System Design Support of Cyber-Physical Systems

A co-simulation and co-modelling approach

YUNYUN NI

System Design Support of Cyber-Physical Systems

A co-simulation and co-modelling approach

Y. Ni

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering Mathematics and
Computer Science, University of Twente

Robotics and Mechatronics Group

CTIT PhD Thesis Series No. 15-355
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2010-2012) under grant agreement no. INFSo-ICT-248134 DESTECS (Design Support and Tools for Embedded Control Systems).

Title: System Design Support of Cyber-Physical Systems
A co-simulation and co-modelling approach
Author: Y. Ni
ISBN: 978-90-365-3855-8
ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 15-355)
DOI: 10.3990/1.9789036538558

Cover design by Gilberto Bradford and Yunyun Ni

Copyright © 2015 by Y. Ni, Enschede, The Netherlands.

All rights reserved. No part of this publication may be reproduced by print, photocopy or any other means without the prior written permission from the copyright owner.

Printed by Ipskamp Drukkers

SYSTEM DESIGN SUPPORT OF CYBER-PHYSICAL SYSTEMS

A CO-SIMULATION AND CO-MODELLING APPROACH

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op 18 Juni 2015 om 12.45 uur

door

Yunyun Ni
geboren op 26 Mei 1983
te Zhejiang, P.R. China

Dit proefschrift is goedgekeurd door:

prof.dr.ir. S. Stramigioli, promotor

dr.ir. J.F. Broenink, co-promotor

Graduation committee

Chairman and Secretary

prof.dr.ir. P.M.G. Apers University of Twente

Supervisor

prof.dr.ir. S. Stramigioli University of Twente

Co-Supervisor

dr.ir. J.F. Broenink University of Twente

Members

prof.dr.ir. C.H. Slump University of Twente

prof.dr.ir. G.J.M. Smit University of Twente

dr.ir. J.P.M. Voeten Eindhoven University of Technology

prof.dr. H. Vangheluwe University of Antwerp

Summary

Cyber-physical Systems (CPS) are present in our everyday life. Along the increasing complexity of the devices, the process of designing CPS devices is also becoming more challenging. Due to the nature of CPS, it is no longer sufficient to isolate the components of CPS and study them individually, because the cyber part and physical part interact with each other. However, the current situation is that often engineers are not trained to have a mindset of considering systems as a whole, but instead have a more specialized knowledge in a specific discipline. This situation also applies to the existing theory development. In practise, ad hoc approaches are often adopted in the design process. This often leads to expensive rework later or even possibly a severe design flaw which might be fatal for the system itself.

The goal of this research is to support system-level design for CPS devices from methods perspective with corresponding tooling support to bridge the existing design gap.

In this work, a set of methods are provided that support different expertise to understand CPS design from a system level, instead of only considering one single specific discipline. In order to avoid confusions across the different domains, a list of explained terms is provided. Co-design support guidelines (co-design flows) that consider different backgrounds of the possible designers and different system properties are discussed in detail. Following the step-wise refinement design steps, a CPS is firstly modelled in a top-level model, then detailed out with different focuses of the interest: such as dynamic-behaviour oriented, control-logic oriented and contract oriented.

A way of working, to reduce unnecessary design iterations and help engineers to structure the cyber part software in a way that the whole system can act more robust, is also discussed. This includes a general method of modelling the possible faults using a layered controller structure. From these two aspects, the resulting system design is made more robust (fault-tolerant).

From a tooling support aspect, a design-support software framework is introduced. A co-simulation framework is presented that supports expertise from different domains to work together, which can enable early stage testing to avoid high expenses (both money and time wise) in the development cycle. This framework combines two domain-specific tools with an appropriate synchronization scheme for the co-simulation engine. Additionally, a domain-specific scripting language (DCL) is introduced to ease the change of a co-model scenario during the co-simulation, such as fault injections. With the aid of design space exploration (DSE) tool support, system designers can make better early-stage design choices.

The methods and the tooling support introduced in this work are demonstrated in two different case studies with different focuses: the mobile robot case study aims to demonstrate the design space exploration facility; the slider setup is focussed on demonstrating fault-tolerant mechanisms. Besides these two case studies, the meth-

ods and tooling have been also tested widely in industrial consortium of the DESTTECS project.

It is recommended that the methods and tooling are applied in other application domains, such as medical healthcare devices, automotive industries, etc. Cyber-Physical Systems is still a relatively young and changing research area. This work is only a little step in the CPS system-level design support, a lot of work still needs to be done. Thinking of CPS devices on a larger scale and higher complexity can bring us to further integration of a wider range of models.

Samenvatting

Cyber-physical systems (CPS) zijn alom aanwezig in ons dagelijks leven. Naast de toenemende complexiteit van deze apparaten wordt ook het ontwerpproces van CPS alsmaar uitdagender. Vanwege de aard van CPS is het niet langer toereikend om de componenten van een CPS te isoleren en afzonderlijk te bestuderen, omdat het “cyber” gedeelte en het fysieke gedeelte elkaar beïnvloeden. Echter, ingenieurs zijn zelden opgeleid om een systeem als een geheel te beschouwen, maar hebben juist gespecialiseerde kennis van een specifiek vakgebied. Deze situatie doet zich ook voor op het gebied van theoretische ontwikkelingen. In de praktijk worden vaak ad hoc oplossingen toegepast gedurende het ontwerpproces. Dit leidt vaak tot duur nawerk of mogelijk zelfs tot serieuze ontwerpfouten die mogelijk fataal zijn voor het systeem.

Het doel van dit onderzoek is om te voorzien in een *system-level* ontwerpondersteuning voor CPS vanuit een methodologisch oogpunt, met daarnaast ondersteunde *tooling* om zo het bestaande gat in het ontwerpproces te dichten.

In dit werk wordt een methodiek gepresenteerd die diverse disciplines voor het begrip van CPS ontwerp op systeem niveau ondersteunt, in plaats van een focus op één enkele discipline. Om verwarring tussen de diverse domeinen te voorkomen is in een begrippenlijst opgenomen. Ondersteunende *co-design* richtlijnen (*co-design flows*) die de diverse achtergronden van de ontwerpers en de diverse systeemeigenschappen in overweging nemen worden in detail bediscussieerd. Middels stapsgewijze verfijning wordt een CPS eerst gemodelleerd op *top-level* niveau en daarna wordt het model gedetailleerder uitgewerkt met focus op afwisselende interessegebieden, zoals dynamisch gedrag, regellogica en contracten.

Een werkwijze die onnodige ontwerpiteraties vermindert en ingenieurs hulp biedt bij het structureren van de software, op zodanige wijze dat het gehele systeem robuuster wordt, wordt ook besproken. Dit omvat een algemene methodiek voor het modelleren van mogelijke fouten met behulp van een gelaagde structuur voor een regelaar. Met behulp van deze aspecten kan het systeemontwerp robuuster (fouttolerant) gemaakt worden.

Op het gebied van *tooling* wordt een software framework voor de ondersteuning van het ontwerpproces gepresenteerd. Een *co-simulation framework*, dat ondersteuning biedt voor samenwerking tussen diverse domeinen, wordt gepresenteerd met als doel om vroegtijdig testen mogelijk te maken en hoge kosten (zowel in tijd als geld) te voorkomen. Dit framework combineert twee domeinspecifieke tools met behulp van een synchronisatiemechanisme voor de *co-simulation engine*. Daarnaast wordt een domeinspecifieke scripttaal (DCL) geïntroduceerd, die als doel heeft om het wijzigen van het *co-model* scenario tijdens de *co-simulation* te vergemakkelijken, om zo bijvoorbeeld foutinjectie te simuleren. Met behulp van tools voor *design space exploration* kunnen ontwerpers betere keuzes maken in het begin van het ontwikkelproces.

De methodes en de gereedschappen die in dit proefschrift geïntroduceerd zijn, wor-

den gedemonstreerd in twee verschillende gebruikersscenario's, met elk een andere focus: de mobiele robot studie heeft als doel om de faciliteiten voor het verkennen van de ontwerpruimte te demonstreren; de *slider* opstelling is gericht op het demonstreren van fouttolerante mechanismen. Naast deze twee studies zijn de methoden en tools ook uitgebreid getest in het industriële consortium van het DESTecs project.

Het is aanbevolen om de methodieken en gereedschappen te gebruiken in andere toepassingsdomeinen, zoals medische apparaten, autoindustrie, etc. Het onderzoeksgebied van *cyber-physical systems* is nog relatief jong en veranderlijk. Dit onderzoek is slechts een kleine stap in de ondersteuning voor CPS systeemontwerp, met nog veel werk dat nog gedaan moet worden. Het nadenken over CPS op een grotere schaal en met grotere complexiteit kan ons helpen bij verdere integratie van een diverser aantal modellen.

This book is dedicated to my grandpa
"You will always be in my memory".

谨以此书献给我敬爱的爷爷！

Contents

- 1 Introduction 1**
 - 1.1 Background 1
 - 1.2 Goals 4
 - 1.3 Contributions 4
 - 1.4 Scope of the Thesis 5
 - 1.4.1 The DESTECS project 6
 - 1.5 List of Publications 6
 - 1.5.1 Peer Reviewed Publications 6
 - 1.5.2 Project Deliverables 7
 - 1.6 Organization 7
- 2 Terminology and Related work 9**
 - 2.1 Basic Concepts for CPS Modelling and Simulation 9
 - 2.2 Terms for Co-Modelling 12
 - 2.3 Terms for Co-Simulation 13
 - 2.4 Language Concepts 14
 - 2.4.1 Bond Graphs 14
 - 2.4.2 VDM 15
 - 2.4.3 Tools Used 15
 - 2.5 Advanced Topics 16
 - 2.5.1 Design Space Exploration 16
 - 2.5.2 Fault Modelling 17
- 3 Co-Simulation Framework 19**
 - 3.1 Introduction 19
 - 3.2 Related Work 20
 - 3.3 Co-Simulation Synchronization Scheme 21
 - 3.4 Scenario Support: Script Language 24
 - 3.4.1 Design Rationale 24
 - 3.4.2 Main Features 25
 - 3.4.3 A Small Example 26

3.5 Conclusion	29
4 Co-Design and Co-Modelling	31
4.1 Co-Design Workflow	31
4.2 Dynamic-Behaviour Oriented Co-Modelling Approach	35
4.2.1 Plant Modelling	37
4.2.2 Controller Design	40
4.2.3 Co-Simulation	43
4.3 Controller-Logic Oriented Co-Modelling Approach	44
4.3.1 Plant Modelling	46
4.3.2 Controller Design	47
4.3.3 Co-Simulation	49
4.4 Co-simulation Step	49
4.5 Contract Oriented Co-Modelling Approach	53
4.6 Conclusion	54
5 Co-Modelling Advanced Topics	57
5.1 Design Space Exploration (DSE)	57
5.1.1 Design Space Exploration and ACA Aspects	58
5.1.2 Tooling Support Detail of DSE and Discussions	60
5.2 Fault Modelling and Fault-Tolerance Mechanism	61
5.2.1 Fault Selection and Modelling	61
5.2.2 Fault Modelling in Overture	62
5.2.3 Fault Modelling in 20-sim	64
5.2.4 Fault-Tolerant Cyber Software Design	65
5.3 Conclusions	67
6 Case Studies	69
6.1 Case 1: Mobile Robot	69
6.1.1 Goal	69
6.1.2 Setup Description	69
6.1.3 Top-Level Model	70
6.1.4 Plant Model (CT Model)	71
6.1.5 Controller Model (DE Model)	75
6.1.6 Design Space Exploration	76

6.1.7 Conclusion and Evaluation	81
6.2 Case 2: Slider Setup	82
6.2.1 Goal	82
6.2.2 Setup Description	82
6.2.3 Top-Level Model	83
6.2.4 Plant Model (CT Model)	84
6.2.5 Controller Model (DE Model)	85
6.2.6 Fault Modelling and Fault Tolerance Mechanisms	86
6.2.7 Results Analysis	89
6.2.8 Conclusion and Evaluation	90
6.3 Industrial case studies in the DESTecs project	90
7 Conclusion and Outlook	93
7.1 Contributions	93
7.2 Recommendations for Future Work	94
7.3 Final Word	95
A DCL Language Definition	97
A.1 DCL Language Constructs	97
A.1.1 Script	97
A.1.2 Top-Level Statements	97
A.1.3 Statements	98
A.1.4 Types and Expressions	100
A.2 Abstract Syntax and Context Conditions	102
A.3 Concrete Syntax	106
A.3.1 Script	107
A.3.2 Top-Level Statements	107
A.3.3 Statements	107
A.3.4 Expressions	108
Bibliography	111
Index	118
Acknowledgement	120

1

Introduction

1.1 Background

"Cyber-Physical Systems" (CPS) literally mean systems that combine "Cyber" and "Physical" parts. "Cyber" is a prefix derived from "Cybernetic", which comes from the Greek adjective κυβερνητικός meaning skilled in steering or governing (Liddell et al., 1996). Hence it stands for the parts that controls the physical parts, while "physical" here stands for the parts of the physical universe that is of interest to analysis. "Cybernetic" was first promoted by Wiener (1947) as *"the scientific study of control and communication in the animal and the machine"*. The name CPS sounds sophisticated, however, CPS devices can actually be seen everywhere in our daily life, such as industrial robots, auto-mobiles, medical health devices and consumer appliances.

Figure 1.1: CPS current role (DFKI, 2011)

The process of designing CPS devices is a challenging process by its nature: the degree of complexity of the current CPS domain is higher than that of previous indus-

try generations (the current generation is often associated with the name "Industry 4.0" (DFKI, 2011)) as shown in Figure 1.1.

We can no longer only try to understand the cyber and physical parts of the system separately, since these two parts interact with each other. It would not be sufficient to only be concerned with whether the software could crash, but whether the software feeds the right control input into the physical system that it is trying to control and what effect those inputs will have on the physical side as well (see Figure 1.2 as an overview of a CPS architecture). Likewise, it is not sufficient to understand the physical-system dynamics alone, as its effect also depends on the control input generated by the cyber side (computer program).

Figure 1.2: CPS architecture overview

Often, a CPS requires expertise from many domains, e.g., electrical engineering (robotics, control systems), computer science (software engineering, networks communication engineering) and mechanical engineering etc. Furthermore, the market demands rapid innovation and assessment of designs; new products with novel functionality and an acceptable reliability need to reach the market before other competitors bring their products.

However, engineers working in these domains are not trained to pay more attention to systems as a whole (Mosterman et al., 2013); on the contrary, the more higher education they receive, the more specialized they are in a specific discipline. Furthermore, the current theory development in the CPS area (Schutter et al., 2009) also has its own limitations. In detail, the current existing theory (hybrid automaton, switched systems, mixed logical dynamic models etc.), only deal with specific aspects. The subsequent modelling and design problems become too complex computation-wise to be handled. A new theory is needed to elaborate methods that allow to introduce combined continuous-discrete phenomena in a more general sense, which means that it cannot be based on the assumption of a Lipschitz continuous vector field (O'Searcoid, 2006), as CPS has its discontinuities by nature.

CPS design normally follows the workflow as indicated at left side of Figure 1.3, which is an ad-hoc approach in practise (Edwards et al., 1997), where the system-level integration is performed in a late design stage. Many design choices are made implic-

itly, mostly depending on the individual personal experience of the lead systems engineers. System-level decision making is difficult if the rationale behind such decisions are not quantified. It is, therefore, necessary to make design choices explicitly in order to enable the requirements for CPS being preserved well at the system level.

Figure 1.3: CPS design workflow and duration without integrated models (Left) and with integrated models (Right) (Groothuis and Broenink, 2010)

Due to the ad-hoc approach and late integration of the design workflow, many (design) errors are detected during integration. These errors often lead to expensive rework later; or even possibly severe design flaws which might be fatal for the system itself. In addition, the later an error is detected, the more costly it will be to solve it. This certainly leads to many substantial design iterations, which increases the duration of the design time.

It is shown that cyber-physical systems design is a challenging process due to its heterogeneous nature. The efficient design of CPS is hampered by the separation of engineering disciplines in current state-of-the-art development approaches. A set of methods to address system-level design issues across discipline boundaries is lacking (Lee, 2006).

This "design gap" inhibits iterative and concurrent engineering, leading to sub-optimal designs and long development lead times. Moreover, the design gap is widening because of increasing system complexity and increasing capabilities of the system artefacts used.

Additionally, the current theory development in the CPS domain and the domain-specific education system did not help to ease the situation that CPS should be considered as a whole as early as possible in the design phase.

1.2 Goals

This thesis is focused on two main objectives from a method-support aspect. These objectives aim to bridge the design gap and give CPS system-design support, especially from making a rational system-level design decision and designing reliable software that is fault-tolerant. The resulting products of CPS design are less error-prone, and more robust to perturbations.

- *Goal 1:* to provide a set of methods together with a co-design support software framework to enable early-stage testing to avoid high expenses (both money and time wise) in the development cycle. With the assistance of this framework, the engineers can have a practical tool that can facilitate the system-level design of CPS. A set of methods that facilitate different engineering domains to understand CPS design from a system level, instead of only considering one single specific discipline. Engineers can make explicit design decisions, with all this support, based on concrete results instead of merely based on personal experience.
- *Goal 2:* to provide a way of working that can help engineers to structure the software better in a way that the whole system can act more robust against unexpected disturbances. This way of working should support the cyber part design to be "First-Time-Right" (Bezemer, 2013; Brodskiy, 2014) and it also results in shortening design cycles.

1.3 Contributions

The contributions of this work are summarized as follows:

- *Design Workflow:* based on different computational domains, components of CPS devices are modelled in the most suitable formalism. Engineers can have a set of hands-on high-level system co-models during the entire design process. The guidelines of the system models are demonstrated in both case studies.
- *Design Space Exploration:* with the aid of the method and tool support of design space exploration, system designers can make rational early-stage design choices. This method is demonstrated in the first case study.
- *Fault-Tolerance Mechanism:* a general method of modelling possible faults using a layered controller structure to make the design more robust (fault-tolerant). This method is demonstrated in the second case study.
- *A co-simulation software framework:* combines two domain-specific tools with an appropriate synchronization scheme for the co-simulation engine is described. Additionally a domain-specific scripting language is designed and implemented to ease the change of a co-model scenario during the co-simulation.

The first three bullets are all from a co-modelling methods perspective to provide guidance for engineers to have the mindset of how a co-design flow works using the pro-

posed co-simulation framework. The design workflow together with design space exploration are corresponding to *Goal 1*, while fault-tolerant mechanism is corresponding to *Goal 2*. The co-simulation framework plus the scripting language are corresponding to both *Goal 1* and *Goal 2*, as the software support is the base to test the proposed design methods.

1.4 Scope of the Thesis

An illustration of system design support for CPS mentioned in this thesis is shown in Figure 1.4. Block 1 in Figure 1.4 shows the modelling step of this iterative design process. The designer creates models, reusing existing components and sub-models where possible. Inputs are defined, allowing scenarios to be realised during co-simulation. The principles of Block 1 are explained in Chapter 4. The designer can also model non-normal behaviour (faults) to prevent faults, explained in Chapter 5. The engineer creates scenarios (using a graphical user interface where appropriate) to configure one or more co-simulations (Block 2), which are then executed under the control of the co-simulation engine (Block 3). The details about Block 2 and Block 3 are discussed in Chapter 3. The designers can use tools to help visualise and evaluate the test results for the rational design decision making, by using the drawing of time plots or creation of 3D animations as shown in Block 4.

Figure 1.4: CPS design process with aid of co-simulation and co-modelling approach (Broenink et al., 2012).

1.4.1 The DESTECS project

The DESTECS (Design Support and Tooling for Embedded Control Software)¹ project is a EU FP7 project that has been researching and developing methods and open tools that support the collaborative design of dependable real-time embedded control systems using a model-driven approach. It covers continuous-time modelling, discrete-event modelling of controllers and architectures, fault modelling and tolerance, support for iterative design evolution, and open tool frameworks. Bringing this together in methods and tools improves the cost-effectiveness of model-based design. In the DESTECS project, two domain-specific languages/tools were involved: Vienna Development Method (VDM) (Bjoerner and Jones, 1978) and 20-sim (Broenink, 1997).

Since the author's PhD work was based on the participation in the DESTECS project, the content of this thesis is constrained when it comes to the preference of languages/-tools. However, the methods and tooling introduced in the thesis have also been tested by the DESTECS industrial partners.

1.5 List of Publications

The content of this thesis is based on the following publications.

1.5.1 Peer Reviewed Publications

- P.1 Ni, Y. and J. F. Broenink (2014), A co-modelling method for solving incompatibilities during co-design of mechatronic devices, *Journal: Advanced Engineering Informatics*, 5 years Thomson Reuters Impact factor: 2.126, **vol. 28**, pp. 232–240, doi: 10.1016/j.aei.2014.05.004
- P.2 Ni, Y. and J. F. Broenink (2012), Hybrid systems modelling and simulation in DESTECS: a co-simulation approach, in *26th European Simulation and Modelling Conference, ESM 2012, Essen, Germany*, Ed. M. Klumpp, EUROSIS-ETI, Ghent, Belgium, pp. 32–36, ISBN-13 978-90-77381-73-1
- P.3 Broenink, J. F. and Y. Ni (2012), Model-Driven Robot-Software Design using integrated Models and Co-Simulation, in *International Conference on Embedded Computer Systems, SAMOS 2012, Samos, Greece*, Eds. J. McAllister and S. Bhattacharyya, IEEE Computer Society, USA, pp. 339–344, ISBN-13 978-1-4673-2296-6
- P.4 Pierce, K., C. Gamble, Y. Ni and J. F. Broenink (2012), Collaborative Modelling and Co-simulation with DESTECS: A Pilot Study, in *WETICE*, pp. 280–285
- P.5 Broenink, J. F., Y. Ni and M. A. Groothuis (2010d), On Model-driven Design of Robot Software using Co-simulation, in *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, Ed. E. Menegatti

¹www.destecs.org

1.5.2 Project Deliverables

- D.1 Broenink, J. F., J. Fitzgerald, C. Gamble, C. Ingram, A. Mader, J. Marincic, Y. Ni, K. Pierce and X. Zhang (2012), EU7 Project deliverable 2.3 — Methodological Guidelines 3, Technical report, The DESTecs Project (INFSO-ICT-248134)
- D.2 Ni, Y., J. F. Broenink, A. Ribeiro, K. G. Lausdahl, F. Groen, M. Groothuis, K. Pierce, C. Gamble and P. G. Larsen (2011), EU7 Project deliverable 3.3a — Design Space Exploration Tool Support, Technical report, The DESTecs Project (INFSO-ICT-248134)
- D.3 Lausdahl, K. G., A. Ribeiro, P. Visser, F. Groen, Y. Ni, J. F. Broenink, A. Mader, J. W. Coleman and P. G. Larsen (2011), EU7 Project deliverable 3.3b — Co-simulation Foundations, Technical report, The DESTecs Project (INFSO-ICT-248134)
- D.4 Larsen, P. G., C. Kleijn, K. G. Lausdahl, A. Ribeiro and Y. Ni (2011), EU7 Project deliverable 3.3c — User Manual, Technical report, The DESTecs Project (INFSO-ICT-248134)

1.6 Organization

This thesis is focused on the design support of CPS by adopting a co-simulation and co-modelling approach. A co-simulation framework to aid the system design process of CPS is introduced. Essential principles of co-modelling are introduced, as well fault modelling and fault prevention. In addition to that, details of the case study are introduced.

A set of terms used in this thesis is provided in Chapter 2, together with a review of related work.

Details about the co-simulation framework including co-simulation synchronization scheme and the scripting language are described in Chapter 3. The chapter is based on the publications P.2, P.5, D.1 and D.3.

Details about co-design flow and co-modelling techniques are introduced in Chapter 4. This chapter is based on publications P.3 and D.4. Advanced modelling topics, such as design space exploration and fault modelling are introduced in Chapter 5. This chapter is based on publications P.1 and D.2.

In Chapter 6, two case studies illustrate the methods and tool framework introduced in previous chapters with the focus on the advanced topics. One case study focuses on demonstrating design space exploration, while the other one focuses on fault-tolerance mechanism design for controller software. Results of the two case studies have been published in publications P.1 and P.4.

In the end, conclusions and possible future work are given in Chapter 7.

2

Terminology and Related work

As CPS design is a multi-disciplinary subject, it has occurred that basic common terms often have a completely different meaning for experts from different domains (Broenink et al., 2010a). This is the reason to have a separate chapter about important terms which are used through this thesis.

There are quite some terms belonging to the same category while having different focuses, such as embedded control systems, hybrid systems. Previously, *embedded control system* (Heath, 2002) is the term widely used for devices nowadays called CPS. However comparing to traditional embedded control system, CPS in full potential is designed as a network of interacting elements with I/O components shown in Figure 1.2. *Hybrid system theory* (Henzinger, 1996; Alur et al., 1995) is one of the theories that studies the properties of CPS.

In this chapter, first the basic terms that are related with modelling and simulation are introduced, followed by terms that are related with co-simulation and co-modelling.

2.1 Basic Concepts for CPS Modelling and Simulation

A *model* is an abstraction of a system under study, which captures only relevant aspects that are of interest. The activity of creating models is called *modelling*. A system here can be a single CPS of interest, more than one CPS or a partition of the whole CPS. The basic modelling goal is to derive a *competent model* that is as simple as possible yet sufficient to give the information required of it. To judge whether a particular model is competent, mostly, the model is *verified* using simulations. A symbolic execution of a model is called a *simulation*. A model which can be simulated is called an *executable model*.

During the modelling and simulation study of CPS, we start with the motivation of better understanding of the CPS under study. *Verification* is the process to check whether a model implementation and its associated data accurately represent the developer's conceptual description and specifications. *Validation* is the process to check the degree to which a simulation model and its associated data are an accurate representation of the real world from the perspective of the intended use of the model, this is comparing the results from the simulation experiments with the CPS prototype outputs. The results from the simulation experiments can also be compared with the

initial requirements to see whether this is the model that is initially wanted for that application.

Following the general understanding of the CPS modelling and simulation process and essential activities, the details about the model content are explained. There are some general goals of models, such as:

- understand and predict the dynamic behaviour of the CPS under study, which requires a rather detailed model with sophisticated resemblance to the physical behaviour;
- compute a reaction in the cyber part of the CPS under study (using the existing physical plant model).

Model

Figure 2.1: An illustration of a CPS model from a modelling aspect, while DAC stands for digital-to-analog converter, ADC stands for analog-to-digital converter

A CPS system model is shown in Figure 2.1. It has three essential components: *Cyber*, *I/O* and *Plant*. Everything outside of the system under study is considered as *environment*. Environment means that the parts that are not of interests for the particular study purpose. For example, if we study a mobile robot moving on a planar surface (See Chapter 6 case study 1), the surface is modelled inside the *Plant* model. The surface is not considered as *environment* in this case. The cyber part and the physical part directly influence each other by means of *actuators* and *sensors*.

From an execution aspect, to simulate a model, some essential elements such as: *stimuli* and *simulation result* as shown in Figure 2.2. *Stimuli* include those related to normal functioning of the system as well as *disturbances* which are stimuli that tend to deviate the plant from the desired behaviour. A *design parameter* is a property of a model that affects the behaviour it describes, but which remains constant during a given simulation. A *variable* is part of a model that may change during a given simulation.

Figure 2.2: An illustration of a model from an execution aspect

Besides the elements mentioned above, it is essential to notice the properties of the *computational domains* of CPS. By the nature of CPS, CPS contains more than one single computational domain, such as continuous time, discrete time and discrete event. In Ptolemy (Ptolemaeus, 2014), a model of computation is *a collection of rules that govern concurrent execution of the components and the communication between components*, while here we use three different computational domains standing for different data types that are based on time, such as continuous time, discrete time and discrete event.

Continuous time and *discrete time* are two different ways to model variables that evolve over time. Values of variables are viewed as non-discontinuous for potentially only an infinitesimally short amount of time in the *continuous time* (CT) domain, which makes use of *differential equations*. *Dynamic systems* are systems for which the variables are time-dependent. *Differential equations* are used to model continuous dynamic behaviour.

Values of variables are viewed as occurring at fixed, discrete, time intervals in the *discrete time* (DT) domain (e.g. van Amerongen, 2010). The described systems in such time form are called *discrete-time systems*; they make use of *difference equations*, e.g. used to describe control laws. The term continuous-time is often used in lieu of *analogue* (Ogata, 1995), while the term of discrete-time is in lieu of *digital*. The CT simulation is approximated by discretization of the time, which is actually converting the ordinary differential equations into difference equations. Therefore CT and DT are simulated in the same simulator later in the thesis.

Discrete-event domain (DE Simulation) views the points in time where the state of the system changes (e.g. Robinson, 2007). These state changes which occur at a particular instant in time are *events*. Discrete-event modelling is typically (Liu, 1998) used for digital hardware (Ashenden, 2001; Thomas and Moorby, 2008), communication systems (Banks et al., 2004), and embedded software (Chiodo et al., 1994). A specific example of a discrete event can be: at a certain time, a certain endstop (positioning

sensor) is touched by the moving slider (see Chapter 6), this sends a signal (a state) to the cyber part. Based on this state, the cyber part can decide to stop the slider motion.

2.2 Terms for Co-Modelling

Often the CPS design activities involve more than one person. *Collaborative* literally means two or more parties working together. Hence *collaborative modelling* stands for a modelling process which involves at least two designers. However, in this thesis, *co-modelling* (Combined Modelling) is a modelling process which involves at least two different modelling domains, but this activity can be done by one single designer who knows about the properties of the domains involved. This is a different term comparing to collaborative modelling. Our co-modelling approach is one of the ways to perform collaborative modelling.

We use the concept of co-model and co-simulation to express and execute CPS models (Broenink et al., 2010c) in this thesis.

Figure 2.3: Co-model from a modelling aspect

A *co-model* is a model which consists of two component models, one formulated in a DE formalism, the other in a CT formalism and one contract as shown in Figure 2.3. Shared variables, shared design parameters, and events define the nature of the communication between the two models. Shared variables and shared design parameters are recorded in a *contract*. Shared variables written by the DE constituent model are called *controlled* variables and those written by the CT constituent model are called *monitored* variables. An *event* is an action that is initiated in one model that leads to an action in the other model. Events that occur at a specific time are *time events*. Events that occur in response to a change in a model (when a specified variable reaches or passes a threshold) (Cellier, 1979) are named *state events*.

In the DESTECs project and thus in this thesis, CT models are represented using the bond-graph formalism (Paynter, 1961; Karnopp and Rosenberg, 1968; Breedveld, 2009) together with the graphical and equation-based models supported by 20-sim (Broenink, 1990, 1997; Kleijn, 2009). DE models are represented using the VDM

(Vienna Development Method) formalism (Bjoerner and Jones, 1978; Jones, 1990; Fitzgerald et al., 2008), which includes object-oriented and real-time extensions for modelling embedded software, supported by the Overture tool (Larsen et al., 2010)).

A *scenario* configures and governs the execution of a co-simulation (Figure 2.3) and consists of scripts. The *script* governs the co-simulation during execution by changing selected values in the co-model. Values can be changed at a given time, or in response to a change in the state of the co-model. Scripts are defined using a *command language* (a simple, domain-specific programming language, (Broenink et al., 2012)) and are contained in a *script file*. Scripts can be used for dynamic fault injection (activation of latent non-normative behaviour, details see Chapter 5 and Chapter 6) and for mimicking user inputs (activation of normative behaviour). The details about the language definitions, description and examples can be found in Chapter 3 and Appendix. A.

2.3 Terms for Co-Simulation

Co-simulation is a simulation that couples models from two different simulators. In this case, it couples two different computational domains: discrete-event (DE) and continuous-time (CT). The simulation of controller models and plant models is performed by the *discrete-event* (DE) simulator and *continuous-time* (CT) simulator, respectively. In a co-simulation, the DE and CT models execute in their own simulators with the steering of a co-simulation engine as shown in Figure 2.4 (DT models are also executed in the CT simulator here). This is the co-simulation framework choice made in this thesis.

There are also other ways instead of implementing a co-simulation engine. However, in this thesis, we only discuss the framework by implementing a co-simulation engine, as the thesis work is based on the DESTECs project work.

This co-simulation engine handles time, parameters and variables at the interface of DE and CT models.

Figure 2.4: Co-model and co-simulation from a tool aspect

The overall coordination and control of the co-simulation is through a *co-simulation*

engine. The co-simulation engine is responsible for the progress of time in the co-simulation and the propagation of information between the two constituent models. An executable co-model accept stimuli, formulated in scripts. The states of a co-model can be observed both during and after simulations. The output of a co-simulation is a *simulation result* which can be in different formats. This set of results can be observed during the run and analysed by means of post-processing of the simulation data after the run.

The details about the co-simulation framework are represented in Chapter 3.

2.4 Language Concepts

Due to the influence of the DESTecs project, in this thesis, *bond graphs* are chosen to describe CT models. The same reason as for CT formalism language applies for the DE formalism language as well. Here we use the Vienna Development Method (VDM) to formulate DE models. Similarly, the tools used have been also bounded by the languages chosen. 20-sim is used to perform CT simulation, while Overture is used to perform DE simulation.

2.4.1 Bond Graphs

A bond graph is a domain-independent description of physical-system dynamic behaviour. This means that systems from different physical domains (cf. electrical, mechanical, hydraulic, thermodynamic) are described in the same way. The basis of bond graphs is a representation of energy and energy exchange. Analogies between domains are more than just equations being analogous: the *physical concepts* used are analogous.

Bond graphs are labelled and directed graphs (Paynter, 1961; Karnopp and Rosenberg, 1968; Breedveld, 2009), in which the vertices represent submodels and the edges represent an ideal energy connection between the power ports of the submodels. The *vertices* are idealised descriptions of physical phenomena: they are *concepts*, denoting the relevant (i.e. dominant and interesting) aspects of the dynamic behaviour of the system. A submodel can either be a bond graph, thus allowing hierarchical models, or it can be a set of equations in the variables of its *power ports*. Each power port represents two variables, *effort* and *flow*, whose product is the power exchanged through that port. Examples are voltage and current (electrical domain), force and velocity (mechanical domain). The *edges* are called *bonds*. They denote point-to-point connections between the submodel power ports, describing the energy exchange between the two submodels the bond connects.

Bond-graph submodels can be reused elegantly, because bond graph models are *non-causal*. The submodels can be seen as *objects*, albeit in a *part-of* hierarchy, thus bond graph modelling is a form of *object-oriented* physical systems modelling. Besides power ports, bond graph submodels can have signal inputs and signal outputs as their interfacing elements. Bond-graph models can also be rendered using domain-specific symbols, allowing them to stay close to drawing standards in those domains.

These kind of models are called *Ideal Physical Models* (IPM), see Chapter 4 for an example.

2.4.2 VDM

VDM (Bjoerner and Jones, 1978; Dürr, 1992) is a model-oriented formal method that permits description of functionality at a high level of abstraction. VDM-SL, the base modelling language, has been standardised (Plat and Larsen, 1992). Extensions have been defined for *object-orientation* (e.g.: VDM++) and asynchronous real-time embedded and distributed systems (e.g.: VDM-RT) (Fitzgerald et al., 2008).

In VDM, data are defined in such abstractions as *records*, *sets*, *sequences* and *mappings* (Fitzgerald et al., 2013). *Types* may be restricted by invariant predicates. Primitive types are booleans, natural numbers and real numbers. Constants are defined in a section of text that begins with the keyword *values*. A state is modelled by means of typed variables.

Functions and *operations* can be specified explicitly, or implicitly by means of logical postconditions over inputs, outputs and state variables. The assumptions on which functions and operations rely are recorded as logical preconditions. Functions work on input values and calculate a result based only on those input values. Operations, read from and write to instance variables in the model.

In VDM++ and VDM-RT, a model is organized into class definitions, with each class optionally containing state in the form of instance variables. A class may also be defined as either an *interface* or an *abstract class*. An interface defines only the signatures of the operations provided by a class; it is not possible to instantiate an interface, it must be *implemented* by another class.

However, unlike other languages, such as Java, it is possible in VDM to instantiate and execute an object of both an abstract class and an interface class, but attempting to invoke an abstract operation will result in an error. Objects may *aggregate* other objects, bringing them together to perform an interleaving function.

The concurrency is modelled by interleaved *threads* synchronised using predefined predicates. The *sync* keyword contains mutual exclusion constraints that prevent multiple threads accessing the interface simultaneously. The *periodic* keyword give the period, jitter, delay and offset (in nanoseconds).

2.4.3 Tools Used

2.4.3.1 20-sim

20-sim is a multi-domain modelling and simulation tool for complex physical systems. All model libraries of 20-sim are open source (Broenink, 1999). It supports mixed-mode integration techniques to allow the modelling and simulation of computer-controlled physical systems that contain continuous as well as discrete time parts. 20-sim supports bond-graph modelling (Paynter, 1961; Breedveld, 2009).

To prepare for simulation, *computational causality* of the bonds is determined by the compiler, dictating which of the two variables of each power port will be computed as

a result (output) of that submodel's equations and, consequently, the other variable of that port will be the cause (input). This also implies that the submodel's equations might need to be rewritten. This action is also performed by the compiler.

2.4.3.2 Overture

The Overture Tool¹ is an open-source integrated development environment (IDE) for developing and analysing VDM models. The tool suite is written entirely in Java and built on top of the Eclipse platform.

The formal language VDM-RT, is used here to represent models. The tool has also been extended with the capability to output the simulation results derived from execution of VDM-RT models for further analysis.

Association and inheritance relationships can be defined between classes, and the structure of such VDM models can be expressed in UML diagram. This feature is supported by the tool. The Overture tool also supports asynchronous communication and primitives for modelling deployment to a distributed hardware architecture.

2.5 Advanced Topics

Terms that are related with advanced topics in addition to co-modelling and co-simulation, such as design space exploration and fault modelling are introduced in this section.

2.5.1 Design Space Exploration

Once co-models have been constructed, they can be evaluated through co-simulations. When two or more co-models represent different possible solutions to the same problem, these co-models are called *design alternatives*. *Design Space Exploration* (DSE) is an activity in which designers build and evaluate co-models in order to reach a design from a set of requirements. It involves making a selection from alternatives on the basis of criteria that are important to the designers (e.g. cost, performance).

These co-models evolve through a series of design steps in the whole system design process. The simulation results are evaluated comparatively against these criteria that are of interest to the designers. After this evaluation, the designers can select the best candidate model for subsequent development.

The designers can then evaluate the suitability of co-models based on these test results and feed these back into the modelling process. This feedback could include the selection and rejection of design alternatives, or the selection of a design for implementation.

The key challenges in DSE support are: helping the designers to navigate the large volume of data that results from co-simulation; the production of scenarios and alternative co-models; and in providing support for evaluation, ranking and modification of models. Currently, no software tool exists that can support co-simulation based

¹www.overturetool.org

DSE, yet such a tool is essential to bridge the gap between engineers from different disciplines. The tool support for DSE is described in this thesis.

Within the DESTECs project, tool support for the selection of a single design from a set of design alternatives and performing so-called *Automated Co-model Execution* (ACE) has been developed. This supports specification of ranges of values for co-model, and running simulations settings for each combination of these settings. The output of ACE is the set of test results from each simulation.

Automated Co-model Analysis (ACA) supports the definition and execution of a set of co-simulation runs. ACA also allows the definition of a *ranking function* which assigns a value to each design based upon its ability to meet the requirements defined by an engineer. After the co-simulation runs are complete, the ranking function is applied to all test results and outputs the *analysis results* which contain the rank(s) obtained by each design simulated.

Design space exploration and ACA are discussed in detail in Chapter 5, a case study associate with these topics is presented in Chapter 6.

2.5.2 Fault Modelling

In order to study how the cyber part can be designed to be robust against faults or disturbances in the physical part, we study how to model non-normative behaviour. In this way, along the co-design flow, the whole system under study can be made *fault tolerant*.

We regard a *fault* as the cause of an *error*, which is part of the system state that may lead to a *failure* in which a system's delivered service deviates from its specification (Avizienis et al., 2004). *Persistent errors* are those errors whose activation is reproducible. *Intermittent errors* are those elusive errors whose activation conditions depend on complex combinations of internal state and external requests. *Transient errors* that appear at a particular time, remain in the system for some period and then disappear. These classes of errors are also of interest in practical usage.

Prevention of development faults is an improvement of development processes in order to reduce the number of faults introduced in the produced systems. It is based on the recording of faults in the products, and the elimination of the causes of the faults via process modifications. *Fault prevention* means to prevent the occurrence or introduction of faults. *Fault tolerance* encompass techniques that "deal with faults" such that no failures occur. *Fault removal* means to reduce the number and severity of faults. *Fault forecasting* means to estimate the present number, the future incidence, and the likely consequences of faults. This thesis only deals with fault prevention and fault tolerance.

Certainly we should distinguish the parts of the model that represent the system to be built from parts of the model that are not part of the system but are added in order to facilitate the modelling and analysis of those parts describing *non-normative behaviour* of components.

This leads to the definition of two sets of behaviour that we may model. *Normal be-*

haviour describes a model as that it is an abstraction of the system. This kind of behaviour is considered as an simplification of the reality, it is just for the engineers to understand the working principle other than the exact behaviour of the system. It certainly is a competent model as well. In the terminology of Avižienis et al. this behaviour is within specification. Another behaviour set is *fault behaviour*. Fault behaviour describes how the object might behave when a fault has been activated and emerges from the object as a failure to adhere to its specification.

With this in mind, the term *fault modelling* is the act of extending a model to encompass fault behaviours. Parts that represent faults are distinct from the normal behaviour. *Fault injection* is the act of triggering fault behaviour (during simulation).

Fault modelling and fault injection are discussed in detail in Chapter 5, a case study associate with these topics is presented in Chapter 6.

3

Co-Simulation Framework

3.1 Introduction

In order to design CPS in shorter design cycles, lower cost and better quality, engineers often use simulations, as it has the advantage of avoiding expensive physical prototypes in early stages of the design process. However, when it comes to the question of how to establish a simulation environment that can combine different domain specific tools working together, the answer is not that trivial, which is due to the hybrid nature of CPS. In this thesis, the feasibility of such a simulation environment is first discussed in this chapter, followed by details about how to use this facility to aid the design process of CPS in Chapter 4 and 5.

From a simulation perspective (Gheorghe, 2009), several approaches are currently used for CPS simulation:

1. Formulate the system model homogeneously, i.e. using a single simulation language to express the hybrid system, and thus use a single simulator.
2. Formulate the system model heterogeneously, i.e. using different domain-specific modelling languages to model components from different domains, and thus need a means to couple the involved simulators.

In the first approach, a model transformation from one domain to another is needed in order to model the CPS in a single language. This regularly involves more abstractions and simplifications than originally planned, compromising model fidelity. Further, engineers from different specific domains often do not fully understand the other domains in general, causing misunderstandings and abstracting away relevant aspects, resulting in inappropriate model parts. However, the single modelling formalism approach does work in the case that *one* of the domains is most relevant for the design: When the discrete-event (DE) behaviour of the system is dominant, a system model can be made in which the continuous-time (CT) part is simplified and formulated in the DE formalism (an extensive model is too cumbersome to make). Comparably, when the CT part of the system is dominant, a purely CT representation can be made, in which the DE part is modelled very concise.

The second approach preserves the hybrid properties of the systems by modelling the heterogeneous components in their own most suitable formalism. In this way, the CT components of the system are modelled in one language that is best suitable for physics dynamic modelling, while the DE components are modelled in another most appropriate modelling language. In this case, no sacrifice in the CT or DE domain modelling needs to be made.

This approach, however, has the risk that since each of the simulators (that are used to execute the models, see Figure 2.2) have their own notion of time, they simply do not work together naturally. A proper synchronization scheme that couples these different simulators is therefore needed.

In this thesis, the second approach, the heterogeneous system modelling approach, is adopted. The two different domain-specific modelling formalism for the CT and DE domain are 20-sim and VDM (due to the choice of the DESTecs project). The focus of the chapter is to introduce a proper co-simulation synchronization scheme to couple 20-sim and VDM simulators in order to set up a foundation for co-modelling methods introduced later in Chapter 4 and 5.

In this chapter, first we discuss related work in Section 3.2. In Section 3.3, we discuss the lock-step synchronization scheme that used to couple these two domain specific tools (20-sim and VDM). To support the development of simulation scenarios, in addition, we introduce a script language that allows to control the co-simulation execution.

3.2 Related Work

The *Ptolemy II* (Davis et al., 1999) project, developed at the University of California at Berkeley, supports heterogeneous simulation, where per diagram a Model of Computation must be specified. Wolff et al. (2012) did a tool comparison between Ptolemy II and DESTecs tool. It is indicated that in Ptolemy II, heterogeneous composition of different MoCs is enabled via hierarchies where every hierarchy level represents exactly one MoC. A special actor, the director, enforces the MoC on each hierarchy level. Transparent composite actors do not contain a director and are mere logical groupings of actors. The DESTecs tool has the nature of object-oriented both from VDM and 20-sim. This is more intuitive than Ptolemy II from a modelling point of view (Verhaar, 2008).

Modelica (Fritzson and Engelson, 1998) is an object-oriented, equation based multi-domain language for simulating controlled physical systems, and provides a number of open and closed source libraries of physical components. The DE modelling primitives of Modelica are closer to the abstraction level found in programming languages than the DE formalism of VDM. However, in general, Modelica simulators cannot perform co-simulations (defined in Chapter 2) that combine DE and CT computation domains together.

The *Functional Mockup Interface* (FMI) (Modelisar, 2010) is a tool-independent standard for exchanging data between dynamic models, which is executed by implementing Functional Mock-up Units (FMU) that contain concrete mathematical models de-

scribing possible events in the related models. However, as indicated in Chen et al. (2011), due to the fact that FMU is at a lower abstraction level comparing to Modelica and more target-oriented, it is less flexible.

In the Design Tools and ViewCorrect projects, the graphical tool gCSP (Jovanović, 2006) was developed. This tool is capable of graphical modelling of concurrent process-oriented software based on the CSP formalism (Hoare, 1978). Its follow-up tool is TERRA (Bezemer et al., 2012), based on meta modelling of the domain. Co-simulation of networked control systems has been tried out (ten Berge et al., 2006), but the tool never reached maturity.

*Cosimate*¹ is a backplane co-simulation tool offering interfaces to tools like Simulink, Modelsim, Modelica. Only time synchronization is supported, exchanging data between simulators every time step. Cosimate has been tried out on the control of a mechatronic test set up (Groothuis et al., 2008). The two models involved have to be connected in a rather cumbersome way.

AToM3 (De Lara and Vangheluwe, 2002) is a tool for multi-paradigm modelling developed at McGill University. The two main tasks of AToM3 are meta-modelling and model-transforming. This tool is used to perform modelling in different formalisms. It can translate models between formalisms for the purpose of simulation and analysis. However this tool is at different abstraction level (symbolic level) compared to the co-simulation approach used in this thesis.

3.3 Co-Simulation Synchronization Scheme

We use the concept of co-model and co-simulation to express and execute CPS models as shown in Section 2.2 and 2.3. In a co-simulation, the DE and CT models execute in their own simulators with the steering of a co-simulation engine.

As each simulator has its own notion of time, the co-simulation engine needs to incorporate different notions. We define a system model representation as follows (Sontag, 1990; Schutter et al., 2009)

$$x(\sigma) = \phi(\tau, \sigma, x(\tau), u) \quad (3.1)$$

where

- τ : initial time $\tau \in$ time set T ;
- σ : final time $\sigma \in T$ with $\sigma \geq \tau$;
- x : state variable array X in time set $[\tau, \sigma]$;
- u : a function that maps $[\tau, \sigma]$ to control inputs U ;
- ϕ : a mapping from the initial state $x(\tau)$, the initial time τ , the final time σ and the function u to the value of the state at time σ .

This formulation includes both time-driven and event-driven cases. When x and u belong to infinite sets, this representation is a time-driven system. When x and u belong

¹www.chiastek.com/products/cosimate.html

to finite/countable sets, this representation is an event-driven system, also known as automaton. In the case that $\tau, \sigma \in \mathbb{R}$, the system is continuous time. When $\tau, \sigma \in \mathbb{Z}$, the system is discrete time. We can also have combinations of continuous and discrete time, or of time-driven and event-driven system. This can result a CPS representation. Here we adopt this notation in the co-simulation scheme with respect to time.

For a proper co-simulation framework, state events, being detected and precisely localized in the CT simulator, need to be communicated to the DE simulator. This functionality must be supported by the co-simulation engine as shown in Figure 2.4.

There are two options to achieve a synchronization for this purpose (Ni and Broenink, 2012):

- In an optimistic (Carothers et al., 1999) co-simulation, each simulator proceeds at its own pace. If a signal is received from the other simulator, the time at which the event occurred is determined. A problem occurs if the simulator's internal clock has passed the time at which the event occurred. If this happens, a roll back of the simulator has to be performed to put it in the state it was in at the time of the received event, see Figure 3.1. This roll-back mechanism is not available in all simulators.
- In a lock-step co-simulation, all simulators calculate alternatively and at equal time-steps. There is no need for a roll-back mechanism.

In this thesis, Overture does not support the roll-back mechanism, as the simulator cannot store the symbolic operation results of a previous step. Hence, the lock-step synchronization scheme, shown in Figure 3.2, is chosen.

In Figure 3.1 and 3.2, the synchronization schemes with and without roll-back mechanism underlying the co-simulation between a DE simulator (top) and a CT simulator (bottom) are illustrated. The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronizes the shared variables, events and the simulation time.

Figure 3.1: Co-simulation synchronization scheme with the roll-back scheme

In Figure 3.1, at the initial time τ , the DE simulator has processed all internal zero-time transitions and it wants to move time forward by $t_i - \tau$ shown as a co-simulation step in the figure. State events occur when the solution of a differential equation crosses some boundary value, e.g. a zero-crossing event. In the co-simulation step from t_i to t_{i+1} (attempted step 5), due to the event at t'_i earlier than the original co-simulation step $X_{t_{i+1}}$, the DE and CT simulators need to exchange states at t'_i , so a roll-back mechanism is needed. Instead of moving to t_{i+1} , the CT simulator moves to t'_i (step 5a), the DE simulator also needs to t'_i (step 5b) to synchronize with CT simulator (step 5c).

Figure 3.2: Co-simulation synchronization scheme

In Figure 3.2, the same as Figure 3.1, at the initial time τ , the DE simulator has processed all internal zero-time transitions and it wants to move time forward by $t_i - \tau$. Instead of actually performing this time step, transfer is given to the CT simulator through the co-simulation engine as shown in step 1 in the figure. The CT simulator tries to move forward to X_{t_i} (step 2). There is no event happening in between τ and t_i . The state of the shared variables in the CT model is updated at t_i (step 3). Then the DE simulator moves to t_i (step 4).

Next, when the DE simulator tries to move to t_{i+1} , again the transfer is given to the CT simulator through the co-simulation engine. Hence, the CT simulator tries to move forward to $X_{t_{i+1}}$ (step 5). As the CT simulator advances, it encounters a state event at t'_i (step 5a). The actual time that it reached state event $X_{t'_i}$ is communicated back to the DE side (step 5b). While the CT simulator has been progressing, the DE simulator remains unchanged, so its local simulation time remains at t_i and state U_{t_i} . The DE simulator then advances by $t'_i - t_i$ (step 5c), so that both DE and CT are again synchronized at the same simulation time, and the controlled variables are updated and the next time step is proposed to CT.

Figure 3.1 and Figure 3.2 both illustrate an iterative synchronization scheme for solving equation 3.1. The synchronization time step is always determined by the DE simulator.

Due to the fact that this synchronization scheme does not store the previous state,

there is quite some overhead between the CT and DE simulators, which can slow the performance of the co-simulation.

Figure 3.3: CT domain state event detection

If one or more state events happen at the CT side in between τ and t_i as seen in Figure 3.3, the CT simulator detects and localizes the first event, and hands over control to the DE simulator, which handles the event and start a new simulation step. In the situation that the event is a special case, such as so-called even-root problem (Zhang et al., 2008), if the integration step (for the variable integration steps case) is too large, there is a danger of missing two events, see Figure 3.3. The CT simulator has the possibility to specify a maximum integration time step in order to avoid this problem.

So far, a proper co-simulation synchronization scheme that couples two domain specific simulator has been introduced. With this scheme, different types of events are ensured to be detected.

3.4 Scenario Support: Script Language

3.4.1 Design Rationale

The designer creates models, reusing existing components and submodels where possible. However, the designer would like to influence the co-models behaviour: such as injecting faults during a co-simulation. Information about inputs and faults are needed to add to the co-model interface, making this behaviour configurable through scenarios. Scenarios are realised during co-simulation as shown in Figure 2.3 and 2.4.

For the purpose of injecting faults during a co-simulation, a script language called DESTECs Command Language (DCL) has been designed to be expressive enough to allow engineers to influence a co-model during co-simulation, without being overly complex.

In order to keep script language simple, some design decisions have been made. For example, it does not allow local variables to be defined. Since the local variables are not allowed, it is necessary to have a command that can restore the previous value of the variable before it has been used. The type of triggering faults can be categorized as: state-event triggered or time triggered. Hence the *condition-action* pairs is sufficient enough to address these two categories of faults triggering.

3.4.2 Main Features

As opposed to a typical procedural language, DCL allows engineers to define *condition-action* pairs (using the statement **when**), which perform an *action* when the *condition* becomes true during a co-simulation. DCL allows these conditions to reference the current co-simulation time and the state of the co-model, and to combine them with logical operators. Actions can assign values to selected parts of the co-model and also provide information back to the engineer, as well as terminating the simulation. The *condition* represents the cases of time event and state event to fulfil the responding requirements for time-triggered and state-event triggered faults.

```
when condition do
    statement1
[after
    statement2]
```

A simple when-statement that assigns a value to a variable in the DE model at a given point of time (time event) in the co-simulation can be written as follows:

```
when time = 5 do
    de x := 10
```

or an example of event-trigger (state event) case

```
when ct slider.x > X_max do
    quit
```

The **time** keyword yields the current co-simulation time. The **de** keyword indicates that x resides in the DE model. The **ct** keyword is used to access a global variable in the CT model.

Statements can also be grouped in blocks. Expressions of time can optionally include a unit given in curly braces. Time units are assumed to be in seconds if no unit is given.

Logical operators can be used in expressions. When the condition becomes true, the statement(s) in the **do** clause will execute once. When the condition becomes false again, the optional **after** clause will be executed once. The **do** clause may then be executed again if the condition becomes true.

Since DCL does not allow local variables to be defined, a special statement, **revert**, may be used in an **after** clause to change a value back to what it was when the **do** clause executed.

```

when time >= 10 and time < 15 do
  // suppose x = 5...
  de x := 10
  // now x = 10
after
  revert x
  // x = 5

```

The script language described informally with the main features here is explained in detail in Appendix A.1, which includes a VDM-SL specification and concrete syntax given in EBNF. This script language is implemented using ANTLR (Parr, 2007) based on the EBNF listed in the appendix.

3.4.3 A Small Example

In this section, an example is given as an illustration of benefits and flexibilities using a script. The example concerns a water tank and a controller (Verhoef, 2009) that should maintain the level of water in the tank between lower and upper bounds (Figure 3.4). The tank is continuously filled by the input flow φ_{in} (m^3/s) and can be drained by opening a valve in the base of the tank, resulting in the output flow φ_{out} . The flow of water out of the tank is described by Equation 3.5, where ρ is the water density, g is gravity constant, A is the surface area of the water tank, R is the resistance in the valve and V is the volume, C is the capacity of the tank. No water flows if the valve is closed. When the water reaches the “high” level mark, the valve must be opened; when the water reaches the “low” level mark, the valve must be closed.

$$\frac{dV}{dt} = \varphi_{in} - \varphi_{out} \quad (3.2)$$

$$\varphi = \frac{1}{R \cdot C} \cdot V \quad (3.3)$$

$$C = \frac{A}{\rho \cdot g} \quad (3.4)$$

$$\varphi_{out} = \begin{cases} \frac{1}{C \cdot R} \cdot V & \text{if valve open} \\ 0 & \text{if valve closed} \end{cases} \quad (3.5)$$

Figure 3.4: Overview of the water tank controller case study

We present a co-model describing the water tank and a possible controller. We begin with the contract, and then examine possible co-models consistent with the contract.

Co-simulation Contract

Contracts specify information that is to be shared between co-models during a co-simulation, specifically, shared design parameters, shared variables, and events. A contract for this water tank co-model is shown in Table 3.1.

Table 3.1: Co-simulation contract for the water tank example using one level sensor

	Name	Type	Default	Notes
design parameters	<i>maxLevel</i>	real	3.0 (m)	
	<i>minLevel</i>	real	2.0 (m)	
monitored	<i>level</i>	real	1.0 (m)	
controlled	<i>valveControl</i>	bool	false	defaults to closed

We assume that the tank has one level sensor that reports the actual height of the water. We also assume that there is a single actuator, which allows the controller to open and close the valve. The values of the high and low water marks are included.

Model

The model for the water tank is given in Figure 3.5. The flow source, tank and valve map directly to the elements in Figure 3.4. The tank provides a direct reading of the current water *level*. The valve can be opened and closed using the *valveControl* signal. The contract in Table 3.1 maps naturally to the variables in this model.

Figure 3.5: Model of the water tank

The model also includes one input and one fault. The input allows the flow of water into the tank to be controlled, the intention being that a script can control the flow

where a scenario demands it, in this case by setting the value of the *waterFlow* variable within *FlowSource*.

```
when ct level < maxLevel do
  ct FlowSource\waterFlow := 1.0
```

The faulty behaviour described is a stuck valve, where the valve could not open even when the maximum level of the tank is reached. This fault can be injected during a co-simulation by setting the *valveStuck* variable of *Valve* to true.

```
when time >= 6 and time < 7
  ct Valve\valveStuck := true
after
  revert ct Valve\valveStuck
```

The simulation results of this small example can be visualized in Figure 3.6 and 3.7.

Figure 3.6: Result of the simulation without any script

Figure 3.7: An example of the result of the simulation with the scripting of activating the stuck valve at time= 6s for 1s

3.5 Conclusion

A co-simulation framework that is able to combine two domain-specific tools (20-sim and Overture in this thesis) properly and key features of the script language (DCL) have been proposed and discussed in this chapter.

The co-simulation framework can properly synchronize the two different tools by the lock-step synchronization scheme in the co-simulation engine. With the aid of DCL, a co-model can be modified even during the process of a co-simulation. This enables the possibility of performing fault injections etc, which can help with the testing fault-tolerant mechanism.

This co-simulation framework has enabled an unique domain-specific tool: the DESTECs tool, which deploys the best from its two constituent tools: 20-sim and Overture. Comparing to the related work, this co-simulation framework can explore more in dynamic system modelling (comparing to *Ptolemy II*) with the functionalities of 20-sim. Its abstraction level of discrete-event modelling primitives are higher than programming languages (comparing to Modelica).

With the facility of this feasible co-simulation framework, the next chapters provide the methods of how to perform proper co-modelling by adopting this facility. First, in Chapter 4 the details of co-design flow and associated co-modelling methods are provided. Advanced topics of co-modelling, such as: design space exploration, fault modelling and fault-tolerant mechanism details are described in Chapter 5. Together, these two chapters provide the essentials of co-design flow and co-modelling methods that can help the designers to design CPS from a true system-level perspective.

4

Co-Design and Co-Modelling

Similar to the current status of simulation support, most of the existing modelling approaches are inclined to one discipline, either from a cyber point of view, or from a physical point of view. In other words, these approaches either abstract from the continuous properties of the system and conclude the system with a purely DE model, or abstract from the discrete state properties of the system and end up with a purely CT representation. As a result, in both cases, either the methods elaborated with CT domain or DE domain only are sufficient for design and analysis. After establishing the co-simulation environment as indicated in Chapter 3, performing system-level CPS design support is feasible in practise, but naturally we also need a method considering both CT and DE domains.

In Chapter 4 and 5, we discuss how to perform proper co-design and how the co-modelling method can be applied together with a co-simulation tool. In this chapter, first we start with the design flow of a co-design process followed by details of co-modelling method. One approach focuses on the dynamic behaviour of the system (Section 4.2), while the other approach focuses on the controller logic of the system (Section 4.3). Chapter 5 is focused on more advanced topics.

4.1 Co-Design Workflow

Considering the existing standards such as "Systems Engineering - system life cycle process" (IEEE 15288 standard) (IEEE, 2008b) and "Systems and software engineering - software life cycle processes" (IEEE 12207 standard) (IEEE, 2008a), we can identify a common design workflow (see Figure 4.1) of a CPS design as follows:

- Level 1: *Requirements and Specification*, this is a development stage where the requirements are captured and analysed into development specifications. However, often at this stage of the whole design process, the requirements for these products are not completely clear. In this stage, CPS design methods should be valuable for designers to consider whether the system that a co-model represents can meet the desired requirements. It can help identify areas of ambiguity or incompleteness and the communication about these flaws.

- Level 2: *Architectural Design*, this stage bridges the gap between requirements/specifications and architecture modelling, see Figure 4.1. In this stage, CPS design methods should be able to partition the CPS of interest into different elements with defined interfaces for further development. This stage also generates multiple design alternatives.
- Level 3: *Detailed Design*, in this stage, the CPS design methods should be able to use the system model to explore the detailed design of the chosen solution and uses simulation to test the evolving design.
- Level 4: *Implementation and Realization*, the purpose of this stage is to realize the elements (including cyber items and physical items as necessary) to a CPS in reality that will satisfy the design requirements the designers had at the very beginning.

Figure 4.1: A cone representing design workflow along with design space exploration, adapted from (Corporaal, 2006)

Abstraction levels along the overall design workflow are shown in Figure 4.1. This cone indicates that an idea can be realized in an almost infinite number of ways (the solution space is extremely large). There are many design decisions. Every decision adds detail to the design, restricts the design space, and lowers the abstraction level. Starting from the top of the cone, the whole solution space is still reachable. At all levels, the designers can make models that estimate the consequences of successive design decisions for certain aspects of the product, without really working out the design completely. This helps the designers to evaluate a large design space without creating all the possible implementations in that space. As Figure 4.1 indicates, the CPS workflow is viewed as a development process that starts with an initial concept and finishes with

a realization in hardware and software via a series of design choices. Each choice involves making a selection from alternatives on the basis of criteria that are important to the designers, such as cost or performance. The alternative selected at each point constrains the range of design alternatives that may be viable in the next steps forward from the current position.

Based on the existing standards (see above) and considering the multi-disciplinary nature of CPS, the co-design flow that combines different engineering disciplines can be visualized as Figure 4.2 (earlier versions are presented in (Broenink et al., 2010b), (Broenink et al., 2007) and (Broenink and Hilderink, 2001)).

Figure 4.2: Overview of co-design workflow

This figure shows three steps (numbered 1 to 3):

Step 1: Top-level Architecture, this is the step based on the requirements to structure the overall system structure, which is level 2 in Figure 4.1. Level 1 in Figure 4.1, is not discussed in a separate step. The author considers that the requirements are represented in the resulting top-level architecture here as the start of step 1.

Step 2: Detail Design, adding implementation details in a stepwise refinement manner, the resulting co-model is co-simulated to verify the results; which is level 3 in Figure 4.1.

Step 2a): Physical Plant Modelling, is performed to mimic the corresponding physical phenomenon that are of interest.

Step 2b): Control Law Design, using the obtained physical plant models to design and test control laws.

Step 3 and Step 4: Implementation and Realization, this is level 4 in Figure 4.1. Once the cyber part is verified, the cyber part will be tested under real-time constraints and the plant model will be prototyped in reality.

Step 1 and 2 comprise the functional design, testing the cyber-part design against a dynamic model of the plant (called a *virtual prototype*). Step 3 and Step 4 deal with adding further implementation details, which are required for code generation, like timing information, converting to and from real signals, and running the code on the final target.

The electronic design column a) and the mechanic design column e) are not in the focus of this thesis, hence they are not discussed further. Depending on the focus and interest of the CPS designers, possible starting points are software, controller design column b), c) or plant dynamics design column d). Details about starting from d) and c) are explain in Section 4.2, whilst starting from b) is described in Section 4.3.

The co-design cone shows that it is important to identify the possible components of the CPS under design according to requirements and in what formalisms they could be modelled during the step of architectural design. This can affect the final implementation of the co-design process. The (a) and (b) points, as shown in Figure 4.1, have different architecture designs, hence in the end, they have different realizations.

The top-level structure of CPS that we use in this thesis is shown in Figure 4.3. The I/O interface block is partly modelled in the continuous-time (CT) domain and partly in the discrete-event (DE) domain. Depending on the specific applications, the number of parts modelled in CT and DE can vary. In other words, the area of the Controller and Plant in Figure 4.3 can vary in different approaches. This will be elaborated in Section 4.2 and 4.3.

The I/O interface is abstractly modelled as A/D (analog to digital) and D/A (digital to analog) conversions, denoting the conversion between the physical and the computer (cyber) part of the system. The principal conversions are time discretization (i.e. sampling) and value discretization. The colour representation for different domains shown in Figure 4.3 applies to the entire thesis when applicable.

Figure 4.3: Top-level model for co-modelling

There are two more factors that should be considered when determining the co-modelling structure and in which domain each component should be modelled:

- The skill set of the co-model development team is one of the factors that can influence the path to co-modelling. Their experience in their own specific domain can certainly help the whole co-design process, however, there is also danger that the experts can be overly-biased by their background. This can be a danger to the collaborative modelling process.

- Legacy models can be existing models of CPS in a single formalism; partial models of a CPS, such as a CT plant model; models of potential components of the CPS; models of other CPS or components that relate to the CPS under design. They can be used directly or can be used as a reference. It is useful to identify these models and sources at this stage to avoid the effort of reinventing the wheel.

In the early stage of the product design, the system designer can decide which of the following co-modelling methods fits the best with the specific CPS. Depending on different starting points and different applications orientation, there are three different focused co-modelling approaches. In the sections below, these three different co-modelling approaches are discussed:

- *Dynamic-behaviour oriented*: in this approach, the dynamic behaviour of CPS is more of interest than the controller-logic behaviour. The initial models are produced in the CT formalism, with a DE model being introduced later to form a co-model. This approach is also called CT-first approach (Broenink et al., 2012) in DESTecs project terminology, details see Section 4.2.
- *Controller-logic oriented*: in this approach, the controller of the physical part behaviour is more of interest than the dynamic behaviour. The initial models are produced in the DE formalism before introducing a CT model to form the initial co-model. This approach is also called DE-first approach (Broenink et al., 2012) in DESTecs project terminology, details see Section 4.3.
- *Contract oriented*: in this approach, a contract has to be defined first as the start of the co-model development. The initial models are considered to be produced both in CT and DE sides in parallel. This approach is called contract-first approach (Broenink et al., 2012) in DESTecs project terminology, details see Section 4.5.

4.2 Dynamic-Behaviour Oriented Co-Modelling Approach

Models are first built in the CT formalism. This comprises a CT model of the plant dynamics and a discrete-time (DT) loop controller. After the loop controllers have been designed, transition to a co-model and co-simulation can take place. The loop controllers are moved from the CT model to an initial DE model¹. As it is shown in Figure 4.4, the Control Laws block (blue colour) in Step 2 merged into the pink block in Step 3. A contract has to be defined describing the variables and parameters shared between the two models. This will initially contain the variables involved in the control loop (i.e. the measured variables from the CT model and the steering variables from the control loop). This co-model can then be extended by supervisory control behaviour around the core loop controller.

An overview of the dynamic-behaviour oriented design flow is given in Figure 4.4. The figure shows three steps (numbered 1 to 3):

¹Most often-used classes are available in the co-simulation tool developed within DESTecs.

Figure 4.4: The overview of the CT-first design flow

Step 1: *Top-level Structure*: based on the different study interests of the CPS under study, a proper initial top-level structure is chosen. As the dynamic behaviour of CPS is more of interest than the controller logic behaviour, then the top-level structure should be structured as shown in Figure 4.5.

Step 2a: *Physical plant modelling*: modelling of relevant dynamic behaviour of the plant.

Step 2b: *Controller design*: using the plant models obtained to design proper control laws to perform required tasks.

Step 3: *Co-simulation*: combining the loop controller and the software structure in a co-model, to describe the control software.

The result can be used to generate controller software that, running on an appropriate computer hardware, controls the physical plant in reality. This "real" physical plant is modelled in the part named "Plant" in Figure 4.5. As one can see that the area of the plant block is way larger than the one in Figure 4.3, which means that this approach puts more effort in the CT domain modelling. Based on the top-level model in Figure 4.5, in the following sections, Step 2a (plant modelling), Step 2b (loop controller design) and Step 3 (co-simulation) will be discussed.

Figure 4.5: Top-level Structure of Dynamic Behaviour Oriented Approach

4.2.1 Plant Modelling

There are two situations of plant modelling, namely the plant exists already (analysis situation) or does not exist (design situation) yet.

Modelling a non-existing plant means that specifications for it need to be described as physical-system models. Models of design alternatives can be made, and compared to each other via simulations. Doing so, the alternative models cover (a part of) the design space. As modelling and simulation in general is faster than making prototypes, Design Space Exploration (DSE) can really be facilitated through modelling and simulation. This DSE is not only done during plant modelling, but also in later phases when controllers are designed. In that case, alternatives can span all parts of the model, such that one can trade off between controller solutions and plant solutions.

For an existing plant, we can model its structure using a phenomenological approach: look at the plant to discover what physical system elements the plant consists of. Values of model parameters can be found using identification techniques: exciting the real apparatus, and measuring responses at relevant points. This often gives only a transfer function between excitation and measurement points. For controller design, this is in most cases detailed enough. Simulations can then be compared with the same experiments applied on the real machine to validate the model.

In general, plant modelling starts with putting essential parts in the model. These initial models are verified via simulations. The model is extended until it is satisfactorily competent, as indicated by the goal of the model. Figure 4.6 gives a standard blue-print of plant modelling using bond-graph concepts (see Karnopp and Rosenberg (1968) and van Amerongen (2010) for an introductory text). The plant is modelled as a *submodel* having a steering input, the steering value from the controller to steer the actuator, and a measurement output, bringing the state of the plant model to the controller. The actuator is explicitly but abstractly modelled as a Modulated Source (MSe). Structuring the plant model like Figure 4.6, facilitates both easy connection to a controller and use within a co-model.

Figure 4.6: A blue-print of plant model

Often plant models are described in their domain-specific formalism in order to stay

closer to the specific engineering domains at hand. In general, these models are *Ideal Physical Models* (IPM), as the modelling elements describe ideal phenomena, just as with bond graphs. Translation from IPMs to bond graphs is an algorithmic process.

By using bond graphs, each submodel is characterized by constitutive relations at the one hand and its interface, the power and signal ports to and from the outside world, at the other hand. Other realizations of such a submodel may contain different or more detailed descriptions, but as long as the interface (number and type of ports) is identical, they can be exchanged in a straightforward manner.

As an example, we consider the torsion-bar setup representing a wide class of electromechanical systems where the angle or angular velocity (or the position and the velocity) of the load has to be controlled. As Figure 4.7a shows, an electric DC motor is coupled to a voltage source on the electrical side and to a mechanical load by means of a belt and a flexible rod.

(a) The top view of the torsion bar setup in reality

(b) Ideal Physical submodel of torsion bar, a typical electrical driven mechanical system

Figure 4.7: The torsion bar as an example to illustrate the methods (Kleijn, 2011)

Figure 4.7b shows the IPM of the plant part of the torsion bar. Its bond-graph submodel is shown in Figure 4.7c. It is explicitly written as a submodel, to have the same structure as the blueprint shown in Figure 4.6.

Figure 4.7: The torsion bar as an example to illustrate the methods (Kleijn, 2011)

By simulation, one can verify the model, i.e. check whether the model as written down in the tools behaves as it was designed to do. For this check, one can run *test simulations* to judge the behaviour of the model under particular circumstances, like equilibrium situation, step response, transient response (see Figure 4.8). Important aspects to look at are the *shape* of the signals and the *value range* of the variables.

Figure 4.8: Simulation result, showing a step response and equilibrium at the beginning

Note that the shape of the curves of the simulation result can be predicted using the (*system theoretic*) order of the system. This order can be determined by interpreting the *computational causality* information of the bond graph.

The plant submodels should have steering signals as their inputs; otherwise a simulation cannot run. Outputs need not to be connected for simulation to proceed, as these are only computed values. A blueprint of a test rig is shown in Figure 4.9. Only

the plant input is connected. Several experiments can be conducted by varying the signals offered to the plant input. The results of simulations can be inspected to verify the model.

Figure 4.9: Blue-print of plant model in a test rig

The torsion bar model can be simulated using a test rig, as shown in Figure 4.9. The signal source is used to generate excitation signals for the torsion bar model.

The following issues are worth considering for the designer while building a CT model:

- **Modelling Step:** Fill in quantities and units of the physicals system variables, to let 20-sim use this metadata for checking the model; use meaningful names in the models and submodels; annotate graph with coloured backgrounds as a means to bring in structure (shown in Figure 4.7c).
- **Simulation Step:** Let excitation signals start with equilibrium values (usually 0), to check that the initial state of the model is in equilibrium; test non-linear model parts separately as submodels in a test rig.
- **Analysis results Step:** Check shape of curves comply with the expected shapes. Use causality information from the bond graph model to verify the system theoretic order of the plant; check results of 'standard' experiments, like step response, transient response; check the ranges of the values of the variables.

4.2.2 Controller Design

Once the plant modelling is done, we can design a controller using that plant model, or a simplified version of it. It is also possible to derive a set of control laws, each having its own operating conditions. This can make each individual control law more simple or give a better performance. A switch between controllers (*Mode-Switch Control*) (Hilhorst, 1992) might introduce a bump in the control signal. Bumps in the control signal are unwanted and therefore should be removed. Switching from one control law to the other must be designed, and the system should behave smoothly when switching from one control law to another (so-called bumpless transfer (Astrom and Wittenmark, 1994) is necessary).

There are two options to start:

- *Continuous-time first*: abstracting away from the digital implementation of the controller. Later, the continuous-time controller is converted to discrete time. Classical control law design techniques can be used.
- *Discrete-time directly*: immediately taking the time discretization of the controller into account, using appropriate control law design techniques.

For both starting points, the procedure to design a control law is the same:

- *Derive a simplified model*: starting from the detailed model, either reduce it automatically (e.g. by linearization and/or order reduction) or simplify it by hand to obtain a model suitable for control law design. It may be necessary that more than one simplified model is needed to cover the whole workspace of the control system. If during the modelling process, enough abstraction was made, the simplification might not be necessary.
- *Verify the simplified model*: verify the simplified model by performing the same simulations as with the detailed model. Results should not differ significantly.
- *Derive the control laws*: derive control laws, using the simplified model acquired in the previous step. This is common practice in controller design, and elaborated on below.
- *Verify the control laws*: construct a test rig in which the control law is connected to the detailed model. Verify the control laws by performing simulations. Run such experiments that the demands on the controller performance can be checked.

Some standard control laws that one can select are:

- *PID control*: proportional, integral and derivative manipulation of the error. For a faster and well-damped response, a derivative of the error signal can be used, and to achieve a zero steady-state error, the integral of the error can be used. PID controllers are most widely used, as they are able to realise reasonably good control performance for a wide range of systems.
- *State feedback control*: all the continuous-time states of the plant are used in the feedback. When not all states are available via measurements, observers can be designed to reconstruct missing states.
- *Adaptive control*: where the control law parameters are updated according to the control performance. The update speed is lower than the update rate of the steering value.
- *Model-reference control*: where a (simplified) model of the plant is calculated as part of the control law. When this model influences the 'normal' control parameters, it is called *Model-Reference Adaptive Control* (MRAC).

- *Feed forward control*: where a component of the steering value is computed purely from the setpoint and not using the error.
- *Force control, impedance control, admittance control*: these are interaction control schemes that are commonly used in robotics applications (when interaction with the environment happen, such as grasping). (Chiaverini et al., 1999).

Once the appropriate control scheme has been selected, it needs to be tuned, for which several approaches exist, depending on the complexity of the control scheme. For instance, the classical Ziegler-Nichols procedure for PID controllers is one of the ways to tune the controller.

For Discrete-Time Controllers, the *sampling time* T_s must be derived. To obtain a stable system (i.e.: no instability caused by the sampling), T_s needs to be about 10 times smaller than the dominant time constant of the plant. In this case, the sampling activity has only a minor influence on the behaviour of the controlled system. Next to this, when the *Sample – Compute – Actuate* approach (see Figure 4.10 top) is implemented, the delay between sampling and actuation must be smaller than about 20% of T_s (van Amerongen, 1994), to keep the assumption that computation time can be neglected. This approach does not always guarantee that the system will work as expected. Using the *Sample – Actuate – Compute* approach instead, and thus taking into account that the computing devices are not ideal, assumes a fixed *control delay* of one sampling period. This fixed control delay can be compensated for in the control logic, see Figure 4.10 bottom.

Figure 4.10: Sampling period and control delay, 'avET' is average execution time (Orlic, 2007)

When a controller is first designed in continuous time, it can be directly translated to a discrete version for computer implementation, but it has to be retuned to obtain the best performance.

Arriving at this stage, the control laws design together with the detailed physical-model can be used in the next step of CPS implementation.

4.2.3 Co-Simulation

When the current expressiveness for the controllers is running out, in this case, when a complex controller is needed (e.g.: a complex supervisory controller is needed), the controller part should be moved to the DE domain. Once a co-model is created, a co-simulation can be run by using the co-simulation tool. Comparing to Figure 4.5, the controller part is entirely in DE domain (pink).

Figure 4.11: Towards co-simulation

In order to translate the entire controller part to VDM as shown in Figure 4.11, perform the following steps:

- *Translate CT controllers to VDM*: using the existing controller template library in the co-simulation tool. A class diagram of the controller library is shown in Figure 4.12.
- *Design and implement the supervisory controllers*: after translating the loop controllers to VDM properly, the supervisory controller should be designed and implemented.

Figure 4.12: Class diagram of the control library

An example of a supervisory controller is given in Listing 4.1.

```

class Supervisory_Controller
.....

operations

    Monitor: ()==>()
    Monitor ()==(
        -- get the sensors status
        let encl = (enc_load/ENC_RESOLUTION)*2*MATH`p,i
            speed = (encl-prev_encl) SAMPLETIME
        -- change different controllers if necessary
        in (if encl<=MIN and encl>= MAX-- out of range
            then EmergencyStop()
            elseif encl>= SLOW_MIN and encl<= SLOW_MAX
            then -- over speed limit in slow region
                if speed > SPEED_LIMIT then TunedPID();
                -- change to another loop controller

            prev_encl := encl
        )
    )

end Supervisory_Controller

```

Listing 4.1: An example of a supervisory controller represented in VDM

Here we reach Step 3 (co-simulation) of the whole design flow as shown in Figure 4.2. Details of this step are given in Section 4.4.

4.3 Controller-Logic Oriented Co-Modelling Approach

In the controller-logic oriented approach, initial models are produced in the discrete-event formalism and the focus is on developing the DE controller first. In this case, the DE formalism we chose is VDM, which has object-oriented property. Hence the mindset in this section needs to be towards object orientation.

Similar to Section 4.2, the design flow for controller-logic oriented approach is shown in Figure 4.13. The figure shows four steps (numbered 1 to 4):

Step 1: Top-level Structure: based on the different study interests of the CPS under study, a proper initial top-level structure is chosen. However as the controller of the physical part behaviour is more of interest than the dynamic behaviour, then the top-level structure should be structured as shown in Figure 4.15.

Step 2a: Physical plant modelling: modelling of relevant dynamic behaviour of the plant.

Step 2b: Controller design: using the plant models obtained to focus on designing proper control laws to perform required tasks, as the main focus of this approach.

Step 3: *Co-simulation*: combining the loop controller and the software structure in a co-model, to describe the control software.

Figure 4.13: The overview of the DE-first design flow

Figure 4.14: Top-level Structure of Controller-Logic Oriented Approach

As shown in the figure above, the development begins with a CPS model fully in the DE formalism. This model should contain objects that represent controller, plant and the I/O interface.

The DE model of a CPS contains a controller and a discrete approximation model of the plant, which are linked by the I/O interface (sensor and actuator) objects. The plant object is used to mimic the behaviour of the CT world in the DE domain. To create these objects, it is necessary to define a class for each one that describes their properties and behaviours.

The plant essentially acts as a basic simulator running in its own thread and provides stimuli to, and observes the responses of, the controller model. The controller accesses a plant model through I/O interface. This requires the I/O interface (sensor and actuator) containing classes that contain references to a plant class, which allows them to read and write the plant object, respectively, as shown in Figure 4.15.

Figure 4.15: An example of a CPS represented in DE model formalism

Based on the top-level model in Figure 4.14, in the following sections, Step 2a (plant modelling), Step 2b (controller design) and Step 3 (co-simulation) will be discussed accordingly. One can see that here the whole area is in pink, which means that it is modelled in DE.

4.3.1 Plant Modelling

Within the plant class, there are two main ways to build a plant model. The first one is to implement a basic numerical integration algorithm, such as the Euler method to act as a basic CT simulator. Another one is to use pre-calculated data or measurement data as the relation between sensor and actuator to mimic the plant behaviour.

These two ways are discussed in the following subsections.

4.3.1.1 Basic Numerical Integration

When the plant (such as Equation 4.1) is a simple dynamic model which can be expressed in Ordinary Differential Equation (ODE), the numerical approximation of the solution (Equation 4.2) is achieved by implementing a basic numerical integration method, such as Euler.

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)); \quad (4.1)$$

$$\mathbf{x} = \int \dot{\mathbf{x}} dt; \quad (4.2)$$

A basic numerical integration method, can be sufficient for a plant to test the core control logic. In case of the Euler method, the integration step $h(k)$ is a constant.

$$\begin{aligned} \mathbf{x}(k+1) &= \mathbf{x}(k) + \dot{\mathbf{x}}(k)h(k); \\ \text{while } t_{k+1} &= t_k + h(k); \end{aligned}$$

However, using this basic Euler integration method, it cannot properly simulate complex dynamic models, as the stability of the numerical method depends on the eigenvalues of the plant dynamics. It is then necessary to explore more mature CT numerical solvers when simulating complex plant dynamics, hence move the simulation to a co-simulation.

4.3.1.2 Data-Driven

When the real plant's measurement data are stored in files, it is possible to construct signals based on those data to simulate the plant reaction.

In general, the idea is to use the built-in functionality of the tools to read in the data that representing the behaviour of the physical plant. These data can be pre-calculated or stored measurement data.

Sensor and actuator objects are used to allow the controller to interact with the plant. In order to do this, classes for these objects are required. For each type of sensor and actuator in the system, a class is created. Each class should declare access operations as appropriate.

4.3.2 Controller Design

Typical controller types are loop controllers and supervisory controllers. Loop controller design is the same as indicated earlier in Section 4.2. A straightforward loop controller written in VDM is given in Listing 4.2. An example of a supervisory controller is shown as in Listing 4.1. A list of different patterns for supervisory controllers design is given at the end of this section.

It is logical to start with a minimum amount of complexity in the controller, as later due to the increasing of the complexity along the design process, debugging can be extremely difficult. A minimum VDM controller requires the following elements:

- *Controller class* that defines a periodic thread. The thread definition requires the name of an operation to call, e.g. Step. This is operation should contain the control logic.
- *Cyber class* with a constructor that creates an instance of the Controller class. It should also create a CPU object and deploy the controller object to it.
- *World class* with a Run operation that executes with the start statement.
- *An instance variable* for each shared variable. These instance variables will be kept up to date by the co-simulation engine. They should be accessible to the controller, so can be defined in the Controller class itself, or statically in the Cyber class.

```

class PID is subclass of Controller

.....

-- default constructor for PID
public PID: () ==> PID
PID() ==
    PID(DEF_K, DEF_TAU_I, DEF_TAU_D, DEF_BETA);

-- calculates output, based on the error
public Output: real ==> real
Output(err) ==
(
    decl factor: real := 1 / (sampletime + tauD * beta);
    uD := factor * (tauD * uD * beta + tauD * k * (err - prev_err)
    + sampletime * k * err);
    uI := uI + sampletime * uD / tauI;
    uI := 0.0;
    prev_err := err;
    return uI + uD
);

.....

end PID

```

Listing 4.2: A standard PID Controller represented in VDM

In the case that it is required to handle events, such as a button being pressed, then a different structure should be employed. The controller has to contain a method that can be invoked when the event occurs. This method is the event handler for the controller and it should not expect to have any parameters to be passed to it. Then the model should also contain an event generator, which reads the plant and contains a statement of the conditions under which the event should be raised. When those conditions are true, the event generator calls the event handler method of the controller.

There are more controller patterns for DE-side controller design, which have been developed in DESTECs project, such as modal controller pattern, kernel pattern and monitor pattern etc (Broenink et al., 2012). Each of this pattern applies to specific purpose.

- A modal controller pattern is to encapsulate modal behaviours using mode objects. Mode objects can be switched easily at run-time to change mode, activate fault-tolerance mechanisms, or switch to degraded behaviours. This is related to the modal switch controllers from a control theory aspect.
- A kernel pattern has a kernel, which is a small and verifiable component that guarantees some property of a system (typically security or safety) by protecting the controller from making unsafe control actions through interception.
- A monitor pattern provides monitoring actions of the controller (or other compo-

nents) and protects from unsafe situations by intervening and instructing the controller to stop the unsafe action.

4.3.3 Co-Simulation

When the plant dynamic becomes too complicated for simple numerical integration methods to approximate, it is better to move to a co-model. This requires definition of a contract and also alternative implementations for the sensor and actuator objects have to be provided. In this way, the sensor and actuator object do not interact with the plant object and act simply as locations for shared variables that are updated by the co-simulation engine.

In a co-model, the plant class is replaced by a CT plant model. Therefore, alternative sensor and actuator classes that simply act as placeholders for shared variables are required. These classes are simple, with sensors only being required to return the local variable in Read operations and actuators being required to set the local variable in Write operations. Once the sensor classes are created, they should be instantiated within the DE model as well.

The classes that make up the DE-model, minus the plant class can then be transferred to a co-model. This is actually a transformation process from Figure 4.15 to Figure 4.16. The event generator can be discarded and the event handler can be invoked by the co-simulation engine when the CT model raises the appropriate event.

Here we reach Step 3 (co-simulation) of the whole design flow as shown in Figure 4.2. Details of this step is given in the next section.

4.4 Co-simulation Step

The co-simulation step is the last step of both dynamic-behaviour oriented and controller-logic oriented approach. It is also the starting step of contract oriented approach. To illustrate the co-simulation step from an operational perspective, the co-simulation step is illustrated by using a design pattern.

Design patterns aim to provide inspiration to designers by describing solutions that have worked in the past. While the exact result of the application of a design pattern is likely to be unique in every case, the core of the solution can be broadly similar over numerous applications. We adopt the style of Gamma et al. (1995) to illustrate.

Intent

To build a well-structured co-model for co-simulation.

Motivation

Once the model is running out the current expressiveness, it is necessary to translate to a co-model, the following pattern can help structuring it. The shared variables are placed in Sensor and Actuator classes, following object-oriented practice. Objects of these types are then passed to the Controller object, which uses them to perform control.

Structure

A class and object diagram are shown in Figure 4.16. The class diagram (Figure 4.16b) shows the full VDM controller consisting of five classes:

(a) Class diagram

(b) Object diagram

Figure 4.16: Full VDM controller (class and object diagram)

- **World.** This class is simply used as a bootstrap to initiate a co-simulation with the Run operation.
- **Cyber.** This class holds references to sensor and actuator objects (which in turn hold shared variables, which will be updated by the co-simulation engine).
- **Controller.** This class is the actual controller, which defines a periodic thread and an operation that implements a control loop (reads sensor values, performs calculations, and writes actuator values).
- **Sensor.** This class is an example sensor class, which holds a shared variable as an instance variable (of type real) and provides an operation Read with which to access the variable.

- **Actuator.** This class is an example actuator class, which holds a shared variable as an instance variable (of type `real`) and provides an operation `Write` with which to modify the variable.

The `Cyber` and `World` classes are a feature of all co-models and are therefore part of the blue-print of VDM standard structure (see Figure 4.16).

Implementation in DE Domain

The following notes can help in the use of these classes and in avoiding some possible errors.

- The `Sensor` and `Actuator` classes should be replaced by classes that match the co-model contract (and with more useful names). `Cyber` and `Controller` will all need to be modified to reflect these changes.
- The `Cyber` class may only define instance variables and a constructor (an operation called `Cyber` with return type `Cyber`).
- The `DESTECS` tool requires that instance variables in the `Cyber` class are given a value, therefore the types are all given as optional (in square brackets in Figure 4.16b) and they are initialized to `nil`.
- The use of `Sensor` and `Actuator` classes, allows access to shared variables to be controlled and multiple identical sensors and actuators can be easily created. It also aids fault / fault tolerance modelling.
- `Sensor` and `actuator` objects are passed to the constructor of the `Controller` class (as opposed to being accessed statically via `Cyber`, following good object-oriented practice).
- The controller is deployed to a CPU, meaning that statements within controller will take time (advances the simulation clock). The default is two cycles per statement. The sensors and actuators will not be executed on the control computer, therefore, they are (implicitly) deployed on a virtual CPU (meaning their computations will take zero simulation time).
- The `Controller` class requires the `Step` operation to be implemented with some control logic as shown in Listing 4.3.
- The first parameter of the periodic thread definition controls the period.


```

class Controller

instance variables

private sensor: Sensor;
private actuator: Actuator

operations

public Controller: Sensor * Actuator ==> Controller
Controller(s, a) ==
(
    sensor := s;
    actuator := a
)

public Step: () ==> ()
Step() ==
(
    -- sensor.Read()
    -- perform calculation
    -- actuator.Write(...)
)

-- 1Hz
thread periodic(1E9, 0, 0, 0)(Step);

end Controller

```

Listing 4.3: An example of a controller represented in VDM

Implementation in CT Domain

In a co-model, the Controller previously implemented in the 20-sim (Figure 4.5) has now been moved into a VDM model for the dynamic-behaviour oriented case. In the case of controller-logic oriented case (Figure 4.15), the plant model has now been moved to a 20-sim model. So the IO Sub block needs now to act as an interface for exchanging data, as shown in Figure 4.17.

Figure 4.17: An illustration of the CT Domain Implementation

4.5 Contract Oriented Co-Modelling Approach

There is also a so-called "contract-first" approach that begins with the definition of a co-simulation contract, followed by concurrent development of the two constituent models as shown in Figure 4.18. In this way, the constituent models must evolve together.

A co-model can be reached early on, though following the concurrent controller-logic oriented and dynamic-behaviour oriented approaches on the constituent models adds extra effort. The detail steps of the concurrent controller-logic oriented and dynamic-behaviour oriented approach follow the steps introduced in Section 4.2 and Section 4.3. It is needed to produce simple plant model for DE and CT test controllers as shown in Figure 4.18. The green blocks, in the integration of initial co-model step, are the interface from CT implementation to the DE implementation, similar as Figure 4.17. If no legacy DE and CT models exist, and the development team has a good mix of DE and CT modellers, clearly this approach would be the best way to go.

Figure 4.18: Contract-First Approach

4.6 Conclusion

In this chapter, three different focused co-modelling approaches are introduced: dynamics-behaviour oriented approach, controller-logic oriented approach and contract oriented approach.

The dynamic-behaviour oriented approach begins with models being produced in the continuous-time formalism. The CT model has basic controllers, such as a loop controller, in order to gain confidence in the dynamics model. This approach allows plant dynamics to be studied early on in the development, and loop controllers to be tuned at an early stage. It can also perform a feasibility study to test whether the proposed design is in fact controllable.

Choose this approach when the dynamics behaviour of the CPS is of a higher priority than the controller logic behaviour. If legacy CT models exist, or when the development team has most CT modelling experience, then this approach is a good way to reach an initial co-model. As well as it applies to the case when the physical plant already exists, and a new controller is to be built.

The controller-logic oriented approach begins with initial models being produced in the discrete-event formalism. This DE model includes a simplified model of the plant that is only required to respond sufficiently to the supervisory controller to test its basic operation. Later, this plant model is replaced by a higher-fidelity CT model as part of the initial co-model. This approach allows supervisory control to be studied early on in the development.

Hence, it is a good choice when the behaviour of controller algorithm is critical, such as in systems with safety or security concerns. Or when the systems do not involve too much physical dynamics. Or when legacy DE models exist, or when the development team has most DE modelling experience, this approach also applies.

The contract-first approach applies for a new project that starts from scratch. The designers can start a project by defining a contract and then from the DE and CT domain concurrently.

In theory, the co-design flow and co-modelling methods proposed here can be applied to other similar tools, such as TERRA, as Bezemer (2013) proposed a similar way of working. However certain twists in the methods are expected, considering different tools have their own properties, which might not be skipped or ignored in this method. It can be valuable to apply these methods with other tools, which can be helpful with the generalization of the methods.

5

Co-Modelling Advanced Topics

There are quite some challenges in the early stage of a CPS design process and one of the difficulties is how to evaluate design alternatives. Making early-stage decisions needs the requirements to be available, but the requirements often only appear when the product designing is eventually finished.

Hence, rigorous design-alternatives evaluation procedures are necessary after any design change in order to assess and to reduce the risk of malfunction or unanticipated system failures. Some CPS have obvious risks associated with failures such as aircraft flight control and vehicle cruise control. Normal approaches for such systems that could cause personal or property damage are to employ redundancy in order to ensure continued operation after equipment failure. However, this approach would need extra cost of hardware redundancy. The extra redundancy can also make the overall system more complex and more error-prone.

A method (design space exploration) with a corresponding tool supporting facility that can assist the designers with a better comparison among different design alternatives in the early stage is handy for the designers. To handle possible fatal design flaws of CPS, modelling possible faulty behaviour of these devices and designing cyber part software that deals with this faulty behaviour at an early design stage is helpful to construct the actual devices "First-Time-Right".

In this chapter, design space exploration, fault modelling and fault-tolerant controller software design are discussed as advanced topics of co-modelling.

5.1 Design Space Exploration (DSE)

Design Space Exploration (DSE) has a specific meaning in Electronic Design Automation (EDA), where it refers primarily to work on architecture synthesis, for example through multi-objective optimization (e.g. Eisenring et al. (1999) and Zitzler and Thiele (1998)), evolutionary algorithms to increase the coverage of the design space exploration, (e.g. Holzer (2008) and Haubelt et al. (2006)), reduction of overall exploration time set by heuristics (e.g. Palermo et al. (2005) and Erbas (2006)) and automated synthesis using genetic programming (e.g. Hu et al. (2008)).

5.1.1 Design Space Exploration and ACA Aspects

Here, we use the term in a different context. During the procedure of design space exploration, often engineers are faced with numerous design choices. Experienced engineers can make the right choices towards an optimal solution based on the previous experience that they had in other CPS design projects. They are skilled in determining which aspects of these choices are most important to reach something close to an optimal solution. To search for an optimal solution, many possible combinations need to be examined. In order to help engineers, it is beneficial to have tool support that can sweep over different alternatives to gain data about the alternatives. This can ease the decision making as to which alternative should be chosen.

In general, we can categorize co-modelling design alternatives into three main types:

- Type 1 - *Parameter variations*: in this case, the model structure and the interface (contract) do not change. Only design parameters or scripts (i.e. the scenario) are changed. This can be done by specifying a sweep over the range of interesting values of the parameters, or more automatically, in case of parameter optimization, as is regularly used in control engineering (Bazaraa et al., 2006). In that case, a criterion function must be specified. The purpose of experimenting with different values of parameters is to find the optimised parameter values for the co-model at hand.
- Type 2 - *Changes inside co-model parts*: changes are inside either the CT model or the DE model, and do *not* affect other parts of the co-model. So, only the co-model implementation at *one* side changes. The purpose of experimenting on this level is to evaluate different model structures within one domain. For example, different kinds of motor structures or different kinds of controller structures.
- Type 3 - *Co-model change*: changes affect *both* the CT model and the DE model, and mostly also the contract. So both the co-model implementations (CT *and* DE side) change. This kind of model variations deal with cross-domain effects. Note that a change of the contract is not always the case. So it is possible that the change at hand affects both sides, but *not* the contract. For example, changing the size or mass of the robot affects the controllers, but not the interface between the CT and DE side. The purpose of experimenting at this level is to evaluate how different model structures affect the whole co-model.

The activity of DSE has been defined as *Automated Co-model Analysis (ACA)* and *visualisation and analysis* of co-simulation results (Ni et al., 2011) in the DESTECs project. ACA itself can be performed by executing settings and/or scripts, as shown in Figure. 5.1.

Figure 5.1: ACA Tool Workflow (DESTTECS Tool Facility (Ni et al., 2011))

Considering the three categories of co-modelling design alternatives, with the ACA tool facility, we categorize the following three main aspects that can be configured in an ACA experiment using the tool:

- Aspect 1 - *Shared design parameters*: this is the main area in which the design choices needs to be taken. This is typically a matter of where precisely sensors and actuators are placed or dimensions of fixed size physical objects.
- Aspect 2 - *Scripts giving inputs to co-simulations*: in order to inject different kinds of excitations signals to the co-simulation, it is possible to make use of a script. Different scripts can be explored in this aspect in order to determine whether the same design choice will be the best under all circumstances or whether there is a trade-off where an engineering decision needs to be taken.
- Aspect 3 - *Injection of specific faults*: in order to determine the fault tolerance of different design alternatives, it can be an advantage to be able to add alternative faulty submodels at specific places either in the DE or CT model and to trigger the faults when needed.

Aspect 1: Different Shared Design Parameters

In order to determine what the best design choices are, one can vary one or more of the shared design parameters in different co-simulation executions. When several parameters are chosen, ACA will ensure that all possible combinations of the variations are executed. This is also called *sweeping* over the chosen parameters. A parameter sweep is typically used to find an optimal value for a design parameter. *Optimal* means that the results of a particular simulation run are evaluated against certain pre-defined criteria to find out the best result according to the criteria. It is also possible to determine the sensitivity of the co-models results for a given change in a certain parameter. This aspect applies to the Type 1 of the three different types of design alternatives.

Aspect 2: Different Scripts with All Combinations

Scripts are used to simulate different kinds of external excitations at specific points of time during the co-simulation of a co-model. There may be many different kinds of circumstances that are worth examining using co-simulation. Different solutions to design decisions may not perform equally well under all of the possible scenarios. It is very likely that some candidate solutions are best in some cases and worse in others. Thus, sound engineering judgement is needed to make the best decisions about the

different design choices. This aspect applies to the Type 1, 2 and 3 of the three different types of design alternatives.

Aspect 3: Different Specific Faults Injection to Determine Fault Tolerance

The last aspect considered for ACA definition of an experiment is automatic replacement of components either in the DE or the CT side with a faulty version of the same component. This kind of automatic *sub co-model changes* is already enabled both at the DE and the CT side manually. In an ACA context, the idea is that the users shall have the possibility to explore the consequences of injecting different kinds of faults before starting co-simulations with the different other alternatives for the significant design choices that needs to be taken. In this way, it can be determined whether the different design alternatives are tolerant against the particular faults injected. Naturally here it boils down to economic considerations whether the components in a particular design choice are significantly more expensive than another design choice weighted against the consequences of not being tolerant against the fault. At the DE side, this is typically done using inheritance and at the CT side this is done using *alternative implementation* of a component that then can be activated. This will be elaborated in Section 5.2. This aspect applies to the Type 2 and 3 of the three different types of design alternatives.

5.1.2 Tooling Support Detail of DSE and Discussions

The ACA facility consists of selecting possible settings and scripts as different inputs to executing co-simulations. In fact, *Automated Co-Model Analysis* performs a combinatorial explosion executing a co-simulation execution for all possible combinations of such settings and scripts. Figure. 5.2 gives an example of these combinations of different settings and scripts.

Figure 5.2: An illustration of the ACA different combinations

The result of generating complete input configurations with selected settings and a script, from the partial configuration shown in Figure. 5.2 would be 4 different complete configurations: A1-B1-C1; A1-B1-C2; A1-B2-C1; and A1-B2-C2. The user can easily get many more configurations by adding more parameters or adding more val-

ues to existing parameters, for example, simply adding a A2 value would result in 4 more different configurations.

The result of the execution of each individual scenario is stored, together with the scenario so that it is possible to re-run an experiment. The results and scenario can also be used later in the post-analysis phase when necessary.

It is assumed that that sweeping over one specific parameter (for example power to an engine) will be monotonically increasing or decreasing. Thus in case such a sweep from the lowest to the highest value for such a parameter is carried out (with unchanged values for the other parameters).

The DSE tool is designed to cater for engineers from different domains during the different stages of a co-simulation based development. In early stages, systems engineers will use the tool to perform design evaluation and revisions. Later on, after essential system-level design decisions are made, engineers from different domains will need to use the DSE tool in order to assess the effects on system-level behaviour of alternatives confined to their own constituent model. It is possible to compare the results of different combinations using fixed ranking functions and this will yield different results of the analysis that can be used by an engineer to judge the best possible design candidate.

Details on how ACA can be performed can be found in the mobile robot case study in Chapter 6.

5.2 Fault Modelling and Fault-Tolerance Mechanism

Faults, and the mechanisms to address them, are a source of complexity in system modelling and design. This section discusses fault related issues: fault modelling and fault-tolerance mechanism.

As we already know that modelling is an abstraction of the reality, which means that a model is not 'exactly' the same as reality. In order to make the CPS device design more robust, it is useful to model possible fault behaviour during the early design phase.

In this section, we introduce how to perform fault modelling and how to make controller software compatible with different abstraction levels (with faults and without faults) of the same plant model.

5.2.1 Fault Selection and Modelling

We use the terms *normal behaviour* and *fault behaviour* as introduced in Chapter 2. A *fault* model includes faulty behaviour of the component being modelled. This model mimics the behaviour of the component exhibiting the envisaged fault.

When triggering the fault behaviour in the model, it is important that the introduced fault behaviour is distinct from the normal behaviour. This is to ensure that any faults modelled do not get included in the implemented system. The general approach to achieve this goal of separation depends on whether the fault is being modelled in the DE model or the CT model.

When introducing a fault into the DE model, we advocate taking the original class that exhibits the desired behaviour and creating a subclass that extends it (see Section 5.2.2). This subclass is then instantiated in the model rather than the original superclass whenever faulty behaviour is required. The subclass implementation will only contain the operations and data required to model the faulty behaviour along with at least one function from the original class. This last operation overrides its counterpart in the superclass and so will be the one used when the model is executed. The original class may also require fields to be given protected rather than private status for the subclass to function; finally, constructors for the subclass, rather than the original class, will need to be invoked. These modifications will mean that the controller model is distorted from the controller that would be implemented in the final product, but these distortions are necessary to allow testing.

For the plant models (CT models), we propose the use of a fault injector pattern where a fault block effectively wraps the component allowing it to intercept and alter inputs of that component. Figure 5.3 shows an example of a component which is wrapped by a fault block. Here the component's input may be pre-processed by the fault block before being passed to the component, the output from the component may be processed before release to the next part of the model. If a component is modelled using more than one submodel, then a fault block that intercepts the inputs and outputs of each submodel may be used.

Figure 5.3: Symbolic Paradigm of Fault Modelling

Note that such a modelled fault must be triggered in order to see the effects of that fault model in simulation results as shown in Figure. 5.3. In this thesis, we only consider persistent errors (Section 2.5.2). One can use a scripting language (Ni and Broenink, 2012) to activate faults, or one can physically inject faults, e.g. physical switches to turn off/on certain components.

The following sections will discuss how to perform fault modelling in the two tools: Overture and 20-sim.

5.2.2 Fault Modelling in Overture

As an example, consider an actuator modelled in VDM. The normative state and behaviour of the sensor is defined in a class, A. This class can be instantiated as an object that will exhibit normative behaviour. In order to introduce non-normative behaviour, a subclass A' can be defined, i.e. A' is subclass of A. An object of type A' will behave like an object of type A except where this functionality is explicitly overridden by the subclass. The A' class therefore contains the extra state (indicating that the object is

in an error state) and functionality (defining the non-normative behaviour) as show in Listing 5.1 and Listing 5.2, where a valve actuator is modelled as a class and a subclass is used to introduce a stuck valve as shown in Figure 5.4. This stuck valve is activated through DCL scripting language.

```

class ValveActuator

types
  ValveCommand = <OPEN> | <CLOSE>;

operations
  public Command: ValveCommand ==> ()
  Command(c) == duration(50)
  cases c:
    <OPEN>  -> i.SetValve(true),
    <CLOSE> -> i.SetValve(false)
  end
  post i.ReadValve() <=> c = <OPEN> and
       not i.ReadValve() <=> c = <CLOSE>

end ValveActuator

```

Listing 5.1: Explicit model of a valve in VDM

```

class ValveActuator

class ValveActuatorSticky is subclass of ValveActuator

instance variables
  private stuck : bool := false

operations
  public Command: ValveCommand ==> ()
  Command(c) == duration(50)
  if not stuck then
    cases c:
      <OPEN>  -> i.SetValve(true),
      <CLOSE> -> i.SetValve(false)
    end
  pre not stuck
  post i.ReadValve() <=> c = <OPEN> and
       not i.ReadValve() <=> c = <CLOSE>
  errs STUCK : stuck -> i.ReadValve() = ~i.ReadValve();

  private SetStuckState: bool ==> ()
  SetStuckState(b) == stuck := b
  post stuck <=> b and not stuck <=> not b;

end ValveActuator

```

Listing 5.2: A subclass that introduces a stuck valve actuator

Figure 5.4: A class diagram of the stuck valve

If an engineer wishes a component to exhibit more than one non-normative behaviour, he must explicitly create a class to represent this. For example, consider that an engineer has two non-normative implementations of an actuator, A' and A'' . If he wishes to simulate a sensor which can exhibit *both* non-normative behaviours, he must define a third class, A''' , which is a subclass of both A' and A'' . This can be triggered by scripting.

5.2.3 Fault Modelling in 20-sim

The approach to separate the normative and non-normative behaviour advocated above is to use submodels. Consider a simple block, which is implemented by a set of equations and has no submodels. Introducing non-normative functionality directly into this block would not achieve the same separation of functionality as the proposed in the VDM approach. In order to achieve separation, the block could be altered to include two submodels, where the first contains the normative behaviour and the second implements the non-normative behaviour. Additional submodels could be introduced as necessary. A separate implementation could therefore be created for each non-normative behaviour, in a similar fashion to the proposed use of subclasses in VDM. A 'no-fault' implementation would allow the signals to pass through it unaltered. Then alternate implementations are added, each mutating the data that passes through it as required by the fault it represents. The designer is then able to select the implementation of the fault block required for the simulation to be performed.

An example of a fault introduced in 20-sim is given in Figure 5.5. A leak in a valve is

modelled as an alternative route for water to flow from a tank to a drain, bypassing the valve. The rate of flow is a constant (K). In the example in Figure 5.5, the block representing the leak is clearly separate from the valve itself. Figure 5.6 is the script to activate the modelled fault.

Figure 5.5: A valve (in 20-sim) which can exhibit a leak

```
when time = 10 do
  ct Leak\leakActive := true
```

Figure 5.6: A script to activate the leak at 10s

5.2.4 Fault-Tolerant Cyber Software Design

In order to design the CPS more robust against possible faults, in this subsection, we introduce how to use a layered-structure to design controller software.

Fault tolerance means to avoid failures in the presences of faults. The procedure of fault modelling and designing controller software that handles these faults is as follows:

1. Identify the faults and make a model which can represent the faulty behaviour;
2. Determine the priority of the faults by analysing the consequence of the faults and design the corresponding fault-tolerant software.

Figure 5.7: Controller Layered Structure (Broenink and Ni, 2012)

We propose a controller structure (Figure 5.7) which helps to make the implementation "First-Time-Right". In Figure 5.7, the *Measurement and Actuation* block of the embedded control software denotes filtering and scaling to adapt the value ranges. The *Safety layer* block checks all signals going to and coming from the hardware. Safety issues are on all controller levels, shown by its U-shape in the figure. The *Safety layer* deals with the fault handling issues. More detail of the *Safety layer* is given in Figure 5.8. The *Loop control* block contains the control algorithms controlling the actuators. The *Sequence control* block is a kind of task level controller: it commands the loop controllers, by computing the setpoints of the control algorithms, and if applicable, enabling them and adapting parameters. The *Supervisory control* block is a strategy controller: calculations, often taking considerable amount of computing time (relative to the sampling period), that instruct the Sequence controller to determine its next task.

Figure 5.8: Safety Layer Components

As it is shown in Figure 5.8, there are three main blocks in the safety layer:

- Error detection: use the sensor values and reasoning algorithms to detect what fault has occurred and give a status of the sensors.
- Safety controllers: different safety controllers are available to take over control depending on the exact situation of the detected fault. Examples are: immediately stop the whole device; limit the outputs; bring the device to a safe state, etc. These strategies are device dependent.
- Decision maker: based on the severity of the fault, it decides which controller output is passed to the device, so either the steering value from the loop controller in the normal case or the value calculated by the selected appropriate safety controller.

All blocks can be computed at any time and decision maker only passes the values computed by the selected controller to the *Meas.& Actuation*. For performance reasons, the safety controller may only be calculated when needed. However, when such a safety controller needs the sensor values, it might be necessary to always calculate this safety controller to ensure a bumpless transfer of the steering value from the loop controller to the safety controller at the moment of selection. A case study on this is given in Section 6.2.

5.3 Conclusions

In this chapter, some advanced topics of co-modelling: design space exploration, fault modelling and fault-tolerant mechanism are introduced.

The design space exploration activity is support by automatic co-model analysis (ACA) facility. This enables DSE experiments to be carried out in a systematic manner. It also permits the systematic exploration of the design space with the aim of selecting models of optimal solutions, and allowing trade-offs between the computing and physical elements of alternative designs. Design alternatives can be distinguished according to the impact, i.e. the amount of changes that they have with respect to the co-model. An example is given in the mobile robot case study in Section 6.1.

A general fault modelling method is introduced. In this way, it is possible to include a fault in a model and simulate it. A layer structure of cyber software is presented, which can help to design a system that is more robust (can handle faults.) An example of adopting this fault-tolerant mechanism on a CPS setup is described in Section 6.2

Certainly, in a complicated system (as Qamar (2013) showed), there certainly are more than one variation. Effort on researching and developing different algorithms on finding the best combination of different variations can be beneficial. The targeted modelling error types can also be extended, not only for persistent errors, which are easier to be modelled.

6

Case Studies

Having described essentials methods and techniques for co-modelling using 20-sim and VDM in Chapter 4 and 5, we present two case studies that illustrate the methods and tooling introduced earlier from different aspects.

The first case study (Section 6.1) involves a mobile robot, which is an example that provides a good basis for illustrating the principles of co-model construction and design space exploration.

The second case study is a slider setup (Section 6.2), which illustrates the principles of co-model construction and fault-tolerant mechanism of controller software design.

6.1 Case 1: Mobile Robot

6.1.1 Goal

In this case study, we use a mobile robot example named R2-G2Px to demonstrate the contract-first co-modelling method (introduced in Section 4.5) and design space exploration activity (introduced in Section 5.1).

In the following sections, it follows the steps illustrated earlier in Chapter 4. Section 6.1.3 is Step 1, while Section 6.1.4 and Section 6.1.5 are Step 2a and Step 2b respectively.

6.1.2 Setup Description

Considering the team members of this case study (Pierce et al., 2012) were a good mixture of CT and DE modellers, this case study adopted the contract-first approach. The case study is based on a mobile robot as shown in Figure 6.1. This small mobile robot is designed that can be controlled by an embedded controller to follow lines on the floor. An example line for the robot to follow is given in Figure 6.2.

Figure 6.1: A photo of the mobile robot

Figure 6.2: An example of a line following pattern

In order to set up a dynamic model, the initial design concept is based on the following description:

- two wheels, which connect to servo motors that provide movement and differential steering;
- two infrared line-follow sensors, which sense the lines on the floor;
- two position encoders, one per wheel. Each encoder faces a 44-segmented (22 black and 22 white) disc attached to the wheel, allowing it to keep a count of how many segments have passed the encoder and in which direction;
- one robot body.

6.1.3 Top-Level Model

By applying the methods introduced in Chapter 4, we obtain the following top-level model as illustrated in Figure 6.3. The co-model of the robot consists of a DE controller

Figure 6.3: Top-Level model of mobile robot

model and a CT plant model. The CT model (Section 6.1.4) includes the dynamics elements of robot (the body and wheels), models of the sensors, and data representing

the line on the surface under the robot. The DE model (Section 6.1.5) reads data from the two line-following sensors and controls the speed of the two wheels. This suggests that four shared variables are required in the co-model: two monitored variables, one for each sensor; and two controlled variables, one for each wheel, as listed in the contract in Table 6.1. In this case, only shared variables are used (no events and shared design parameters are required). The PWM of the motors report the speed of the wheels, from -1 (clockwise full speed) to 1 (anti-clockwise full speed). The line-follow sensors report the "colour" of the floor, from 0 (black) to 255 (white).

Table 6.1: Co-simulation contract for the mobile robot co-model

Variables Type	Name	Type	Notes
controlled	velocity_Left	real	range: [-1,1]
	velocity_Right	real	range: [-1,1]
monitored	lfLeft	int	range: [0,255]
	lfRight	int	range: [0,255]

This co-model of the mobile robot includes two sensors, however, expansion to three (or more) sensors is discussed in Section 6.1.6.

6.1.4 Plant Model (CT Model)

Assuming that we only consider the planar movement (x, y direction) of the robot body, with the conceptual model as shown in Figure 6.4 in mind, we can analyse the dynamics of the robot as such, that it is represented in Figure 6.6.

Figure 6.4: Robot Dynamics

CoM is **C**enter **O**f **M**ass, the black dot stands for the center of rotation. Three more assumptions are as follows

- the wheels are rigidly attached to the shafts of the motors;
- all mechanical structures are infinitely stiff (such that we can adopt a rigid-body approach in modelling);
- the gearbox is assumed to be ideal (no backlash, no friction between the toothed wheels).

Hence, we use the Cartesian coordinate as shown in Figure 6.4. The relation of the angular velocity ω and the velocity of the wheels V_L and V_R can be represented as:

$$V_{R,y} = \frac{D}{2} \cdot \omega \quad (6.1)$$

$$V_{L,y} = -\frac{D}{2} \cdot \omega \quad (6.2)$$

The MTF block in Figure 6.6c represents the transformation from the local frame to the inertia frame following the rule in Equation 6.3.

$$\begin{bmatrix} V_{inertia,x} \\ V_{inertia,y} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} V_{loc,x} \\ V_{loc,y} \end{bmatrix} \quad (6.3)$$

Figure 6.5 shows the plant bond-graph model of the robot. This model includes continuous-rotation servo motors with speed control, wheels to translate the servo outputs to linear motion and a robot body that connects the two drive units and acts as a mass to be moved. Together these elements form the Plant submodel that can be seen in the top-level domain model (Figure 6.3).

Figure 6.5: Plant submodel of the mobile robot model

Transformation from rotation to translation that performed by wheels is modelled in Figure 6.6a. The servo motors are modelled as gyrators in Figure 6.6b.

(a) Wheel Dynamics in Bond Graph representation

(b) Bond Graph representation of servo motor

(c) Robot Body Dynamics in Bond Graph representation

Figure 6.6: The detailed dynamics of the robot

In order to perform simulations, the used parameters are listed in Table 6.2

Table 6.2: Physics parameters of the robot (Visser, 2011)

Parameter	Value
Inductance (L)	$1.71 \times 10^{-3} H$
Resistance (R_{el})	8.2Ω
Motor constant (k_m)	$0.796 Vs$
Bearing resistance (b)	$0.0126 Nm/s$
Wheel inertia ($J_w h$)	$12.44 \times 10^{-6} kg/m^2$
Wheel radius (r)	$0.03325 m$
Robot mass (m_{robot})	$0.414 kg$
Robot inertia (J_{robot})	$6.6 \times 10^{-4} kg/m^2$
Robot body width (W)	$0.074 m$
Robot body length (l)	$0.120 m$

6.1.5 Controller Model (DE Model)

Figure 6.7: R2-G2Px VDM controller (class and object diagram)

A class diagram of the controller is given in Figure 6.7. The core of the model is the Controller class, which defines a periodic thread that calls a Step operation to perform the

control loop. The line-following sensors and servos (actuators) are accessed through the Cyber class. Two objects of each type are used to access the four shared variables shown in Table 6.1.

6.1.6 Design Space Exploration

A 3D representation was created in 20-sim with a screen shot shown in Figure 6.8. This view also shows the path on the floor in black (the floor texture is purely decorative and is seen as 'white' by the robot). Note that although the sensors on the real robot are placed underneath the body, those in the 3D view are placed on the front (represented as spheres) for the convenience of giving visual feedback about the state of the sensors. The colour of the sphere represents the lightness of the floor as interpreted by the sensor (i.e. if the sensor sees black, the sphere is black).

Figure 6.8: 3D Representation of R2-G2Px in 20-sim, the two spheres representing the infrared sensors, while the colour of the sphere represents the status of the sensors

The concept of DSE (Design Space Exploration) and the general approach were given in Chapter 2 and Section 5.1. The co-modelling processes described in earlier chapters are also explained in detail in this chapter. The modelling of the robot is described above in Sections 6.1.3, 6.1.4 and 6.1.5. The Co-simulation approach and in particular evaluation of a set of design alternatives has been reported in (Ni et al., 2011).

However, certain design choices have already been made implicitly beforehand, such as the geometry of the robot ($10 \times 7 \times 7$ cm), and the fact that it has two wheels at given positions on its body. These choices restrict the design space as described in the design flow (Section 4.1). Within this design space, we can still consider a number of realistic design alternatives that could be made. These include:

- Wheel diameter.
- Servo power.
- Speed of control action.
- Number of line-follow sensors.
- Position of line-follow sensors.

As an extension to the current robot example, a number of new simulated designs are being considered, these designs vary five characteristic of the robot, as follows:

The choice of wheel diameter and servo power affects the dynamics behaviour of the robot. More powerful servos will allow the robot to move faster at the cost of using more energy. In turn, this will also affect the ability of the controller to move the robot and the speed at which it can do so. The frequency of the controller also affects the behaviour of the system. The trade-offs presented here are used as examples in this system. The choice between design alternatives and of which features to trade-off is up to the engineer and the requirements of the actual system being modelled.

With the facility provided by the co-simulation tool, it is possible to perform a parameter sweep experiment, where all combinations of parameters are simulated. This gives the most complete set of results and easier for the engineers to make decision for the optimal design.

One of the design choices that the robot can illustrate is that of the number of IR line-follow sensors it has. Figure 6.9 presents 3D visualisations of the robot with one, two and three line-follow sensors. With a single sensor, the controller has limited information about the position of the line. It must swing left-to-right then right-to-left while moving forward, looking for boundary crossings. White-to-black indicates that whether it has found the line and black-to-white indicates that it must switch direction to rediscover the line. With two sensors, the robot can move faster, turning towards the sensor that sees black. If both sensors are black, it can continue straight ahead. If both are white, it can recall the last sensor that saw black and turn in that direction to try and find the line again. Three sensors give the controller even more information. Then it can adjust the speed of its turn depending on the number of sensors that can see the line. However, there are costs to increasing the number of sensors, such as increased weight, power requirements and reduced space on the robot.

Figure 6.9: Three potential design alternatives for the mobile robot, with different numbers of sensors

Once, the choice of two sensors is made, the design space is restricted to only include those robots with two sensors. It is then possible to consider designs in which the placement of the sensors differs, for example, by increasing their lateral placement to the sides of the robot body or longitudinal placement to the front. Co-simulation and evaluation of various design alternatives with different sensor placements provided a good example to illustrate the ACA (Automated Co-model Analysis) features.

To demonstrate ACA on the mobile robot co-model, the position of the line-follow sensors (with respect to wheel axle) was varied in two-dimensions:

- *Lateral offset*: The distance of the line-follow sensors from the wheel axle along the width of the robot (i.e. to the side).
- *Longitudinal offset*: The distance of the line-follow sensors from the wheel axle along the length of the robot (i.e. to the front).

This co-model uses the contract as shown in Table 6.3. The encoders (encLeft and encRight) report the distance travelled by the wheels (in meters). The controlled variable (leds) allows the controller to set the state of RGB LEDs in the 3D robot model to give feedback on the current controller state. The two shared design parameters control (LF_lateral_offset, LF_longitudinal_offset) the position of the line-follow sensors with respect to the robot body.

	Name	Type	Default	Notes
controlled	velocity_Left	real		range: [-1,1]
	velocity_Right	real		range: [-1,1]
	leds	matrix		shape: [8,3]
monitored	encLeft	real		unit: meter
	encRight	real		unit: meter
	lfLeft	int		range: [0,255]
	lfRight	int		range: [0,255]
design parameters	LF_lateral_offset	real	0.01	unit: meter
	LF_longitudinal_offset	real	0.065	unit: meter

Table 6.3: Co-simulation contract for the R2-G2Px co-model Part II

Three values were selected for the each of these two parameters: [0.01, 0.03, 0.05] for lateral offset and [0.01, 0.07, 0.13] for longitudinal offset (both in meter). This gives a total of nine experiments. These are defined in Figure 6.10.

		Longitudinal sensor offset		
		0.01m	0.07m	0.13m
Lateral sensor offset	0.01m	(a)	(b)	(c)
	0.03m	(d)	(e)	(f)
	0.05m	(g)	(h)	(i)

Figure 6.10: The nine sensor position experiments with longitudinal offset (columns) and lateral offset (rows)

Figure 6.11 illustrates the nine positions of the sensors for the nine experiments (emphasising the left-hand sensors) from a top view.

In order to select the best design from these nine experiments in a more automated way, it is necessary to select some metrics which can then be ranked automatically. This is achieved using post-analysis of the co-simulation outputs. The following metrics were considered as ways of ranking these designs:

- *Maximum y-distance*: This is a measure of how much progress the robot made serving the purpose of the designed task: following the line. It is useful for normalising other metrics.
- *Speed*: This is a measure of how fast the robot travelled on average (the co-simulation time was set to 40 seconds).
- *Total energy*: This is a measure of the amount of energy used by the robot. Going further or faster will result in an overall increase in energy use.
- *Deviation area*: Calculate the area of the gaps between the path taken by the robot and the black line.
- *Maximum deviation*: This is a measure of the maximum distance that the robot strayed from the line.

Figure 6.11: The nine sensor positions visualised as a top view of the robot

The raw measurement data for each design are given in Table 6.4, however they are not currently ranked. Inspection of these metrics can give some idea of how well the robot performed. For example, judging by "Max. y-distance", design (b) travelled the furthest and design (c) the least far. Design (g) has a very large deviation area and maximum deviation (indeed, it lost the line and turned round). Note however that the measures for deviation area and energy are dependent on how far the robot travelled.

Design	Max. y-distance	Speed	Energy	Deviation area	Max. deviation
a	1.26 <i>m</i>	0.0316 <i>m/s</i>	32.8 <i>J</i>	0.033 <i>m</i> ²	0.0433 <i>m</i>
b	1.32 <i>m</i>	0.033 <i>m/s</i>	29.6 <i>J</i>	0.03 <i>m</i> ²	0.0388 <i>m</i>
c	1.12 <i>m</i>	0.0279 <i>m/s</i>	16.2 <i>J</i>	0.0744 <i>m</i> ²	0.115 <i>m</i>
d	0.892 <i>m</i>	0.0223 <i>m/s</i>	18.9 <i>J</i>	0.065 <i>m</i> ²	0.0743 <i>m</i>
e	1.19 <i>m</i>	0.0298 <i>m/s</i>	27.1 <i>J</i>	0.0412 <i>m</i> ²	0.0698 <i>m</i>
f	0.717 <i>m</i>	0.0179 <i>m/s</i>	8.39 <i>J</i>	0.0254 <i>m</i> ²	0.0312 <i>m</i>
g	0.604 <i>m</i>	0.0151 <i>m/s</i>	15.7 <i>J</i>	0.0711 <i>m</i> ²	0.0708 <i>m</i>
h	1.14 <i>m</i>	0.0284 <i>m/s</i>	27.4 <i>J</i>	0.0913 <i>m</i> ²	0.124 <i>m</i>
i	0.473 <i>m</i>	0.0118 <i>m/s</i>	4.8 <i>J</i>	0.0207 <i>m</i> ²	0.0393 <i>m</i>

Table 6.4: Raw measurement data

Design	Speed	Energy per meter	Deviation per meter	Max. deviation
a	0.0316 <i>m/s</i>	26 <i>J/m</i>	0.0261 <i>m</i> ² / <i>m</i>	0.0433 <i>m</i>
b	0.033 <i>m/s</i>	22.4 <i>J/m</i>	0.0227 <i>m</i> ² / <i>m</i>	0.0388 <i>m</i>
c	0.0279 <i>m/s</i>	14.5 <i>J/m</i>	0.0666 <i>m</i> ² / <i>m</i>	0.115 <i>m</i>
d	0.0223 <i>m/s</i>	21.1 <i>J/m</i>	0.0729 <i>m</i> ² / <i>m</i>	0.0743 <i>m</i>
e	0.0298 <i>m/s</i>	22.8 <i>J/m</i>	0.0346 <i>m</i> ² / <i>m</i>	0.0698 <i>m</i>
f	0.0179 <i>m/s</i>	11.7 <i>J/m</i>	0.0355 <i>m</i> ² / <i>m</i>	0.0312 <i>m</i>
g	0.0151 <i>m/s</i>	26 <i>J/m</i>	0.118 <i>m</i> ² / <i>m</i>	0.0708 <i>m</i>
h	0.0284 <i>m/s</i>	24.1 <i>J/m</i>	0.0804 <i>m</i> ² / <i>m</i>	0.124 <i>m</i>
i	0.0118 <i>m/s</i>	10.1 <i>J/m</i>	0.0438 <i>m</i> ² / <i>m</i>	0.0393 <i>m</i>

Table 6.5: Measurement data with energy and deviation normalised by maximum y-distance

For example, the energy usage for design (i) are very low, since it did not travel very far. Similarly, by travelling further, other designs have more chance to lose the line and increase their deviation area. Therefore, before considering the ranking of designs, it is useful to normalise the energy and deviation area by distance travelled. The results are presented in a revised set of metrics in Table 6.5.

Table 6.5 contains four derived metrics by which we might rank the nine robot designs. These are: average speed; energy used (per meter of progress), deviation from the line (per meter of progress); and maximum deviation from the line. Taking each of these in turn, the designs can be ranked according to each metric (single-attribute ranking). The results of these simple ranking are given in Tables 6.6a–6.6b, where the two tables show the best design for each metric.

From these tables, it can be seen that the various designs have pros and cons. Considering the speed alone, designs (b) and (a) are the fastest (Table 6.6a). Note that by purely looking at energy usage (Figure 6.6b), designs (i) and (f) are very good (even when considering the short distances they cover). Looking at the shape of the line shows that they actually spent much of the time rotating on the spot while trying to

Rank	Design	Speed
1	b	0.033 <i>m/s</i>
2	a	0.0316 <i>m/s</i>
3	e	0.0298 <i>m/s</i>
4	h	0.0284 <i>m/s</i>
5	c	0.0279 <i>m/s</i>
6	d	0.0223 <i>m/s</i>
7	f	0.0179 <i>m/s</i>
8	g	0.0151 <i>m/s</i>
9	i	0.0118 <i>m/s</i>

(a) Designs ranked by speed

Rank	Design	Energy per meter
1	i	10.1 <i>J/m</i>
2	f	11.7 <i>J/m</i>
3	c	14.5 <i>J/m</i>
4	d	21.1 <i>J/m</i>
5	b	22.4 <i>J/m</i>
6	e	22.8 <i>J/m</i>
7	h	24.1 <i>J/m</i>
8	a	26 <i>J/m</i>
9	g	26 <i>J/m</i>

(b) Designs ranked by energy per meter

Table 6.6: Ranking

find the line (since their sensors are far from the robot body), so did not have time to reach a higher speed (that would use more energy). If the extra requirement was the lowest energy use possible, design (f) would be the best (still considering the furthest distance it travelled).

6.1.7 Conclusion and Evaluation

In this case study, details of the mobile robot have been given. The step-wise refinement method of co-modelling, such as creating a co-model, CT modelling and DE modelling has been demonstrated through this case study. The possible design space exploration of the robot is discussed in Section 6.1.6.

This case study is a demonstration for DSE concepts from Chapter 5 and an illustration to bring ACA features of the DESTecs tool and produce a proof-of-concept for post-analysis of results and ranking functions.

We can see that the methods permit the systematic exploration of the design space with the aim of selecting models of optimal solutions, and allowing trade-offs between the computing and physical elements of alternative designs. This set of methods meet "Goal 1" in Chapter 1.

6.2 Case 2: Slider Setup

6.2.1 Goal

In this section, we use a mechatronic device, called *slider*, to demonstrate the fault modelling, fault injection and fault-tolerant mechanisms methods introduced in Section 5.2.

Besides the focus of demonstrating these fault-tolerant mechanism, the procedures of preforming co-design of this setup follow the normal procedure of a co-design proposed in this thesis as well. In the following sections, the order of steps introduced earlier in Chapter 4 is followed. Section 6.2.3 is Step 1, while Section 6.2.4 and Section 6.2.5 are Step 2a and Step 2b respectively.

6.2.2 Setup Description

The demo setup was designed to demonstrate a typical mechatronic systems in practical situations, is shown in Figure 6.12a. This setup originates from the principle of a printing device: a slider moving back and forth over a rail guide which is controlled by an embedded computer (PC block in Figure 6.12). The frame of the system is flexible, which introduces vibrations in the setup when the slider accelerates, see Figure 6.13. The device has six sensors in total: one motor position encoder, one position sensor with respect to the fixed world and two position sensor with respect to the frame (upper and lower), two endstops (left and right), as shown in Figure 6.13.

Figure 6.12: The slider setup (Dirne, 2005)

The lowest eigenfrequency of the fourth-order dynamic system is at a visible range, which is smaller than 15 Hz.

Figure 6.13: The Dynamic Plant Part of the Slider

6.2.2.1 Task

The specific task for the slider demo in this case study focused on the tasks: hardware self-protection and software self-protection. For the software safety layer, it is important to protect the setup from unwanted controller behavior. In this specific case it is when the slider moves out of bounds. In order to visualize the slider co-model, a 3D representation was created as shown in Figure 6.12b.

The key features of the slider that are relevant to this task are:

- the two endstops' position
- motor sensor
- linear strip

6.2.3 Top-Level Model

By applying the co-modelling methods mentioned earlier, we can have the following top-level model in Figure 6.14. The co-model of the slider consists of Controller block, a Plant block and a IO in between. The plant block contains the slider dynamics model. The Controller block reads data from sensors to determine the position of the slider to control the motor servo. This suggests seven variables that should be exchanged between the plant and the controller via the IO, the shared variables: one motor steering variable and the six sensor values.

Figure 6.14: Top-level of the Slider

6.2.4 Plant Model (CT Model)

Figure 6.15: Slider dynamic model

The details of the Plant block from Figure 6.14 are shown in Figure 6.15, with parameters values listed in Table 6.7. This is a dynamic model which is based on the basic fourth-order mass-spring-mass system. As both the flexibility of the frame and belt is taken into account, this model is a sixth-order model. Including Coulomb friction of the damping phenomena makes this model non-linear. Figure 6.15 shows a IPM (Ideal Physical Model) which can be seen as a domain-specific bond graph. In this model, the top and bottom position sensors are considered one, because the slider stays up-

right during movement, thus these sensors read the same value all the time. Hence in Figure 6.14, the signal connection from Plant to IO has a multiplicity of 5. The signal connection from IO to Controller has a multiplicity of 6, since here, all four encoders are considered (next to the two end switches). The coloured ellipses labelled with letters indicate the components as listed in Table 6.7.

Components	Parameter	Value
Frame (a)	Mass of frame	$0.8kg$
	Spring constant	$4.4KN/m$
	Damping	$4.4Ns/m$
Motor (b)	Motor mass	$1e^{-5}kg$
Belt (c)	Spring constant	$800N/m$
	Damping	$1Ns/m$
Slider (d)	Mass	$0.3kg$
Rail (e)	Viscous friction	$3Ns/m$
	Coulomb friction	$0.5N$

Table 6.7: Mechanical parameters specification for the setup

6.2.5 Controller Model (DE Model)

The VDM implementation is represented in the class diagram shown in Figure 6.16.

Figure 6.16: Slider VDM controller (class and object diagram)

From a signal point view, the controller block in Figure 6.14 can be detailed in a diagram as shown in Figure 6.17.

Figure 6.17: Safety Layer Structure

6.2.6 Fault Modelling and Fault Tolerance Mechanisms

Scenarios of completely failure of sensor were investigated for this setup. There are four sensors considered: 1 motor position encoder, 1 position sensor with respect to the fixed world and 2 position sensors with respect to the frame (upper/lower sensor). In order to detect the malfunctioned sensors, we use a matrix as given below:

$$\text{Comparison Matrix} = \begin{bmatrix} |FX - FX| & |FX - SB| & |FX - ST| & |FX - MP| \\ |SB - FX| & |SB - SB| & |SB - ST| & |SB - MP| \\ |ST - FX| & |ST - SB| & |ST - ST| & |ST - MP| \\ |MP - FX| & |MP - SB| & |MP - ST| & |MP - MP| \end{bmatrix} \quad (6.4)$$

where FX , SB , ST and MP denote the values from sensors FixedWorld, SliderBottom, SliderTop and MotorPosition accordingly. So if the FixedWorld sensor was broken, the difference comparing to the other sensors would be out of the normal range: e.g. $|SB - FX|$, $|FX - ST|$ and $|FX - MP|$ are all out of range. So, we can deduce that the FixedWorld sensor is broken. In this particular set up, a switch (as shown in Figure 6.18), disabling the sensor's power, was made for each of the sensors to mimic the sensor failure: when a sensor loses power, then that sensor stops functioning, so applying the switch means injecting a fault into the system.

Figure 6.18: Switches for Physical Faults Injection (van de Vijver, 2014)

Based on the method introduced in Section. 5.2, we can structure the controller software of the slider as indicated in Figure 6.19 and Figure 6.20. In detail, Figure 6.19 shows a layered structure of the overall controller software.

Figure 6.19: Controller Implementation (Layered structure)

Figure 6.20 shows the components of the slider safety layer: *Detect broken sensors*, *Different safety controllers*, *Safety decision making*. Detecting broken sensors: identifying how many sensors are broken, by using the algorithm mentioned above. Different safety controllers: panic controller (the slider stops moving immediately); safety controller (limit the motor output, reduce the motor speed); homing controller (move the

slider to the safe area, the middle of the frame, in this case). Safety decision making: based on the sensors status, the decision maker decides which controller to enable.

Figure 6.20: Safety Layer Components on the Slider set up

Figure 6.21: Slider position comparison with safety layer and without safety layer (measurements from the setup)

In Figure 6.21, it is shown that with the aid of the used safety layer in the controller software, the system can react more robust. As is indicated in Figure 6.21, the slider

can move in between ± 10 cm (in between the endstops while the origin point is the middle of the rail).

Figure 6.22: Slider position together with broken sensors fault scenarios. scenario A: sensor SB broken; detection A: detection of fault A; scenario B: sensor ST broken; detection B: detection of fault B; scenario C: sensor FW broken; detection C: detection of fault C.

Note that due to the fact that the DESTECs tool has not yet facilities to generate C-code from the VDM controller models, we have specified the controller in 20-sim, according to the structure give in Figure 6.19. Using the automatic code generation feature of 20-sim and the deployment and experiment tool 20-sim 4C¹, we have conducted experiments on the real system, of which the key result is shown in Figure 6.21 and Figure 6.22.

6.2.7 Results Analysis

In Figure 6.21 and Figure 6.22, it is shown that with the aid of the used safety layer in the controller software, the system can react more robust when faults are injected into

¹<http://www.20sim.com/products/20sim4c.html>

the system. As indicated in Figure 6.21, in between the green dashed lines ($\pm 10\text{ cm}$) (the origin point is the middle of the rail) are the safe range that the setup can operate. The slider can move in between the endstops with the aid of the safety layer. In Figure 6.22, the grey dotted vertical line indicates when a sensor error is injected while the green dotted vertical line indicates when the software safety layer detects the broken sensor. It can be seen that even when the system has faults injected, it can still operate normally: in this case, the slider still moves within the safety zone (in between the endstops). In this case, the *safety controller* and *homing controller* (activated when two sensors are broken) from Figure 6.20 were activated sequentially.

6.2.8 Conclusion and Evaluation

In this case study, details of the slider have been given. A co-design procedure has been demonstrated through this case study. Details about fault modelling and how to design a robust software to cope with possible faults are discussed in Section 6.2.6.

From this case study, we can see that by using the proposed set of methods: fault modelling, fault injection, a fault-tolerant cyber-part software can sufficiently structured. Applying the co-modelling methods, both fault modelling and fault-tolerant mechanisms are beneficial in managing the additional design complexity introduced by the need for confidence in correct and safe functioning of CPS devices. This set of methods meet "Goal 2" in Chapter 1.

It is recommended to implement the code generation functionality to the co-simulation tool. As it has been shown in the slider case study, in order to perform test and analysis experiments on the real setup, the code generation function from 20-sim has been used.

6.3 Industrial case studies in the DESTecs project

The co-modelling approach and co-simulation tool introduced in this thesis have also been tested by the DESTecs industrial partners (Verhoef et al., 2012). Their applications are: a dredging excavator (Verhaert N.V.), a document handling system (Neopost) and a self-balancing scooter (CHESS). Those applications are very different in nature, thus different approaches were adopted. In Verhaert's application dynamic-behaviour oriented approach is adopted, Neopost adopted controller-logic oriented approach while CHESS application used all three approaches: controller-logic, dynamic-behaviour oriented and contract oriented.

In the case study of Verhaert, it was reported that the co-modelling approach and co-simulation tool enabled it to efficiently validate their emergency supervisor of the excavator when some changes need to be made to their controller to accommodate some unexpected shortcomings. In the end, a fully working version of their "assisted mode" excavator operation on their scale model test setup has been reached.

In the case study of Neopost, with respect to the system under development, Neopost identified and solved, due to the co-modelling work performed, several flaws in the system design. These defects were discovered even before physical parts of the system were realised, which saved several weeks of development time.

In the case study of CHESS, there are numerous faults which have been explored by developing and co-simulating a ChessWay co-model. By using co-modelling approach and co-simulation tool, it was feasible to make a trade-off between safety, functionality and cost price.

By applying the methods and tool in those applications, it is shown that the methods can also be applied to scaled-up applications.

7

Conclusion and Outlook

System-level design support for cyber-physical systems becomes rather challenging due to the separation of engineering disciplines in the current development process. Increasingly complex CPS devices are currently being designed and built in industry using ad-hoc approaches. To tackle this complexity, in this thesis, solutions for CPS system-level design support have been proposed .

The co-modelling methods proposed in this thesis aim to provide a system-level way of considering and designing CPS instead of from a conventional single-domain perspective. It aims to help engineers to make rational design choices in the early design phase, and also to reduce possible design flaws and expensive rework. The co-simulation framework has been designed to provide a software environment for different domain expertise working together.

The co-design flow and associated co-modelling methods have been made to provide the designers a system-level way of designing CPS by using the facilities of the tooling framework.

This chapter summarizes the contributions and lists possible future work on the topics discussed in this thesis.

7.1 Contributions

The main contribution of this thesis are:

- *Methods*

A co-design workflow and associated co-modelling methods have been provided to help designers to understand the essentials of designing CPS as a whole. Design space exploration, fault modelling and fault-tolerant mechanisms have been discussed in detail with demonstrations of case studies. Together, these approaches are aimed at designing CPS devices "First-Time-Right", both from a fault prevention and fault-tolerant perspective.

- *Co-Design Workflow*

The co-design workflow provides detailed co-modelling steps for different expertise to understand CPS design from a system level, instead of only considering

one single specific discipline. It incorporates the existing standards and takes the multi-disciplinary nature of CPS into account.

These co-design support guidelines can be applied by potential designers with different backgrounds to design various CPS with different system properties. Three differently focused co-modelling approaches (dynamics-behaviour oriented approach, controller-logic oriented approach and contract-first approach) that are introduced in this thesis have their own advantages and disadvantages. The designers can choose the appropriate approach based on the particular goals of the projects. These approaches have been tailored to the co-simulation framework proposed in this thesis.

- *Design Space Exploration*

The designers can make explicit early-stage design choices better, having the design space exploration tool support. A systematic approach of design space exploration has been described with support of the mobile robot case study. A classification for different design alternatives of co-models has also been given.

- *Fault-tolerant Mechanism*

It is feasible to design system control software (following the layered structure of controller software) that can react more robust even when faults are injected into the system. The fault modelling and fault tolerance mechanism of the controller software have been introduced in Chapter 5 and have been demonstrated using the slider setup case study. This allows for designing and testing controller software that can handle faults, i.e. make the software more robust.

- *Tooling Support: Co-simulation Framework*

The co-simulation engine synchronization scheme, which makes the time synchronization between two domain specific simulators possible, has been proposed. This synchronization scheme has been implemented in the co-simulation software framework, which contributes to the feasibility of a tool support for CPS design. In addition to that, a DCL script language has been designed. It also enables the possibility of performing fault injections, which makes the designing of fault-tolerant cyber software more intuitive. It is executed as a scenario of a co-model, which provides the variations of scenario and the possibility of changing a co-model during a co-simulation when it is necessary.

7.2 Recommendations for Future Work

Co-Design Workflow

The co-design flow and co-modelling methods need to be verified on more different CPS case studies. It can also be of interest to apply this method with other commercial off-the-shelf tools to abstract away the particularities from the tools and result in a set of more generalized methods.

Design Space Exploration

The ACA tool support can be enhanced by implementing more sophisticated higher-dimensions optimization algorithms, as the current implementation is just using single dimension optimization.

The tool support for design space exploration are based on the concept of co-model and co-modelling. The case study only considered one variation (IR sensor positions) on the alternatives. By performing experiments and comparison, it gives the designer a more illustrative impression of the alternatives. Certainly, in a more complicated system, there might be more than one variation. Effort on developing certain algorithms on finding the best combination of different variations can be of interest.

Fault-Modelling

It can be beneficial to extend the class of supporting faults to also intermittent and transient errors. In the thesis, with fault modelling of the various abnormal phenomena, the type of error supported are normally considered persistent. In the supporting case study, only the persistent sensor errors were considered.

Co-Simulation Tool Framework

Some specific tool implementations can be improved for the current co-simulation framework, such as implementing the code generation facility of VDM (e.g. Maimaiti, 2011). This is useful for the designers to validate the designs.

Extension to other tools can also be an improvement for the current co-simulation framework. This co-simulation framework and synchronization scheme, introduced in this thesis, only covers the case of two different tools (20-sim and Overture) combined together. This can be extended to the case of two different tools other than 20-sim and Overture. This can also be extended to the case of more than two different tools, e.g.: three different tools combined together. This case can be complicated to deal with already.

It is beneficial to have standard protocols for future co-simulation frameworks. In principle, the way of "co-simulation thinking" can be quite useful in the current design software development process. Nowadays different software tools often need to exchange information with other tools. Like in automobile industry, they advocate Function Mock-Up Interface (Modelisar, 2010) as a standard interface among different suppliers to exchange models. This kind of approach of using a standard interface or protocol to exchange model information can also be applied to other business, such as building construction designing, health-care devices designing etc.

7.3 Final Word

Cyber-Physical Systems is still a relatively young and changing research area. This thesis is focused on design support for cyber-physical systems from a system-level point of view. This work can be considered as a contribution to modelling-driven engineering. By using this work, the designers can have a new instrument for their daily work.

However, certainly, this is a little step in the CPS system-level design support, a lot of work still need to be done.

Thinking of CPS devices on a larger scale and higher complexity can bring us to further integration of a wider range of models. The possible application topics can be human-car aid driving (the designed software can be tolerant to the mis-operations by humans: as error injections) or network of self driving cars (where both the dynamics of individual cars and the interaction of the individual cars are needed to be considered). Providing methods and tools to support such applications has enormous potential benefit in systems development and engineering.

Together, such techniques could help us more rapidly to develop cyber-physical products and also earn our trust.

DCL Language Definition

This appendix specifies DCL. Section A.1 introduces the proposed language constructs. Section A.2 presents the abstract syntax and context conditions for DCL in VDM-SL. Finally, Section A.3 gives a concrete syntax in EBNF.

A.1 DCL Language Constructs

In this section, concrete syntax is given with keywords shown in bold and placeholders for clauses shown in italics.

A.1.1 Script

A script is contained in a text file and consists of a set of **when** statements, which act as a set of condition-action pairs. The **include** statement can be used to parse **when** statements from other files.

The **when** and **include** statements are top-level statements and the only statements which may appear by themselves in a script. All other statements must appear within a **when** statement. Conversely, **when** and **include** statements cannot be nested within a **when** statement. Top-level statements are separated by semicolons.

A.1.2 Top-Level Statements

A.1.2.1 When Statement

The **when** statement defines a condition-action pair and has the following form:

```
when condition do  
    statement1  
    [after  
        statement2]
```

When *condition* becomes true, the **do** clause will execute *statement1* exactly once. If *condition* becomes false again, the optional **after** clause will execute *statement2*. Once *condition* has become false, *statement1* will execute again if *condition* becomes true again (and so on).

A.1.2.2 Include Statement

The **include** statement has the following form:

```
include "filename"
```

This statement causes the contents of the given *filename* to be parsed exactly as if its contents had been inserted in the current file at this point. The *filename* must exist in the filesystem. The included script may also contain **include** statements, as long as there are no circular references.

A.1.3 Statements

The following statements may only appear within the **do** or **after** clause of a **when** statement.

A.1.3.1 Assignment Statement

The assignment statement has the following form:

```
id := expression
```

This statement is used to change the values of design parameters in the co-model. The left-hand side must be a known identifier within the constituent models (non-shared design parameter) or contract (shared design parameter). The right-hand side must be an expression of the same type as the left-hand side. See Section for a description of expressions.

A.1.3.2 Block Statement

The block statement allows one or more statements to be grouped together as a sequence, for inclusion in the **do** or **after** clause of a **when** statement. It does not however allow local variables to be defined. Blocks of statements are enclosed in parentheses and separated by semicolons, e.g.

```
(  
    statement_1;  
    statement_2;  
    ...,  
    statement_n  
)
```

A.1.3.3 Revert Statement

The **revert** statement has the following form:

```
revert id
```

It is a special statement that may only appear in an **after** clause. If an *id* is assigned to in a **when** clause, the **revert** statement can be used to change the value of *id* back to what it was immediately preceding the assignment. In the following example, assuming that $x = 5$ at **time** = 5, then $x = 5$ will be true again after **time** = 10:

```
when time > 5 and time < 10 do
  x := 10
after
  revert x
```

A.1.3.4 Print Statement

The **print** statement has the following form:

```
print "message"
```

This statement causes the formatted string, *message*, to be printed to the console of the tool (or to be logged during batch execution mode). It can be used by the engineer for simple debugging of co-simulations. The engineer can choose to suppress **print** messages (causing them not to be displayed or logged) by altering the settings of the scenario.

A.1.3.5 Quit Statement

The **quit** statement has the following form:

```
quit
```

This statement causes the co-simulation to terminate. It should be used for normal termination, for example, when a given state has been reached:

```
when de controller.jobdone = 50 do
  quit
```

Stopping the termination at a given time can be controlled in the settings of the scenario, or through an event in the script.

A.1.4 Types and Expressions

A.1.4.1 Boolean Type

The following operators are available and all yield expressions of type boolean:

Operator	Name	Signature
not b	Negation	bool $\rightarrow$ bool
a and b	Conjunction	bool * bool $\rightarrow$ bool
a or b	Disjunction	bool * bool $\rightarrow$ bool
a => b	Implication	bool * bool $\rightarrow$ bool
a <=> b	Biimplication	bool * bool $\rightarrow$ bool
a = b	Equality	bool * bool $\rightarrow$ bool
a <> b	Inequality	bool * bool $\rightarrow$ bool
a for b	Duration	bool * time $\rightarrow$ bool

The only interesting addition is the **for** operator, which yields true if the condition, a, has held for a continuous period of time, b, (and false otherwise).

A.1.4.2 Numeric Types

The language supports integer and real types (where integers are included in the real type). The following operators are available for numeric types and all yield expressions of type real:

Operator	Name	Signature
-x	Unary minus	real $\rightarrow$ real
abs x	Absolute value	real $\rightarrow$ real
x + y	Sum	real * real $\rightarrow$ real
x - y	Difference	real * real $\rightarrow$ real
x * y	Product	real * real $\rightarrow$ real
x / y	Division	real * real $\rightarrow$ real
x ** y	Power	real * real $\rightarrow$ real

The following operators are available for numeric types and all yield expressions of type integer:

Operator	Name	Signature
x div y	Integer division	int * int $\rightarrow$ int
x mod y	Modulus	int * int $\rightarrow$ int
floor x	Floor	real $\rightarrow$ int
ceil x	Ceiling	real $\rightarrow$ int

The following operators are available for numeric types and expressions of type boolean:

Operator	Name	Signature
<code>x < y</code>	Less than	<code>real * real → bool</code>
<code>x > y</code>	Greater than	<code>real * real → bool</code>
<code>x <= y</code>	Less or equal	<code>real * real → bool</code>
<code>x >= y</code>	Greater or equal	<code>real * real → bool</code>
<code>x = y</code>	Equal	<code>real * real → bool</code>
<code>x <> y</code>	Not equal	<code>real * real → bool</code>

Note: Comparison of real numbers as approximated by floating point values is never exact. The co-simulation engine will take this into account by internally handling tolerance (epsilon) in floating point comparisons. Settings could be provided to allow power users to control this behaviour.

A.1.4.3 Time Type

The time type is similar to the real type, except that it must not be negative. The expression `time` is of type time and yields the current time in the co-simulation.

Time literals may be followed by an optional unit, given in curly braces, e.g. `{ms}`. If no unit is provided, it is assumed to be in seconds. The following units can be given:

Unit	Contraction
<code>microseconds</code>	<code>us</code>
<code>milliseconds</code>	<code>ms</code>
<code>seconds</code>	<code>s</code>
<code>minutes</code>	<code>m</code>
<code>hours</code>	<code>h</code>

The operators defined for the numeric types are valid for time types (except for `abs` and unary minus).

A.1.4.4 Identifier Expressions

Identifiers refer to named values in the constituent models (design parameters) or contract (shared design parameters) of the co-model that the script is controlling during a co-simulation. The keywords `de` and `ct` can prefix an identifier in order to define the constituent model in which the value resides. The syntax of identifiers follows the rules of VDM and 20-sim for `de` and `ct` identifiers, respectively. Where no prefix is given, it is assumed that the value is in the contract (shared design parameter).

A.1.4.5 Formatted Strings

Messages output from the script using `print` and `error` statements can be given as formatted strings (as in `printf` in C, in the style of Python), e.g.

```
"format string" % id_1, id_2, ..., id_n
```

This causes *"format string"* to be output with any placeholders (e.g. "%s") replaced by the corresponding *id* from the list of identifiers.

A.2 Abstract Syntax and Context Conditions

The following listing presents formal definition for DCL in VDM-SL, giving an abstract syntax and context-conditions. The context-conditions define those scripts in the abstract syntax which are valid using a function *wf_Script*. This is mainly involves type-checking. Note that this function needs to know the types of identifiers in the co-model and so a new type, *CoModelIds*, is introduced to hold this information. Also note that this current definition *does not* check if the script is allowed to access identifiers in the co-model, but merely that the identifiers exist and are of the correct type.

```

module DCL

imports from Framework
types Type

exports
types Script; struct CoModelIds; struct Id
functions wf_Script: Script * CoModelIds -> bool

definitions

types

-- script

Script = set of TopLevelStmt;

TopLevelStmt = ImportStmt | WhenStmt;

ImportStmt :: path : token;

WhenStmt :: condition : Expr
           action : Stmt
           after : [Stmt];

-- statements

Stmt = BlockStmt | AssignStmt | RevertStmt | LogStmt | <Quit>;

BlockStmt = seq of Stmt;

AssignStmt :: id : Id
            e : Expr;

RevertStmt :: id : Id;

LogStmt = PrintStmt | ErrorStmt;

PrintStmt :: msg : FormatString;
ErrorStmt :: msg : FormatString;

```

```

-- expressions

Expr = bool | real | Time | <Time> | Id | UnaryExpr | BinaryExpr;

Time :: value : real
      unit : [TimeUnit]
inv t == t.value > 0;

TimeUnit = <Microsecond> | <Millisecond> | <Second> |
           <Minute> | <Hour>;

Id :: model : <DE> | <CT>
    id : token;

UnaryExpr :: op : <Not> | <Minus> | <Abs> | <Floor>
           e : Expr;

BinaryExpr :: a : Expr
            op : <And> | <Or> | <Implies> | <Biimplication> |
                <Equals> | <NotEqual> | <Plus> | <Minus> |
                <Multiply> | <Divide> | <Pow> | <LessThan> |
                <GreaterThan> | <Leq> | <Geq> | <For>
            b : Expr;

FormatString :: str : seq of char
              args : [seq of Id];

-- type checking

Type = Framework`Type | <TIME_TP>;

CoModelIds ::      de : map Id to Type
                  contract : map Id to Type
                  ct : map Id to Type;

values

BoolOps = {<And>, <Or>, <Implies>, <Biimplication>, <Equals>,
           <NotEqual>, <For>, <LessThan>, <GreaterThan>,
           <Leq>, <Geq>};

NumericOps = {<Plus>, <Minus>, <Multiply>, <Divide>, <Pow>};

-- context conditions

functions

-- script

wf_Script: Script * CoModelIds -> bool
wf_Script(s, ids) ==
  forall tls in set s & wf_TopLevelStmt(tls, ids);

wf_TopLevelStmt: TopLevelStmt * CoModelIds -> bool
wf_TopLevelStmt(s, ids) ==
  cases s:

```

```

-- when statement
mk_WhenStmt(condition, action, after) ->
  typeof(condition, ids) = <BOOL_TP> and
  wf_Expr(condition, ids) and
  wf_Stmt(action, ids) and
  assigns(action) = {} and
  after <> nil => (wf_Stmt(action, ids) and
    reverts(after) subset assigns(action)),

-- import statement
mk_ImportStmt(-) -> true
end;

-- statements

wf_Stmt: Stmt * CoModelIds -> bool
wf_Stmt(s, ids) ==
  cases s:
    -- assignment statement
    mk_AssignStmt(id, e) ->
      idknown(id, ids) and
      typeof(id, ids) = typeof(e, ids) and
      wf_Expr(e, ids),
    -- revert statement
    mk_RevertStmt(id) ->
      idknown(id, ids),
    -- print statement
    mk_PrintStmt(mk_FormatString(-,args)) ->
      forall i in set inds args & idknown(args(i), ids),
    -- error statement
    mk_ErrorStmt(mk_FormatString(-,args)) ->
      forall i in set inds args & idknown(args(i), ids),
    -- quit statement
    <Quit> -> true
  end;

-- expressions

wf_Expr: Expr * CoModelIds -> bool
wf_Expr(e, ids) ==
  cases e:
    -- unary expression
    mk_UnaryExpr(op, expr) ->
      wf_Expr(expr, ids) and
      op in set {<Not>} =>
        typeof(expr, ids) = <BOOL_TP> and
      op in set {<Minus>, <Abs>, <Floor>} =>
        typeof(expr, ids) in set {<REAL_TP>, <TIME_TP>},
    -- binary expression
    mk_BinaryExpr(a, op, b) ->
      wf_Expr(a,ids) and
      wf_Expr(b,ids) and
      op in set {<And>, <Or>, <Implies>, <Biimplication>} =>
        (typeof(a,ids) = <BOOL_TP> and
          typeof(b,ids) = <BOOL_TP>) and
      op in set
        NumericOps union

```

```

    {<LessThan>, <GreaterThan>, <Leq>, <Geq>} =>
    {typeof(a, ids), typeof(b, ids)} subset
    {<REAL_TP>, <TIME_TP>} and
op in set {<Equals>, <NotEqual>} =>
    typeof(a,ids) = typeof(b,ids) and
op = <For> =>
    (typeof(a,ids) = <BOOL_TP> and
    typeof(b,ids) = <TIME_TP>)
others ->
    -- simple expressions
    is_real(e) or
    is_bool(e) or
    is_Time(e) or
    e = <Time> or
    idknown(e,ids)
end;

-- auxiliary functions

-- return the type of an expression
typeof: Expr * CoModelIds -> Type | <ERROR>
typeof(e, ids) ==
cases e:
mk_UnaryExpr(op, a) ->
    if op = <Not> then <BOOL_TP>
    else typeof(a,ids),
mk_BinaryExpr(a, op, b) ->
    if op = <For> or op in set BoolOps
    then <BOOL_TP>
    else if not (idknown(a,ids) or idknown(b,ids))
    then <ERROR>
    else
    (
        if <TIME_TP> in set {idtype(a,ids)} union {idtype(b,ids)}
        then <TIME_TP>
        else <REAL_TP>
    )
others ->
    -- simple expressions
    if is_real(e) then <REAL_TP>
    else if is_bool(e) then <BOOL_TP>
    else if is_Id(e) then
        if idknown(e,ids) then idtype(e,ids)
        else <ERROR>
    else if is_Time(e) then <TIME_TP>
    else if e = <Time> then <TIME_TP>
    else <ERROR>
end;

-- return true if an id is known, false otherwise
idknown: Id * CoModelIds -> bool
idknown(id, ids) ==
cases id.model:
    nil -> id in set dom ids.contract,
    <DE> -> id in set dom ids.de,
    <CT> -> id in set dom ids.ct

```

```

    end;

    -- return the type of an id
    idtype: Id * CoModelIds -> Type
    idtype(id, ids) ==
        cases id.model:
            nil    -> ids.contract(id),
            <DE> -> ids.de(id),
            <DE> -> ids.ct(id)
        end
    pre idknown(id, ids);

    -- return the set of all ids that are assigned to in a statement
    assigns: Stmt -> set of Id
    assigns(s) ==
        cases s:
            mk_AssignStmt(id,-) -> {id},
            stmt ^ stmts        -> assigns(stmt) union assigns(stmts),
            stmt                 -> assigns(stmt),
            []                   -> {},
            others               -> {}
        end;

    -- return the set of all ids that are reverted in a statement
    reverts: Stmt -> set of Id
    reverts(s) ==
        cases s:
            mk_RevertStmt(id) -> {id},
            stmt ^ stmts      -> reverts(stmt) union reverts(stmts),
            stmt               -> reverts(stmt),
            []                 -> {},
            others             -> {}
        end
    end DCL

```

A.3 Concrete Syntax

This section gives a concrete syntax for DCL in a BNF dialect, using the following special symbols .

,	the concatenate symbol
=	the define symbol
	the definition separator symbol (alternatives)
[]	enclose optional syntactic items
{ }	enclose syntactic items which may occur zero or more times
' '	single quotes are used to enclose terminal symbols
meta identifier	non-terminal symbols are written in lower-case letters (possibly including spaces)
;	terminator symbol to represent the end of a rule.
()	used for grouping, e.g. "a, (b c)" is equivalent to "a, b a, c".
-	denotes subtraction from a set of terminal symbols (e.g. "character - ('")" denotes all characters excepting the double quote character.)

A.3.1 Script

script = top-level statement, [{ ';' , top-level statement }], [';'] ;

A.3.2 Top-Level Statements

top-level statement = include statement
| when statement ;

include statement = **'include'**, '"', file path, '"' ;

when statement = **'when'**, expression, **'do'**, statement, [**'after'**, statement] ;

A.3.3 Statements

statement = assign statement
| block statement
| revert statement
| print statement
| error statement
| **'quit'** ;

block statement = (' (' | '{'), statement, [{ ';' , statement }], [';'], (')' | '}')
| '(', statement, [{ ';' , statement }], [';'], ')' ;

assign statement = identifier, **':='**, expression ;

revert statement = **'revert'**, identifier ;

print statement = **'print'**, formatted string ;

error statement = **'error'**, formatted string ;

A.3.4 Expressions

```

expression = boolean literal
            | numeric literal
            | time literal
            | 'time'
            | identifier
            | unary expression
            | binary expression ;

```

```

boolean literal = 'true' | 'false' ;

```

```

time literal = numeric literal, [ '{', time unit, '}' ] ;

```

```

time unit = ( 'microseconds' | 'us' )
            | ( 'milliseconds' | 'ms' )
            | ( 'seconds' | 's' )
            | ( 'minutes' | 'm' )
            | ( 'hours' | 'h' ) ;

```

```

numeric literal = numeral, [ '.', digit, { digit } ], [ exponent ] ;

```

```

exponent = ( 'E' | 'e' ), [ '+' | '-' ], numeral ;

```

```

numeral = digit, { digit } ;

```

```

identifier = [ domain ], identifier literal ;

```

```

domain = 'de' | 'ct' ;

```

```

unary expression = unary operator, expression ;

```

```

unary operator = '+'
                | '-'
                | 'abs'
                | 'floor'
                | 'ceil' ;

```

```

binary expression = expression, binary operator, expression ;

```

```

binary operator = '+'
                 | '-'
                 | '*'
                 | '/'

```

```

| 'div'
| 'mod'
| '<'
| '<='
| '>'
| '>='
| '='
| '<>'
| 'or'
| 'and'
| '=>'
| '<=>'
| 'for' ;

```

```
formatted string = '"', string, '"', [ '%', identifier, [ {',', identifier}]] ;
```

It is assumed that **digit**, **string**, and **file path** have their obvious definitions. Since no variables can be defined in DCL, all identifiers will exist in the co-model and therefore **identifier literal** will conform to the conventions of VDM and 20-sim.

Bibliography

- Alur, R., C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine (1995), The Algorithmic Analysis of Hybrid Systems, *Theoretical Computer Science B*, **vol. 138**, pp. 3–34.
- van Amerongen, J. (1994), *Regeltechniek*, Open Universiteit, ISBN 90-358-1303-0.
- van Amerongen, J. (2010), *Dynamical Systems for Creative Technology*, Controllab Products B.V., Enschede.
- Ashenden, P. J. (2001), *The Designer's Guide to VHDL*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, ISBN 1558606742.
- Astrom, K. J. and B. Wittenmark (1994), *Adaptive Control*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, ISBN 0201558661.
- Avizienis, A., J.-C. Laprie, B. Randell and C. Landwehr (2004), Basic Concepts and Taxonomy of Dependable and Secure Computing, *IEEE Transactions on Dependable and Secure Computing*, **vol. 1**, pp. 11–33, ISSN 1545-5971.
- Banks, J., J. Carson, B. L. Nelson and D. Nicol (2004), *Discrete-Event System Simulation (4th Edition)*, Prentice Hall, 4 edition, ISBN 0131446797.
- Bazaraa, M., H. Sherali and C. Shetty (2006), *Nonlinear programming: theory and algorithms*, Wiley-Interscience, ISBN 9780471486008.
- ten Berge, M., B. Orlic and J. Broenink (2006), Co-Simulation of Networked Embedded Control Systems, a CSP-like process-oriented approach, in *Proc. IEEE International Symposium on Computer Aided Control Systems Conference*, IEEE, pp. 434–439.
- Bezemer, M. M. (2013), *Cyber-physical systems software development: way of working and tool suite*, Ph.D. thesis, University of Twente, Enschede.
- Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2012), Design and Use of CSP Meta-Model for Embedded Control Software Development, in *Communicating Process Architectures 2012*, volume 69 of *Concurrent System Engineering Series*, Eds. P. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing, pp. 185–199, ISBN 978-0-9565409-5-9.
- <http://wotug.org/papers/CPA-2012/Bezemer12a/Bezemer12a.pdf>
- Bjoerner, D. and C. Jones (1978), *The Vienna Development Method: The Meta-language*, Lecture Notes In Computer Science, New York : Springer-Verlag, ISBN 9783540087663.
- Breedveld, P. C. (2009), Chapter 1: Port-Based Modeling of Dynamic Systems, in *Modeling and Control of Complex Physical Systems - The Port-Hamiltonian Approach*, Springer Verlag, pp. 1–52.
- Brodskiy, Y. (2014), *Robust autonomy for interactive robots*, Ph.D. thesis, University of Twente, Enschede.
- Broenink, J. (1999), Object-oriented modeling with bond graphs and Modelica, in

- Proceedings of the International Conference on Bond Graph Modeling and Simulation*, pp. 163–168, ISBN 1-56555-155-9.
- Broenink, J. F. (1990), *Computer-aided physical-systems modeling and simulation: a bond-graph approach*, Ph.D. thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands.
- Broenink, J. F. (1997), Modelling, Simulation and Analysis with 20-Sim, *Journal A Special Issue CACSD*, vol. 38, pp. 22–25.
- Broenink, J. F., J. Fitzgerald, C. Gamble, C. Ingram, A. Mader, J. Marincic, Y. Ni, K. Pierce and X. Zhang (2012), EU7 Project deliverable 2.3 — Methodological Guidelines 3, Technical report, The DESTECs Project (INFSO-ICT-248134).
- Broenink, J. F., J. Fitzgerald, K. Pierce, Y. Ni and X. Zhang (2010a), EU7 Project deliverable 2.1 — Methodological Guidelines 1, Technical report, The DESTECs Project (INFSO-ICT-248134).
- Broenink, J. F., M. A. Groothuis, P. Visser and M. M. Bezemer (2010b), Model-Driven Robot-Software Design Using Template-Based Target Descriptions, in *ICRA 2010 workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, Eds. D. Kubus, K. Nilsson and R. Johansson, IEEE, IEEE, pp. 73 – 77.
- Broenink, J. F., M. A. Groothuis, P. Visser and B. Orlic (2007), A Model-Driven Approach to Embedded Control System Implementation, in *Proceedings of the 2007 Western Multiconference on Computer Simulation WMC 2007, San Diego*, Eds. J. Anderson and R. Huntsinger, SCS, San Diego, San Diego, pp. 137–144, ISBN 1-56555-311-X.
- Broenink, J. F. and G. Hilderink (2001), A structured approach to embedded control systems implementation, in *Proceedings of the 2001 IEEE International Conference on Control Applications*, Eds. M. W. Spong, D. Repperger and J. M. I. Zannatha, IEEE, pp. 761–766, ISBN 0-7803-6733-2, doi: 10.1109/CCA.2001.973960.
- Broenink, J. F., P. G. Larsen, M. Verhoef, C. Kleijn, D. S. Jovanović, K. G. Pierce and F. Wouters (2010c), Design Support and Tooling for Dependable Embedded Control Systems, in *Proceedings of SERENE '10*, ACM, ACM Sigsoft, pp. 77 – 82.
- Broenink, J. F. and Y. Ni (2012), Model-Driven Robot-Software Design using integrated Models and Co-Simulation, in *International Conference on Embedded Computer Systems, SAMOS 2012, Samos, Greece*, Eds. J. McAllister and S. Bhattacharyya, IEEE Computer Society, USA, pp. 339–344, ISBN-13 978-1-4673-2296-6.
- Broenink, J. F., Y. Ni and M. A. Groothuis (2010d), On Model-driven Design of Robot Software using Co-simulation, in *SIMPAR, Workshop on Simulation Technologies in the Robot Development Process*, Ed. E. Menegatti.
- Carothers, C. D., K. S. Perumalla and R. M. Fujimoto (1999), Efficient Optimistic Parallel Simulations Using Reverse Computation, *Journal: ACM Trans Model Comput Simul*, vol. 9, pp. 224–253.
- Cellier, F. (1979), *Combined continuous discrete systems simulation by use of digital computers: techniques and tools*, Ph.D. thesis, ETH, Zuerich, Switzerland.
- Chen, W., M. Huhn and P. Fritzson (2011), A Generic FMU Interface for Modelica., in

- EOOLT*, volume 56 of *Linköping Electronic Conference Proceedings*, Eds. F. E. Cellier, D. Broman, P. Fritzson and E. A. Lee, Linköping University Electronic Press, pp. 19–24.
- Chiaverini, S., B. Siciliano and L. Villani (1999), A survey of robot interaction control schemes with experimental comparison, *Mechatronics, IEEE/ASME Transactions On*, **vol. 4**, pp. 273–285.
- Chiodo, M., P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli and L. Lavagno (1994), Hardware-Software Codesign of Embedded Systems, *IEEE Micro*, **vol. 14**, pp. 26–36, ISSN 0272-1732.
- Corporaal, H. (2006), Embedded System Design, in *Progress White Papers 2006*, Ed. F. Karelse, STW, Utrecht, pp. 7–27.
- Davis, J., R. Galicia, M. Goel, C. Hylands, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay and Y. Xiong (1999), Ptolemy-II: Heterogeneous Concurrent Modeling and Design in Java, Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley.
- De Lara, J. and H. Vangheluwe (2002), ATOM3: A Tool for Multi-formalism and Meta-modelling, in *Fundamental approaches to software engineering*, Springer Berlin Heidelberg, pp. 174–188.
- DFKI (2011), German Research Center for Artificial Intelligence–Industry 4.0.
http://www.siemens.com/innovation/apps/pof_microsite/_pof-spring-2013/_html_en/industry-40.html
- Dirne, H. (2005), Demonstrator of advanced controllers, Msc report, University of Twente.
- Dürr, E. (1992), Syntactic Description of the VDM++ Language, Technical report, CAP Gemini, P.O. Box 2575, 3500 GN Utrecht, NL.
- Edwards, S., L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli (1997), Design of Embedded Systems: Formal Models, Validation, and Synthesis, in *PROCEEDINGS OF THE IEEE*, pp. 366–390.
- Eisenring, M., E. Zitzler and L. Thiele (1999), CoFrame: A Modular Co-Design Framework for Heterogeneous Distributed Systems, Technical Report 81, Computer Engineering and Networks Laboratory, ETH Zurich.
- Erbas, C. (2006), *System-Level Modeling and Design Space Exploration for Multiprocessor Embedded System-on-Chip Architectures*, Ph.D. thesis, Amsterdam University.
- Fitzgerald, J., P. G. Larsen and M. Verhoef (2008), Vienna Development Method, *Wiley Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, John Wiley & Sons, Inc.
- Fitzgerald, J. S., P. G. Larsen, K. G. Pierce and M. H. G. Verhoef (2013), A formal approach to collaborative modelling and co-simulation for embedded systems, *Mathematical Structures in Computer Science*, **vol. 23**, pp. 726–750.
- Fritzson, P. and V. Engelson (1998), Modelica - A Unified Object-Oriented Language

- for System Modelling and Simulation, in *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 67–90.
- Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995), *Design Patterns. Elements of Reusable Object-Oriented Software.*, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company.
- Gheorghe, L. (2009), *Continuous/Discrete Co-Simulation Interfaces from formalization to implementation*, Ph.D. thesis, University of Montreal.
- Groothuis, M., A. Damstra and J. Broenink (2008), Virtual Prototyping through Co-simulation of a Cartesian Plotter, in *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, IEEE Industrial Electronics Society, 08HT8968C, pp. 697–700.
- Groothuis, M. A. and J. F. Broenink (2010), A multiple-view method for embedded mechatronic software design, Invited lecture at PROGRESS track at STW.ICT conference, 18-19 Nov 2010, Veldhoven, NL.
- Haubelt, C., T. Schlichter and J. Teich (2006), Improving Automatic Design Space Exploration by Integrating Symbolic Techniques into Multi-Objective Evolutionary Algorithms, *International Journal of Computational Intelligence Research*, pp. 239–254.
- Heath, S. (2002), *Embedded Systems Design*, Elsevier Science, ISBN 9780080477565.
- Henzinger, T. A. (1996), The Theory of Hybrid Automata, in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Society, Washington, DC, USA, LICS '96, pp. 278–, ISBN 0-8186-7463-6.
- Hilhorst, R. A. (1992), *Supervisory control of mode-switch processes*, Ph.D. thesis, University of Twente, Enschede, Netherlands.
- Hoare, C. A. R. (1978), Communicating Sequential Processes, *Journal: Commun ACM*, vol. 21, pp. 666–677.
- Holzer, M. (2008), *Design Space Exploration for the Development of Embedded Systems*, Ph.D. thesis, Technischen.
- Hu, J., Z. Fan, J. Wang, S. Li, K. Seo, X. Peng, J. Terpenney, R. Rosenberg and E. Goodman (2008), GPBG: A Framework for Evolutionary Design of Multi-domain Engineering Systems Using Genetic Programming and Bond Graphs, in *Design by Evolution*, Eds. P. F. Hingston, L. C. Barone and Z. Michalewicz, Springer Berlin Heidelberg, Natural Computing Series, pp. 319–345, ISBN 978-3-540-74111-4.
- IEEE (2008a), *International Standard ISO/IEC 12207:2008(E), IEEE Std 12207-2008 (Revision of IEEE/EIA 12207.0-1996) Systems and software engineering — Software life cycle processes*, ISO/IEC and IEEE Computer Society.
- IEEE (2008b), *International Standard ISO/IEC 15288:2008(E), IEEE Std 15288-2008 (Revision of IEEE Std 15288-2004) Systems and software engineering — System life cycle processes*, ISO/IEC and IEEE Computer Society.
- Jones, C. B. (1990), *Systematic Software Development using VDM*, Prentice Hall International, second edition, ISBN 0-13-880733-7.

- Jovanović, D. (2006), *Designing dependable process-oriented software, a CSP approach*, PhD thesis, University of Twente, Enschede, The Netherlands.
- Karnopp, D. and R. Rosenberg (1968), *Analysis and simulation of multiport systems: the bond graph approach to physical system dynamic*, MIT Press, Cambridge, MA, USA.
- Kleijn, C. (2009), *20-sim 4.1 Reference Manual*, Controllab Product B.V., Enschede, ISBN 978-90-79499-05-2-7.
- Kleijn, C. (2011), *Torsion Bar 2.0 Reference Manual*, Getting Started with 20-sim, ISBN 9789079499090.
- Larsen, P. G., N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl and M. Verhoef (2010), The Overture Initiative – Integrating Tools for VDM, *ACM Software Engineering Notes*, vol. 35.
- Larsen, P. G., C. Kleijn, K. G. Lausdahl, A. Ribeiro and Y. Ni (2011), EU7 Project deliverable 3.3c — User Manual, Technical report, The DESTecs Project (INFSO-ICT-248134).
- Lausdahl, K. G., A. Ribeiro, P. Visser, F. Groen, Y. Ni, J. F. Broenink, A. Mader, J. W. Coleman and P. G. Larsen (2011), EU7 Project deliverable 3.3b — Co-simulation Foundations, Technical report, The DESTecs Project (INFSO-ICT-248134).
- Lee, E. A. (2006), Cyber-Physical Systems - Are Computing Foundations Adequate ?, in *NSF Workshop on Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, Austin, Texas, USA.
- Liddell, H. G., R. Scott, G. Crane, H. S. Jones and R. McKenzie (1996), *A Greek-English lexicon*, Oxford University Press, 9th edition.
- Liu, J. (1998), Continuous Time and Mixed-Signal Simulation in Ptolemy II, Technical Report UCB/ERL M98/74, EECS Department, University of California, Berkeley.
- Maimaiti, M. (2011), Toward Development of Overture/VDM++ to Java code generator, master thesis, UPPSALA Universitet.
- Modelisar (2010), Functional Mock-up Interface for Model Exchange 1.0., Technical report, available <http://www.functional-mockup-interface.org>.
- Mosterman, P., J. Zander and Z. Han (2013), The Towers of Hanoi as a Cyber-Physical System Education Case Study, in *Proceedings of the First Workshop on Cyber-Physical Systems Education (CPS-Ed 2013) at Cyber Physical Systems Week*.
- Ni, Y. and J. F. Broenink (2012), Hybrid systems modelling and simulation in DESTecs: a co-simulation approach, in *26th European Simulation and Modelling Conference, ESM 2012, Essen, Germany*, Ed. M. Klumpp, EUROSIS-ETI, Ghent, Belgium, pp. 32–36, ISBN-13 978-90-77381-73-1.
- Ni, Y. and J. F. Broenink (2014), A co-modelling method for solving incompatibilities during co-design of mechatronic devices, *Journal: Advanced Engineering Informatics*, 5 years Thomson Reuters Impact factor: 2.126, vol. 28, pp. 232–240, doi: 10.1016/j.aei.2014.05.004.
- Ni, Y., J. F. Broenink, A. Ribeiro, K. G. Lausdahl, F. Groen, M. Groothuis, K. Pierce,

- C. Gamble and P. G. Larsen (2011), EU7 Project deliverable 3.3a — Design Space Exploration Tool Support, Technical report, The DESTECs Project (INFSO-ICT-248134).
- Ogata, K. (1995), *Discrete-Time Control Systems*, Prentice Hall International, Inc., second edition.
- Orlic, B. (2007), *SystemCSP : a graphical language for designing concurrent component-based embedded control systems*, Ph.D. thesis, University of Twente, Enschede.
- O'Searcoid, M. (2006), *Metric Spaces*, Springer Undergraduate Mathematics Series, Section 9.4, Springer, ISBN 9781846286278.
- Palermo, G., C. Silvano and V. Zaccaria (2005), Multi-objective design space exploration of embedded systems, *J. Embedded Comput.*, **vol. 1**, pp. 305–316, ISSN 1740-4460.
- Parr, T. (2007), *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*, Pragmatic Bookshelf, Raleigh, NC, ISBN 978-0-9787-3925-6.
- Paynter, H. M. (1961), *Analysis and design of engineering systems*, MIT Press, Cambridge, MA.
- Pierce, K., C. Gamble, Y. Ni and J. F. Broenink (2012), Collaborative Modelling and Co-simulation with DESTECs: A Pilot Study, in *WETICE*, pp. 280–285.
- Plat, N. and P. G. Larsen (1992), *An Overview of the ISO VDM-SL Standard*, ACM, New York, NY, USA, volume 27, pp. 76–82, doi:10.1145/142137.142153.
- Ptolemaeus, C. (Ed.) (2014), *System Design, Modeling, and Simulation using Ptolemy II*, Ptolemy.org.
<http://ptolemy.org/books/Systems>
- Qamar, A. (2013), *Model and Dependency Management in Mechatronic Design*, Ph.D. thesis, PhD thesis, KTH-Royal Institute of Technology.
- Robinson, S. (2007), *Simulation: The Practice of Model Development and Use*, John Wiley & Sons, ISBN 0470847727.
- Schutter, B., W. Heemels, J. Lunze and C. Prieur (2009), *Survey of modeling, analysis, and control of hybrid systems*, Cambridge University Press, chapter 2, pp. 33–55, Handbook of Hybrid Systems Control - Theory, Tools, Applications.
- Sontag, E. (1990), *Mathematical control theory: deterministic finite dimensional systems*, Texts in applied mathematics, Springer-Verlag, ISBN 9780387973661.
- Thomas, D. and P. Moorby (2008), *The Verilog Hardware Description Language*, Springer Publishing Company, Incorporated, 5th ed. edition, ISBN 0387849300, 9780387849300.
- Verhaar, C. (2008), An integrated embedded control software design case study using Ptolemy II, Msc thesis, EWI.
- Verhoef, M. (2009), *Modeling and Validating Distributed Embedded Real-Time Control Systems*, Ph.D. thesis, Radboud University Nijmegen.
- Verhoef, M., B. Bos, P. van Eijk, J. Remijnse, E. Visser, M. de Paepe, Y. de Witte,

- K. Rombaut and R. van Lembergen (2012), D4.3 — Industrial Case Studies, Final report (public), The DESTECS Project (CNECT-ICT-248134).
- van de Vijver, B. (2014), A software safety layer for the Mechatronic Demonstrator, Bsc report 003ram2014, University of Twente.
- Visser, E. (2011), Modeling and Controlling the R2-G2P Robot, bachelor thesis, Control Engineering, University of Twente.
- Wiener, N. (1947), *Cybernetics Or Control and Communication in the Animal and the Machine*, M.I.T. paperback series, Wiley, ISBN 9780262730099.
- Wolff, S., K. Pierce and P. Derler (2012), Multi-domain Modelling in DESTECS and Ptolemy – a Tool Comparison, Technical report ece-tr-15, Aarhus University, Department of Engineering.
- Zhang, F., M. Yeddapanudi and P. Mosterman (2008), Zero-Crossing Location and Detection Algorithms for Hybrid System Simulation, in *Proceedings of the 17th IFAC World Congress*, volume 17.
- Zitzler, E. and L. Thiele (1998), An Evolutionary Approach for Multiobjective Optimization: The Strength Pareto Approach, TIK Report 43, Computer Engineering and Networks Laboratory (TIK), ETH Zurich.

Index

- Actuators, 10
- Analogue, 11
- Analysis Results, 17
- Automated Co-model Analysis, 17, 58
- Automated Co-model Execution, 17
- Bond Graphs, 14
 - Bonds, 14
 - Computational Causality, 15
 - Edges, 14
 - Effort, 14
 - Flow, 14
 - Physical Concepts, 14
 - Power Ports, 14
 - Vertices, 14
- Bumpless transfer, 40
- Co-Model, 12, 43
- Co-Modelling, 12
- Co-Simulation, 13
- Co-Simulation Engine, 14
- Collaborative Modelling, 12
- Computational Domain, 11
- Continuous Time, 11
- Contract, 12
- Control Delay, 42
- Cyber-Physical Systems, 1
- Design Alternatives, 16
- Design Space Exploration, 16, 57
- Difference Equation, 11
- Differential Equation, 11
- Digital, 11
- Discrete Time, 11
- Discrete-event Simulation, 11
- Discrete-Time Controllers, 42
- Dynamic Systems, 11
- Effort Modulated Source, 37
- Embedded Control System, 9
- Error, 17
- Event, 11, 12
- Failure, 17
- Fault, 17
- Fault Behaviour, 18
- Fault Forecasting, 17
- Fault Injection, 18
- Fault Modelling, 18
- Fault Prevention, 17
- Fault Removal, 17
- Fault Tolerance, 17
- Fault Tolerant, 17
- Hybrid System Theory, 9
- Hybrid Systems, 9
- Ideal Physical Models, 15, 38
- Intermittent Errors, 17
- Measurement output, 37
- Model-Switch Control, 40
- Modelling & Simulation
 - Validation, 9
 - Verification, 9
- Modelling & Simulation
 - Competent Model, 9
 - Design Parameter, 10
 - Executable Model, 9
 - Model, 9
 - Modelling, 9
 - Simulation, 9
 - Simulation result, 10
 - Simulation Verification, 9
 - Stimuli, 10
 - Variable, 10
 - Disturbance, 10
 - Environment, 10
- Non-Normative Behaviour, 17
- Normal Behaviour, 18
- Optimal, 59
- Persistent Errors, 17
- Ranking Function, 17

Sampling Time, 42
Scenario, 13
Script, 13
Sensors, 10
Simulation Result, 14
Steering Input, 37

Transient Errors, 17

Validate, 37
VDM
 Interface, 15
 Abstract Class, 15
 Aggregate, 15

 Functions, 15
 Mappings, 15
 Object Oriented, 15
 Operations, 15
 Periodic, 15
 Records, 15
 Sequences, 15
 Sets, 15
 Sync, 15
 Threads, 15
 Types, 15
 Values, 15
Vienna Development Method, 15

Acknowledgement

It was quite a story of my PhD journey. Thanks for every single one who I met during this journey, in some aspects, all of you helped me to grow from your own roles, no matter it was from a positive or a negative perspective.

Thanks to Jan Broenink for giving me the opportunity to do a PhD in a topic that I was interested in. Thanks Stefano Stramigioli, for being my promotor.

Thanks to my colleagues in DESTecs project, Augusto, Kenneth Lauthsel, Carl, Peter Visser, Ken Pierce. It was my honour to work with people like you guys. I enjoyed the process and learned a lot from you guys.

Many thanks to my friends: Cassia, Simone and Yun. Without you girls, it would be impossible for me to make it until the end of my PhD. I still remembered that years ago, one cold and rainy weekend in Amsterdam was the start of everything (of doing a PhD in the Netherlands). At that moment, we could not name any Dutch university immediately :P Through all these years, you girls have always been there (despite the geolocation/time differences etc.) for me whenever I need your support/consultancy. That was/is extremely important for me, although all of you are miles away. But our friendship still last, and it will still be an important part of my life. The distance did not tear us apart, it just made our friendship stronger!

Thanks to Carla, Angelika for helping me out when I had this emergency with my landlord. Carla, thanks for finding me a new place on campus so fast. Angelika, thanks for giving me the key of your house to offer me a place to stay (although in the end, I didn't need it). That meant a lot to me. Especially it was the first month after I moved to this country.

Thanks, Gil. You made my life in Matenweg more rememberable. I remembered a lot of interesting discussions (almost cover the entire universe :P) that we had in the kitchen. Wei, Xueling, Yanting, Chanjuan, Husan, thanks for occasional Chinese gathering and chatting. They are all so enjoyable.

Thanks to my officemates: Geert, Eamon, Tadele, Abeje, Douwe, Yury, Marcel Groothuis. You guys made the life at the office more cozy. The conversations are always very inspiring. Of course, also thanks to other RAM colleagues who are not in the same office room: Guus, Roy, Dian, Stefan, Israel, Momen, Ditske, Maarten, Marcel Schwitz, Jolanda, Robert, Jos. It was always nice to meet randomly in the social corner and start up with some random conversations. Robert, the trip with you to Lyon was fun. Jos, thanks for all the helps (including the one hour moving: the fastest house moving ever. :P).

I have to thank my manager and colleagues in Nedap. Thanks for understanding the situation in the last phase of my thesis and supporting. I enjoy working with you guys.

Thanks, 绿豆 . Thank you for making me happy and of course for all the patience and support. Without your food, possibly I wouldn't be alive to make until the finish of the

thesis. With you, it seems that even the most difficult time seems to be easier. Thank you! My grateful thanks also extend to the Visser family. Dankjewel!

Year 2013, it was extremely hard for my family... My dearest uncle, I am sorry that I was not there when everything happened. But maybe that was for the best. In my memory, you will always be this uncle who spoiled me the most when I was a little kid. You had the most patience to all of my ridiculous demands. I had so much fun with you! I hope that you are happy at where you are now! Rest in peace...

Grandpa, I am so sorry that you could not see all these. It was initially a plan only for 2 years studying abroad for a master degree. Now it has been 8 years already. Sometimes I feel so sorry that I even forgot that you are no longer with us (I dreamt about you like you were still here)... I just assumed that you will be there when I return... I really wish that I could have been in the hospital just one day earlier to talk with you for the last time. I still miss you, very much... For me, all those images about you will not fade along the time.

A very big thanks to my parents. Thank you for supporting me unconditionally along the way and encouraging me to explore my own life. I was not there when you guys needed me, but you were/are always there for me. It is very lucky of me to have open-minded parents like you guys. I am grateful for everything. You guys always think about me first. Your love for me are always there without any condition. I always know that even if the whole world fell that I have a place called home to go. 老爸老妈, 谢谢!

What a journey! Without any of you's help, constant encouragement and wise suggestions, I can't make it. Thank you very much!

Now this chapter of the life is closed. I am looking forward for the next chapter and many chapters after that.

About the author

Yunyun Ni obtained her BSc degree in Electrical Engineering at East China University of Science and Technology in 2004. After her graduation, she worked in Hardinge Inc. as an electrical engineer shortly in China.

In 2009, she completed her master degree in Computational Engineering from Friedrich-Alexander-Universität Erlangen-Nürnberg with honors. From 2010, she joined the Robotic and Mechatronics group at University of Twente as a PhD candidate to work in DESTECS (Design Support and Tooling for Embedded Control Software) project. The results of her PhD research have been presented in this thesis.

PROPOSITIONS

belonging to the thesis titled

SYSTEM DESIGN SUPPORT OF CYBER-PHYSICAL SYSTEMS A CO-SIMULATION AND CO-MODELLING APPROACH

to be defended in public on Thursday June 18, 2015 at 12:45 by

Yunyun Ni

1. While we often think that we understand each other in the same terms, in truth this is rarely the case. (Chapter 2)
2. It is all about time. However time is a relative physical quantity. It is the fourth dimension in which we cannot travel backwards. The co-simulation scheme proposed in this thesis, resembles the physical nature of time.
3. Seeing the other side of the world is already difficult; it is no wonder we have such a hard time seeing the whole world at once. (Chapter 4)
4. The more something tolerates failure, the stronger it grows. This rule applies both to software and human being.(Chapter 5)
5. Knowing when to stop is a decision that few can make correctly. Maybe this is the reason that human need computers to help with making right decisions.
6. Slow is smooth, smooth is fast. It is more important to pass the right finish line.
7. Students cannot learn lessons if the teacher is too merciful.
8. Numbers are everywhere. We might just be simulations in another civilization's lab.
9. We have two ears and one mouth, yet somehow we still do not listen enough.

ISBN 978-90-365-3855-8

```
01010011 01110101 01100011 01100011 01100101 01110011
01110011 00100000 01100100 01101111 01100101 01110011
00100000 01101110 01101111 01110100 00100000 01100001
01101100 01110111 01100001 01111001 01110011 00100000
01100011 01101111 01101101 01100101 00100000 01100110
01110010 01101111 01101101 00100000 01100001 00100000
01110011 01110100 01110010 01101111 01101110 01100111
00100000 01101100 01100101 01100001 01100100 01100101
01110010 00101110 _
```