index)
```

```
        index += 1

        index_p += 1

    #return a tuple including the priority of tasks,messages,
    # the latency of paths and the sum of these latencies
    return (p_t, p_m, L, model.solution.get_objective_value())
```

Bibliography

[Aldazabal *et al.*, 2008] Aitor Aldazabal, Terry Baily, Felix Nanclares, Andrey Sadovykh, Christian Hein, Essern Martin, and Tom Ritter. Automated model driven development processes. In *ECMDA Workshop on Model Driven Tool and Process Integration*, pages 1–12, 2008.

[AUTOSAR, 2012] AUTOSAR. Automotive open system architecture. www.autosar.org, 2012.

[Balarin *et al.*, 2002] Felice Balarin, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Yosinori Watanabe, and Guang Yang. Concurrent execution semantics and sequential simulation algorithms for the metropolis meta-model. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 13–18, New York, NY, USA, 2002. ACM.

[Balarin *et al.*, 2003] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and a. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, April 2003.

[Balsamo *et al.*, 2004] S. Balsamo, a. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: a survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[Barbero *et al.*, 2007] Mikaël Barbero, Marcos Didonet Del Fabro, and Jean Bézivin. Traceability and provenance issues in global model management. In *3rd ECMDA-Traceability Workshop*, 2007.

[Basten and Benthum, 2010] Twan Basten and Emiel Van Benthum. Model-driven design-space exploration for embedded systems: the octopus toolset. In *Leveraging applications of formal methods, verification, and validation, LNCS*, pages 90–105. Springer, 2010.

[Bastoni *et al.*, 2010] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Cache-Related Preemption and Migration Delays : Empirical Approximation and Impact on Schedulability. In *Proceedings of the 6th International Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2010.

[Becker *et al.*, 2009] S Becker, H Koziolek, and R Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, January 2009.

[Bendraou *et al.*, 2007] Reda Bendraou, Benoit Combemale, Xavier Crégut, and Marie-Pierre Gervais. Definition of an eXecutable SPEM 2.0. 2007.

[Bendraou *et al.*, 2010] Reda Bendraou, Jean-Marc Jezequel, Marie-Pierre Gervais, and Xavier Blanc. A comparison of six uml-based languages for software process modeling. *IEEE Transactions on Software Engineering*, 36(5):662–675, September 2010.

[Bergmann *et al.*, 2008] Gábor Bergmann, András Ökrös, István Ráth, Dániel Varró, and Gergely Varró. Incremental pattern matching in the viatra model transformation system. *Proceedings of the third international workshop on Graph and model transformations - GRaMoT '08*, page 25, 2008.

[Bernat *et al.*, 2003] Guillem Bernat, Antoine Colin, and Stefan Petters. pWCET: A tool for probabilistic worst-case execution time analysis of real-time systems. Technical report, University Of York, 2003.

[Bézivin *et al.*, 2003] J Bézivin, S Gérard, PA Muller, and L Rioux. MDA components: Challenges and Opportunities. In *Proc. of First International Workshop on Metamodelling for MDA*, 2003.

[Biehl and Törngren, 2012] Matthias Biehl and Martin Törngren. Constructing tool chains based on spem process models. In *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, pages 267–273, 2012.

[Biehl *et al.*, 2012] Matthias Biehl, Jad El-Khoury, Frederic Loiret, and Martin Törngren. On the modeling and generation of service-oriented tool chains. *Software & Systems Modeling*, pages 1–20, 2012.

[Biehl, 2013] Matthias Biehl. *A Modeling Language for the Description and Development of Tool Chains for Embedded Systems*. PhD thesis, Royal Institute of Technology, KTH, 2013.

[Blochwitz *et al.*, 2011] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Modelica Conference*, 2011.

[Borland, 2003] Spencer Borland. Transforming Statechart Models to DEVS. Master's thesis, McGill, 2003.

[Bottoni and Saporito, 2009] P. Bottoni and A. Saporito. Resource-based enactment and adaptation of workflows from activity diagrams. *Electronic Communications of the EASST*, 18, 2009.

[Boucher and Kelly-Rand, 2011] Michelle Boucher and Colin Kelly-Rand. System Design: Get it Right the First Time. Technical report, Aberdeen Group, 2011.

[Boulanger and Hardebolle, 2008] F. Boulanger and C. Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. *2008 International Conference on Software Testing, Verification, and Validation*, pages 318–327, April 2008.

[Boulanger *et al.*, 2011] Frédéric Boulanger, Ayman Dogui, and Cécile Hardebolle. Semantic Adaptation using CCSL Clock Constraints Semantic Adaptation using CCSL Clock Constraints. *Electronic Communications of the EASST: Proceedings of the 4th International Workshop on Multi-Paradigm Modeling*, 50, 2011.

[Bouyssounouse and Sifakis, 2005] Bruno Bouyssounouse and Joseph Sifakis. *Embedded System Design, The ARTIST Roadmap for Research and Development*. Springer, 2005.

[Broy *et al.*, 2007] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[Broy, 2006] Manfred Broy. Challenges in automotive software engineering. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 33, 2006.

[Bunte *et al.*, 2011] Sven Bunte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. Improving the Confidence in Measurement-Based Timing Analysis. *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 144–151, March 2011.

[Canada Transport, 2009] Canada Transport. Power-Operated Window , Partition , and Roof Panel Systems. Technical report, Standards Research and Development Branch - Road Safety and Motor Vehicle Regulation Directorate, 2009.

[Chevalley and Thévenod-Fosse, 2001] Philippe Chevalley and Pascale Thévenod-Fosse. Automated generation of statistical test cases from uml state diagrams. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, pages 205–214, Washington, DC, USA, 2001. IEEE Computer Society.

[Chilenski, 2001] John Joseph Chilenski. An investigation of three forms of the modified condition decision coverage (mcdc) criterion. Technical report, Office of Aviation Research, 2001.

[Chou, 2002] Shih-Chien Chou. A Process Modeling Language Consisting of High Level UML-based Diagrams and Low Level Process Language. *The Journal of Object Technology*, 1(4):137, 2002.

[Cicchetti *et al.*, 2011] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. JTL: A Bidirectional and Change Propagating Transformation Language. In Brian A. Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering - 3rd Int. Conf., SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, volume 6563 of *Lecture Notes in Computer Science*, pages 183–202. Springer, 2011.

[Colin, 2003] Antoine Colin. Experimental evaluation of code properties for WCET analysis. *Real-Time Systems Symposium*, 2003.

[Cormen *et al.*, 2001] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2001.

[Costagliola *et al.*, 2002] G. Costagliola, A. Delucia, S. Orefice, and G. Polese. A classification framework to support the design of visual languages. *Journal of Visual Languages & Computing*, 13(6):573–600, 2002.

[Cuenot *et al.*, 2007] Philippe Cuenot, S Gerard, H Lonn, M-O Reiser, D Servat, C-J Sjostedt, RT Kolagari, M Torngren, M Weber, et al. Managing complexity of automotive electronics using the east-adl. In *12th IEEE International Conference on Engineering Complex Computer Systems*, pages 353–358. IEEE, 2007.

[Czarnecki and Helsen, 2006] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[Czarnecki *et al.*, 2012] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, and Klaus Schmid. Cool Features and Tough Decisions : A Comparison of Variability Modeling Approaches. In *Variability Modelling of Software-intensive Systems (VaMoS)*, 2012.

[de Souza *et al.*, 2000] S.R.S. de Souza, J.C. Maldonado, and S. Fabbri. Statecharts specifications: A family of coverage testing criteria, 2000.

[Debusmann and Geihs, 2003] Markus Debusmann and Kurt Geihs. Efficient and transparent instrumentation of application components using an aspect-oriented approach. *Self-Managing Distributed Systems*, pages 227–240, 2003.

[Deverge and Puaut, 2005] Jean-François Deverge and Isabelle Puaut. Safe measurement-based WCET estimation. In *5th International Workshop on Worst-Case Execution Time Analysis*, pages 13–16, 2005.

[Dhaussy and Roger, 2011] P; Dhaussy and J. Roger. CTL Language Specification. Technical report, ENSTA Bretagne, 2011.

[Di Natale *et al.*, 2009] M. Di Natale, P. Giusto, and A. Sangiovanni-Vincentelli. Stochastic Analysis of CAN-Based Real-Time Automotive Systems. *IEEE Transactions on Industrial Informatics*, 5(4):388–401, November 2009.

[Diaw *et al.*, 2010] Samba Diaw, Redouane Lbath, Thái Lê Vinh, and Bernard Coulette. SPEM4MDE: a Metamodel for MDE Software Processes Modeling and Enactment. *3rd Workshop on Model-Driven Tool & Process Engineering*, pages 109–121, 2010.

[Diaw *et al.*, 2011] Samba Diaw, Redouane Lbath, and Bernard Coulette. Specification and implementation of SPEM4MDE, a metamodel for MDE software processes. In *International Conference on Software Engineering and Knowledge Engineering*, 2011.

[dSpace, 2013] dSpace. dspace website. http://www.dspace.com, 2013.

[Eker *et al.*, 2003] J. Eker, J.W. Janneck, E.A. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[EMF, 2012] EMF. The Eclipse Modelling Framework, 2012.

[Engblom *et al.*, 2003] Jakob Engblom, Andreas Ermedahl, Mikael Sjodin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):437–455, August 2003.

[Esmaeilsabzali *et al.*, 2009] Shahram Esmaeilsabzali, Nancy A. Day, and Joanne M. Atlee. Big-Step Semantics. Technical report, University of Waterloo, 2009.

[Farsi *et al.*, 1999] M. Farsi, K. Ratcliff, and M. Barbosa. An overview of controller area network. *Computing & Control Engineering Journal*, 10(3):113–120, 1999.

[Favre, 2004] J.M. Favre. Foundations of Model-Driven Reverse Engineering – Episode I: Story of The Fidus Papyrus and the Solarus. In *Post-proceedings of Dagsthul Seminar on Model-Driven Reverse Engineering*, pages 1–29, 2004.

[Feiler *et al.*, 2006] Peter H Feiler, Bruce A Lewis, and Steve Vestal. The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. In *IEEE International Conference on Control Applications*, pages 1206–1211. IEEE, 2006.

[Ferdinand, 2004] Christian Ferdinand. Worst case execution time prediction by static program analysis. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, volume 00, pages 17–19. IEEE Computer Society, 2004.

[Fischmeister and Lam, 2010] Sebastian Fischmeister and Patrick Lam. Time-aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 6(4):652–663, 2010.

[Garousi *et al.*, 2006] Vahid Garousi, L.C. Briand, and Yvan Labiche. Traffic-aware stress testing of distributed systems based on UML models. In *Proceedings of the 28th international conference on Software engineering*, pages 391–400. ACM, 2006.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth Int. Conf. on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[Giese, 2006] Holger Giese. Software Engineering for Software-Intensive Systems: I Introduction. Technical report, University of Paderborn, 2006.

[Guerra *et al.*, 2010] Esther Guerra, Juan De Lara, D. Kolovos, Paige, Richard F., and Osmar Marchi dos Santos. transML: a family of languages to model model transformations. In *Model Driven Engineering Languages and Systems, LNCS 6394*, pages 1–15, 2010.

[Hamann *et al.*, 2006] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems*, 33(1):101–137, 2006.

[Harel and Naamad, 1996] David Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[Harel and Rumpe, 2004] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of semantics? *Computer*, 37(10):64–72, 2004.

[Harel, 1987] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[Hastono *et al.*, 2004] Prih Hastono, Stephan Klaus, and S.A. Huss. An integrated SystemC framework for real-time scheduling assessments on system level. In *Proceedings of IEEE International Real-Time Systems Symposium*, 2004.

[Hebig *et al.*, 2011] Regina Hebig, Andreas Seibel, and Holger Giese. On the Unification of Megamodels. *Electronic Communications of the EASST: Proceedings of the 4th International Workshop on Multi-Paradigm Modeling*, 42, 2011.

[Hegedus and Horváth, 2011] A. Hegedus and A. Horváth. A model-driven framework for guided design space exploration. In *Automated Software Engineering (ASE), 2011*, 2011.

[Heidenreich *et al.*, 2011] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. Safe Composition of Transformations. *The Journal of Object Technology*, 10:7:1, 2011.

[Henderson-Sellers and Gonzalez-Perez, 2005] B. Henderson-Sellers and C. Gonzalez-Perez. A comparison of four process metamodels and the creation of a new generic standard. *Information and Software Technology*, 47(1):49–65, January 2005.

[Henriksson *et al.*, 2002] D. Henriksson, A. Cervin, and K.E. Årzén. Truetime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control. Barcelona, Spain*, 2002.

[Herzner *et al.*, 2007] Wolfgang Herzner, Rupert Schlick, Martin Schlager, Bernhard Leiner, B. Huber, A. Balogh, G. Csertan, A. Le Guennec, T. Le Sergent, N. Suri, and Others. Model-based development of distributed embedded real-time systems with the decos tool-chain. In *proceedings of the 2007 SAE AeroTech Congress & Exhibition*, 2007.

[Hutchinson *et al.*, 2011] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of MDE in industry. *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 471, 2011.

[IBM, 2009] IBM. IBM ILOG CPLEX optimization studio, 2009.

[Jackson *et al.*, 2010] E.K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for mda. In *Proceedings of the 10th ACM international conference on Embedded software*, pages 39–48. ACM, 2010.

[KansomKeat *et al.*, 2008] Supaporn KansomKeat, Jeff Offutt, Aynur Abdurazik, and Andrea Baldini. A Comparative Evaluation of Tests Generated from Different UML Diagrams. In *SNDP*, 2008.

[Keutzer *et al.*, 2000] K. Keutzer, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(12):1523–1543, 2000.

[Kirner *et al.*, 2002] Raimund Kirner, Roland Lang, and Gerald Freiberger. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *14th EUROMICRO Conference on Real-Time Systems,*, 2002.

[Kleppe, 2006] Anneke Kleppe. MCC: A model transformation environment. In *European Conference on Model Driven ArchitectureFoundations and Applications, LNCS 4066*, pages 173–187. Springer, 2006.

[Knieke and Goltz, 2010] Christoph Knieke and Ursula Goltz. An executable semantics for uml 2 activity diagrams. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, FML '10, pages 3:1–3:5, New York, NY, USA, 2010. ACM.

[Köhl and Jegminat, 2005] Susanne Köhl and Dirk Jegminat. How to Do Hardware-in-the-Loop Simulation Right. *SAE Technical Papers*, (724), 2005.

[Kolovos *et al.*, 2008] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. A framework for composing modular and interoperable model management tasks. In *MDTPI Workshop, EC-MDA*, 2008.

[Krause *et al.*, 2007] Matthias Krause, Oliver Bringmann, André Hergenhan, Gökhan Tabanoglu, and Wolfgang Rosentiel. Timing Simulation of Interconnected AUTOSAR Software-Components. In *Conference on Design, Automation and Test in Europe*, 2007.

[Kugele *et al.*, 2009] Stefan Kugele, Wolfgang Haberl, Michael Tautschnig, and Martin Wechs. Optimizing automatic deployment using non-functional requirement annotations. *Leveraging Applications of Formal Methods, Verification and Validation*, pages 400–414, 2009.

[Kuhn *et al.*, 2012] Adrian Kuhn, Gail C. Murphy, and Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In *MoDELS 2012*, pages 352–367, 2012.

[Lacoste-Julien *et al.*, 2004] S. Lacoste-Julien, H. Vangheluwe, J. De Lara, and P.J. Mosterman. Meta-modelling hybrid formalisms. In *International Symposium on Computer Aided Control Systems Design*, pages 65–70. IEEE, 2004.

[Lakshmanan *et al.*, 2010] Karthik Lakshmanan, G. Bhatia, and R. Rajkumar. Integrated end-to-end timing analysis of networked autosar-compliant systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 331–334, 2010.

[Lara and Vangheluwe, 2002] Juan De Lara and Hans Vangheluwe. Using AToM^3 as a Meta-CASE Tool. In *ICEIS*, 2002.

[Lee and Seshia, 2011] E. Lee and S. Seshia. *Introduction to embedded systems, a cyber-physical systems approach*. 1.3 edition, 2011.

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

[Makowitz and Temple, 2006] R. Makowitz and C. Temple. Flexray-a communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems*, pages 207–212, 2006.

[Mannadiar and Vangheluwe, 2011] Raphael Mannadiar and Hans Vangheluwe. Modular artifact synthesis from domain-specific models. *Innovations in Systems and Software Engineering*, 8(1):65–77, September 2011.

[Mannadiar, 2012] Raphael Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling*. PhD thesis, McGill, 2012.

[MathWorks, 2013] MathWorks. Mathworks website. http://www.mathworks.com, 2013.

[Mattsson *et al.*, 1997] Sven Erik Mattsson, Hilding Elmqvist, and Jan F. Broenink. Modelica : An International Effort to Design the Next Generation Modelling Language. *Benelux Quarterly Journal on Automatic Control*, 38(3), 1997.

[Mcquillan and Power, 2005] Jacqueline A. Mcquillan and James F. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, National University of Ireland, Maynooth, 2005.

[Mens and Vangorp, 2006] Tom Mens and Pieter Vangorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[Metzner and Herde, 2006] Alexander Metzner and Christian Herde. RTSATAn Optimal and Efficient Approach to the Task Allocation Problem in Distributed Architectures. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 147–158. IEEE, 2006.

[Minas, 2002] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[Mosterman and Vangheluwe, 2004] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Transactions of the SCS: Simulation*, 80(9):433, September 2004.

[Mosterman *et al.*, 2011] P.J. Mosterman, S. Prabhu, and T. Erkkinen. An industrial embedded control system design process. *Proceedings of the Canadian Engineering Education Association*, 2011.

[Mosterman, 2007] Pieter J. Mosterman. Hybrid dynamic systems: Modeling and execution. In Paul A. Fishwick, editor, *Handbook of Dynamic System Modelling*. CRC Press, 2007.

[Mosterman, 2011] Pieter J. Mosterman. Keynote: Opportunity in Embracing Imperfection : Is simulation the real thing ? In *Spring Simulation MultiConference*, 2011.

[Murata, 1989] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4), 1989.

[Neema *et al.*, 2003] Sandeep Neema, Janos Sztipanovits, Gabor Karsai, and K. Butts. Constraint-based design-space exploration and model synthesis. In *Embedded Software*, pages 290–305. Springer, 2003.

[Offutt, 1999] J Offutt. Generating tests from UML specifications. *Architectural Design*, 1999.

[Oldevik, 2005] Jon Oldevik. Transformation composition modelling framework. *Distributed Applications and Interoperable Systems*, pages 108–114, 2005.

[OMG, 2008a] OMG. MOF Model to Text Language (MTL). Technical report, OMG, January 2008.

[OMG, 2008b] OMG. Software & systems process engineering metamodel specification. www.omg.org/spec/SPEM/2.0/, 2008.

[OMG, 2010] OMG. OMG Systems Modeling Language (OMG SysML). Technical Report June, OMG, 2010.

[OSEK, 2005] OSEK. Osek operating system 2.2.3. www.osek-vdx.org, 2005.

[Palencia and Gonzalez Harbour, 1998] J.C. Palencia and M. Gonzalez Harbour. Schedulability analysis for tasks with static and dynamic offsets. *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 26–37, 1998.

[Pasareanu *et al.*, 2009] CS Pasareanu, Johann Schumann, and Peter Mehlitz. Model based analysis and test generation for flight software. *SMC-IT*, 2009.

[Paynter, 1961] H.M. Paynter. *Analysis and Design of Engineering Systems*. MIT Press, 1961.

[Pelz *et al.*, 2005] Georg Pelz, Peter Oehler, Eliane Fourgeau, and Christoph Grimm. Automotive System Design and AUTOSAR. In *Advances in Design and Specification Languages for SoCs*, pages 293–305. 2005.

[Perrone *et al.*, 2008] Ricardo Perrone, Raimundo Macedo, George Lima, and Veronica Lima. Estimating execution time probability distributions in component-based real-time systems. In *Workshop on component-based real-time systems*, 2008.

[Pimentel *et al.*, 2007] Andy D. Pimentel, Mark Thompson, Simon Polstra, and Cagkan Erbas. Calibration of Abstract Performance Models for System-Level Design Space Exploration. *Journal of Signal Processing Systems*, 50(2):99–114, June 2007.

[Pimentel, 2008] Andy D. Pimentel. The Artemis workbench for system-level performance evaluation of embedded systems. *International Journal of Embedded Systems*, 3(3):181, 2008.

[Pop *et al.*, 2002] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 187–192. ACM, 2002.

[Pop, 2007] Traian Pop. *Analysis and Optimisation of Distributed Embedded*. PhD thesis, linköping, 2007.

[Popovici *et al.*, 2008] Katalin Popovici, Xavier Guerin, Frederic Rousseau, Pier Stanislao Paolucci, and Ahmed Amine Jerraya. Platform-based software design flow for heterogeneous MPSoC. *ACM Transactions on Embedded Computing Systems*, 7(4):1–23, July 2008.

[Posse *et al.*, 2002] Ernesto Posse, J. De Lara, and Hans Vangheluwe. Processing causal block diagrams with graph-grammars in AToM^3. In *European Joint Conference on Theory and Practice of Software (ETAPS), Workshop on Applied Graph Transformation (AGT)*. Citeseer, 2002.

[Prabhu and Mosterman, 2004] S. M. Prabhu and Pieter J. Mosterman. Model-Based Design of a Power Window System: Modeling, Simulation and Validation. In *Proceedings of IMAC-XXII*, 2004.

[Press *et al.*, 1992] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. Numerical recipes in c: the art of scientific computing. 2. *Cambridge: CUP*, 1992.

[Rivera *et al.*, 2009] J.E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL model transformations. In *Proceedings of mtATL*, 2009.

[Rolland, 1998] C. Rolland. A Comprehensive View of Process Engineering. *Advanced Information Systems Engineering*, (June), 1998.

[Rozenberg and Ehrig, 1999] G. Rozenberg and H. Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume 1. World Scientific, 1999.

[Russell *et al.*, 2006] Nick Russell, Wil van der Aalst, Arthur ter Hofstede, and Petia Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Conceptual Modelling 2006: Proceedings of the APCCM2006*, pages 16–19, 2006.

[Sangiovanni-Vincentelli and Di Natale, 2007] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded System Design for Automotive Applications. *Computer*, 40(10):42–51, October 2007.

[Saxena and Karsai, 2010] Tripti Saxena and Gabor Karsai. MDE-based approach for generalizing design space exploration. In *Model Driven Engineering Languages and Systems*, pages 46–60. Springer, 2010.

[Schätz *et al.*, 2010] B. Schätz, Florian Hölzl, and Torbjörn Lundkvist. Design-Space Exploration through Constraint-Based Model-Transformation. In *17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, pages 173–182. IEEE, 2010.

[Seibel *et al.*, 2012] Andreas Seibel, Regina Hebig, Stefan Neumann, and Holger Giese. A Dedicated Language for Context Composition and Execution of True Black-Box Model Transformations. In *Software Language Engineering*, pages 19–39, 2012.

[Sinnen, 2007] Oliver Sinnen. *Task scheduling for parallel systems*. Wiley-Interscience, 2007.

[Spitznagel and Garlan, 1998] Bridget Spitznagel and David Garlan. Architecture-based performance analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, pages 19–20, 1998.

[Syriani and Vangheluwe, 2010] Eugene Syriani and Hans Vangheluwe. De- / Reconstructing Model Transformation Languages. *Electronic Communications of the EASST Ninth International Workshop on Graph Transformation and Visual Modeling Techniques*, 29, 2010.

[Syriani, 2011] Eugene Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, McGill University, 2011.

[Tindell and Burns, 1992] KW Tindell and A Burns. Allocating hard real-time tasks: an NP-hard problem made easy. *Real-Time Systems*, 1992.

[Tindell and Clark, 1994] K. Tindell and J. Clark. Holistic Schedulability analysis for Distributed Hard Real-Time Systems. *Microprocessing and microprogramming*, 40(2-3):117–134, 1994.

[Törngren *et al.*, 2006] Martin Törngren, Dan Henriksson, Ola Redell, Christoph Kirsch, J. El-Khoury, Daniel Simon, Yves Sorel, Hanzalek Zdenek, and K.E. Årzén. *Co-design of Control Systems and Their Real-time Implementation: A Tool Survey*. Department of Machine Design, Royal Institute of Technology, 2006.

[Van Gorp *et al.*, 2004] P. Van Gorp, D. Janssens, and T. Gardner. Write once, deploy N: a performance oriented MDA case study. *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004.*, (Edoc):123–134, 2004.

[Vangheluwe and de Lara, 2004] Hans Vangheluwe and J de Lara. Domain-specific modelling with AToM^3. In *4th OOPSLA Workshop on Domain-Specific Modelling*, 2004.

[Vangheluwe and Vansteenkiste, 1996] Hans Vangheluwe and Ghislain Vansteenkiste. A Multi-Paradigm Modelling and Simulation Methodology : Formalisms and Languages. *Simulation in Industry*, 1996.

[Vangheluwe *et al.*, 2002] Hans Vangheluwe, Juan De Lara, and Pieter J. Mosterman. Multi-Paradigm Modelling and Simulation. In *AI, Simulation and Planning in High Autonomy Systems*, 2002.

[Vangheluwe, 2000] Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design (CASC)*, pages 129–134. Ieee, 2000.

[Vangheluwe, 2012] Hans Vangheluwe. The Discrete EVent System specification (DEVS) formalism. Technical report, 2012.

[Vangheluwe, 2013] Hans Vangheluwe. MSDL projects. DEVS project page. `http://msdl.cs.mcgill.ca/projects/projects/DEVS/`, 2013.

[Vanhooff and Baelen, 2006] Bert Vanhooff and Stefan Van Baelen. Towards a transformation chain modeling language. *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2006.

[Vanhooff *et al.*, 2007] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers. Uniti: A unified transformation infrastructure. *Model Driven Engineering Languages and Systems*, pages 31–45, 2007.

[Wagelaar, 2006] Dennis Wagelaar. Blackbox composition of model transformations using domain-specific modelling languages. In *Workshop on Composition of Model Transformations*, 2006.

[Wainer *et al.*, 2005] G. Wainer, E. Glinsky, and P. MacSween. A Model-Driven Technique for Development of Embedded Systems Based on the DEVS formalism. *Model-Driven Software Development*, pages 363–383, 2005.

[Wainer, 2009] Gabriel Wainer. *Discrete-Event Modeling and Simulation: A Practitioner s Approach*. CRC press, 2009.

[Wilhelm *et al.*, 2008] Reinhard Wilhelm, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, and Reinhold Heckmann. The worst-case execution-time problem: overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, April 2008.

[Wirsing *et al.*, 2006] Martin Wirsing, Juan Bicarregui, Ed Brinksma, Simon Dobson, Peter Druschel, Pierre Fraignaud, Fausto Giunchiglia, Manuel Hermenegildo, Helen Karatza, Stephan Merz, Joseph Sifakis, Mikhail Smirnov, Christian Tschudin, and Franco Zambonelli. Software Intensive Systems. 2006.

[Zheng *et al.*, 2007] Wei Zheng, Qi Zhu, Marco Di Natale, and Alberto Sangiovanni Vincentelli. Definition of Task Allocation and Priority Assignment in Hard Real-Time Distributed Systems. *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, pages 161–170, December 2007.