

John Fitzgerald · Peter Gorm Larsen  
Marcel Verhoef *Editors*

---

# Collaborative Design for Embedded Systems

Co-modelling and Co-simulation

 Springer

# Collaborative Design for Embedded Systems



John Fitzgerald • Peter Gorm Larsen •  
Marcel Verhoef  
Editors

# Collaborative Design for Embedded Systems

Co-modelling and Co-simulation

 Springer

*Editors*

John Fitzgerald  
Newcastle University  
Newcastle upon Tyne  
United Kingdom

Peter Gorm Larsen  
Aarhus University  
Aarhus  
Denmark

Marcel Verhoef  
Chess WISE B.V.  
Haarlem  
The Netherlands

ISBN 978-3-642-54117-9

ISBN 978-3-642-54118-6 (eBook)

DOI 10.1007/978-3-642-54118-6

Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014936992

© Springer-Verlag Berlin Heidelberg 2014

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

Springer is part of Springer Science+Business Media ([www.springer.com](http://www.springer.com))

# Foreword

Embedded systems permeate the world today, and they increase not only in numbers but also in complexity. One by now archetypical example, also quoted in this book, is the modern automobile, with its dozens of electronic control units (ECUs). The behaviour and functionality of the embedded control of a car is not the result of a single piece of software or silicon, but rather *emerges* from a very complex interplay of ECUs. These components communicate with each other in complex, hard-to-predict ways. Protocol mismatches are a common source of error, and agreeing on common interfaces among components is one of the most difficult and time-consuming tasks in managing large designs.

This very problem, of finding a common ground across collaboration interfaces, also applies to the designers themselves who create these complex embedded systems. Embedded, or *cyber-physical*, systems bring a multitude of disciplines together, including, but not limited to, computer science, electrical engineering, physics and psychology. Within each discipline, there is again a range of different fields, with experts that are mostly used to work among peers of their own domain. Vocabularies are different or, even worse, are the same but have different semantics.

In the realm of research projects, few attempt to bring together partners from very diverse fields, or if they do, they limit interaction to high-level, narrow interfaces. The European “Design Support and Tooling for Embedded Control Software” (DESTTECS) project, which we had the pleasure to accompany as reviewers in 2010–2012, did have the ambitious goal to work closely together across domains. DESTTECS combined discrete-event and continuous-time modelling, and this cross-cutting aspect would be a key part of the success or failure of the project. This did pose a non-trivial project risk, despite the fact that the participants were all already highly acknowledged experts in their domains and brought together much professional experience from earlier projects. Quoting from one of the project reports: “The project is quite multidisciplinary and as a consequence it turned out to be beneficial to produce exact definitions of terminology, as in the first half year of the project, understanding of relevant terms was quite different.” This extra effort, not accounted for in the original resource planning, did pay off very well, as we could witness in subsequent project periods.

One outcome of DESTTECS that lasts beyond the project is the *Crescendo approach*, which demonstrated its viability in several full-scale demonstrators developed jointly by academic and industrial partners. Another outcome that we expect to have lasting impact is this book, which makes Crescendo and its underlying concepts accessible in a way that transgresses any collection of separate papers and which has added significant value to the project even after it has formally ended. This book not only straddles the aforementioned boundaries between (mostly) separate fields, but also finds a perfect balance between theoretical underpinnings and practical advice. It should be equally valuable for researchers and practitioners in the field of embedded systems design.

Elancourt, France  
Kiel, Germany

Bernard Dion  
Reinhard von Hanxleden

# Preface

The embedded systems market is a lively—some would say volatile—place. There is a growing demand for products that make the best use of rapidly improving computer hardware to create everything from game consoles to flight controllers. In this setting, the developers of embedded systems have to form creative teams out of disparate engineering disciplines. For example, a product design team might encompass software, mechanical, electrical and control engineers. However, effective collaborative design is not simply a matter of sharing a whiteboard with each other—the bases of engineering disciplines are different, and perhaps the biggest gulf is between software and control engineering. Control engineers describe how phenomena evolve and flow over continuous time, but software is described using logic to relate discrete events.

The semantic gaps between engineering disciplines cost time and money because the results of misunderstandings are often only detected when the physical product is built and software fails to control it properly. Traditional product development involves specialist engineering groups working independently on aspects of the design which is passed between them, sometimes being misinterpreted and distorted as it goes. This is a particularly pressing problem when we attempt to design for resilience: dependability cannot be sprinkled over a completed design, but needs to be integrated from the outset. Indeed, our experience suggests that around 80 % of embedded software is related to complex supervisory control which includes switching between modes or dealing with error detection and recovery. It is essential, then, to have methods and tools to help manage the risk of early-stage design flaws.

This book is a response to the challenge of delivering effective multi-disciplinary design. It builds on the premise that early analysis of design models could lead to early detection of errors and performance bottlenecks, and the models themselves can serve as common bases for communication and the exploration of design alternatives. But how can model-based methods work if the engineering disciplines describe aspects of the product and its environment in such different ways? We focus on the creation and analysis of collaborative models (“co-models”), composed of discrete-event and continuous-time models. Typically, these contain discrete-event



models of control elements to be realised on computers, coupled with continuous-time models of controlled plants and the physical environment. However, rather than forcing diverse disciplines into a single common modelling framework, we show how it is possible to link otherwise separate tools for discrete-event and continuous-time modelling through a harness that allows them to share data during simulations to create a unified “co-simulation”. Co-simulation between tools supports collaboration between designers, allowing engineers from each discipline to continue work within familiar formalisms while being able readily to judge the effects of a design decision in one domain on the behaviour of the other. For example, we might choose to resolve a known hardware fault either by using different sensors or by more complex control software. With co-simulation, we can trade off these alternatives on such factors as performance, energy consumption and cost before a commitment is made.

The methods and tools for co-modelling described in this book were developed in a collaborative research project, “Design Support and Tooling for Embedded Control Software” (DESTECS).<sup>1</sup> We were fortunate in DESTECS to have the cooperation of several pioneering companies, who endured the instabilities of our nascent methods and tools, applying them in several domains. Their experience demonstrated the value of co-modelling in reducing design iterations, easing the development of sophisticated software control and supporting dependability. We hope that this book gives the reader a sense of the potential for innovation enabled by methods and tools that support technically well-founded collaboration across discipline boundaries.

## Structure of the Text

Our goal is to present methods and tools for co-modelling, co-simulation and design space exploration in a thoroughly practical way, with running examples. The book is structured in three parts:

1. Part **I** introduces the technical basis of co-modelling and co-simulation using one Continuous-Time (CT) and one Discrete-Event (DE) formalism. Chapter **1** describes the need for collaborative design and the challenges in providing methods and tools to support it. We then introduce core concepts that underpin the rest of the book (Chap. **2**). In Chaps. **3** and **4**, we present the specific CT and DE technologies that we propose to link through co-modelling: respectively, bond graphs, supported by 20-sim, and the Vienna Development Method (VDM). Both are comprehensive formalisms, so in these chapters, we aim to give the reader a sense of the main features of each. We then introduce the Crescendo technology for co-simulation (Chap. **5**). Finally, Chap. **6** discusses the use of structuring mechanisms to promote the reuse of controller models.

---

<sup>1</sup>European Union Framework 7 project CNECT-ICT-248134, January 2010–December 2012 (see <http://www.destecs.org/>).

2. In Part II, we move from foundations to the application of co-modelling. Chapter 7 introduces two case studies (a line-following robot and a ChessWay personal transporter), which are used in this part of the book. The process by which a co-model is developed is discussed in Chap. 8. Chapter 9 introduces techniques for co-modelling faults and fault tolerance mechanisms, while Chap. 10 examines the support for exploring large design spaces in the search for optimal solutions. Chapter 11 brings these strands together, describing how the technology has been applied on other industrial applications.
3. Part III considers more advanced topics. Chapter 12 reports the experiences of three industry users following their experimental deployment of the Crescendo technology. Chapter 13 gives a technical discussion of the semantics of the co-simulation framework underlying the Crescendo technology and explains how it can be re-used with other CT and DE technologies. Finally, Chap. 14 positions our work in the broader setting of model-based collaborative design and sets out the challenges posed by the development of cyber-physical systems.
4. The appendices include summaries of VDM and 20-sim, a catalogue of design patterns for co-models, and an abstract VDM model of the ChessWay with a focus on its safety aspects. A list of acronyms and a glossary of the main terms used in this book are included.

## Using the Book

The book is aimed at both researchers and practitioners in embedded systems development. Among researchers, the book should be of interest to those working in cyber-physical systems, embedded systems design and formal methods; in control engineering, the material should be of interest to those working on advanced control and modelling technology, especially for fault-tolerant and resilient systems. Among practitioners, we target the book at those in research and product development with an interest in improved design processes and tools. Among academics, we expect the text to be of value for those teaching embedded software development at all levels. We recommend that all readers make use of the Crescendo tool for hands-on experience and access the additional content, including tutorials and training material, on the accompanying web site (see below).

In keeping with the spirit of co-modelling, we assume that the reader has some experience in either software development or control, but we do not assume both. In Part I of the book, the introductions to CT modelling in Chap. 3 and DE (computing) modelling in Chap. 4 are written with readers from both backgrounds in mind.

Practitioners with an interest in the techniques of co-modelling and co-simulation are invited to approach the contents of Parts I and II in the order presented. Most of the chapters in the technical flow of the book assume that preceding chapters will have been covered. The advanced topics in Part III are relatively independent of one another. Among the advanced topics, Chap. 13 will be of technical interest mainly

to those working on the formal semantics of modelling languages and so may be omitted by others on a first reading.

Readers with a primary interest in engineering management may wish to cover the motivation and technical foundations in Chaps. 1 and 2, example case studies in Chap. 7 and elements of co-model creation methodology in Chap. 8, followed by industry applications and deployment experience in Chaps. 11 and 12 and future directions for the technology in Chap. 14.

## Accompanying Web Site

The accompanying web site, [www.crescendotool.org](http://www.crescendotool.org), provides additional material, including tool support for co-simulation, as described in the book, additional example co-models that can be used with the tool and course material. We invite readers wishing to use the material for teaching or research to take the distribution only from this web site and contact the editors for further support.

## Acknowledgments

DESTTECS was supported by the European Commission under the Seventh Framework programme. We are grateful to the expert reviewers, Mr. Bernard Dion (Esterel Technologies) and Prof. Reinhard von Hanxleden (Kiel University), for their constructive suggestions and recommendations throughout the project. It is a pleasure to thank the many contributors to the book, particularly Kenneth Pierce, Carl Gamble, Jan Broenink, Job van Amerongen, Christian Kleijn, Augusto Ribeiro, Kenneth Lausdahl, Bert Bos, Sune Wolff and Joey Coleman. We are very grateful to Koenraad Rombaut and Peter van Eijk, who kindly agreed to be interviewed about their industrial application of the technology that we developed, allowing us to recount their experiences in Chap. 12. We gladly acknowledge our other colleagues from DESTTECS who contributed to the technology: Claire Ingram, Kim Bjerger, José Antonio Esparza Isasa, Claus Ballegard Nielsen, Xiaochen Zhang, Yunyun Ni, Angelika Mader, Jelena Marinčić, Stefan Groothuis, Peter Visser, Frank Groen, Marcel Groothuis, Dusko Jovanovic, Jan Remijnse, Eelke Visser, Michiel De Paepe, Yoni De Witte, Roeland Van Lembergen, Wouter Vleugels, Kim Visser and Jeffrey Simons. We are grateful to Nick Battle, Stefan Hallerstedde, Hiroshi Sako and all those who provided feedback to us on draft material. We would also like to thank Martin Peter Christiansen for providing the T1X tractor example and simulation results. At a personal level, we are deeply grateful to our families and friends for their patience and support since the genesis of this book, especially John Hudson, Margit Sandvang Larsen and Natalie Ree.

Newcastle upon Tyne, UK  
Aarhus, Denmark  
Haarlem, The Netherlands

John Fitzgerald  
Peter Gorm Larsen  
Marcel Verhoef

# List of Acronyms

ACA	Automated co-model analysis
AD	Analog to digital
BDD	Block definition diagram
BLDC	Brushless direct current
CoDES	Collaborative design for embedded systems
CPS	Cyber-physical system
CPU	Central processing unit
CoDES	Collaborative design for embedded systems
CRC	Cyclic-redundancy check
CSL	CoDES scripting language
CT	Continuous-time
DA	Digital to analog
DAL	Development assurance level
DATE	Design automation and test in Europe
DE	Discrete event
DEST ECS	Design support and tooling for embedded control software
DSE	Design space exploration
DT	Discrete time
EDA	Electronic design automation
EMF	Electro motive force
FCFS	First come first served
FDIV	Floating-point DIVision
FMEA	Failure mode and effects analysis
FMI	Functional mockup interface
FMU	Functional mockup unit
ForTIA	Formal Techniques Industry Association
FP	Fixed priority

FPGA	Field-programmable gate array
FSM	Finite state machine
GUI	Graphical user interface
HAZOP	Hazard and operability
HiL	Hardware-in-the-loop
IBD	Internal block diagram
IDE	Integrated design environment
IEEE	Institute of electrical and electronics engineers
IFG	Industry follow group
INCOSE	International Council on Systems Engineering
iOS	Internet operating system
IoT	Internet of things
IPM	Ideal physical model
ISO	International Standards Organisation
MIL	Model in the loop
NFC	Near field communication
NTP	Network time protocol
OMG	Object Management Group
PC	Personal computer
PID	Proportional integral derivative
PWM	Pulse width modulated
RC	Resistor-capacitor
RPC	Remote procedure call
RUP	Rational unified process
SDP	Shared design parameter
SHARD	Software hazard analysis and resolution in design
SI	Système Internationale d'unité
SiL	Software-in-the loop
SIL	Safety integrity level
SoS	System of systems
SOS	Structural operational semantics
SRI	French abbreviation for Inertial Reference System
SSS	Single-state simulation semantics
SUT	System under test
SysML	Systems Modelling Language
TSS	Transactional simulation semantics
TTM	Time to market
UML	Unified modeling language
VDM	Vienna development method
VDM-SL	Vienna Development Method Specification Language
VDM-RT	Vienna development method real time

WAM	Weighted additive method
XBMC	Xbox Media Center
XML	Extensible Markup Language
XTE	Cross track error



# Contents

## Part I Co-modelling and Co-simulation: The Technical Basis

<b>1 Collaborative Development of Embedded Systems</b> .....	3
Marcel Verhoef, Kenneth Pierce, Carl Gamble, and Jan Broenink	
1.1 Introduction .....	3
1.2 Setting the Scene .....	4
1.3 The Embedded Systems Design Challenge .....	8
1.4 Embedded Systems Design: An Illustrative Story .....	10
1.5 A Solution: The Crescendo Approach .....	13
1.6 Conclusion .....	14
<b>2 Co-modelling and Co-simulation in Embedded Systems Design</b> .....	15
John Fitzgerald and Kenneth Pierce	
2.1 Introduction .....	15
2.2 Systems and System Boundaries .....	15
2.3 Models .....	16
2.4 Co-models .....	17
2.5 Co-simulation .....	19
2.6 DSE and Automated Co-model Analysis .....	21
2.7 Co-simulation in Practice .....	22
2.8 Conclusion .....	25
<b>3 Continuous-Time Modelling in 20-sim</b> .....	27
Job van Amerongen, Christian Kleijn, and Carl Gamble	
3.1 Introduction .....	27
3.2 Physical Systems .....	29
3.3 Icons and Iconic Diagrams .....	34
3.4 A Domain-Independent Description: Bond Graphs .....	36
3.5 Simulating Physical Systems with 20-sim .....	46
3.6 Control Systems .....	49



3.7 A Small Note on Notation ..... 58

3.8 Conclusion ..... 59

**4 Discrete-Event Modelling in VDM** ..... 61  
Peter Gorm Larsen, John Fitzgerald, Marcel Verhoef,  
and Kenneth Pierce

4.1 Introduction ..... 61

4.2 Basic Elements: Data and Functionality ..... 63

4.3 Example: A Basic Controller Model ..... 72

4.4 Modelling with Structured Data ..... 73

4.5 Example: Supervisory Control ..... 81

4.6 Example: Controlling for Safety ..... 84

4.7 Object-Oriented Structuring ..... 87

4.8 Concurrency ..... 91

4.9 Modelling Systems ..... 93

4.10 Conclusion ..... 95

**5 Support for Co-modelling and Co-simulation: The  
Crescendo Tool** ..... 97  
Peter Gorm Larsen, Carl Gamble, Kenneth Pierce,  
Augusto Ribeiro, and Kenneth Lausdahl

5.1 Introduction ..... 97

5.2 Importing the Torsion Bar Co-model ..... 98

5.3 Crescendo Contracts ..... 98

5.4 Starting a Co-simulation ..... 104

5.5 Using Scripts and SDPs ..... 108

5.6 Changing the Torsion Bar Model ..... 109

5.7 Conclusion ..... 114

**6 Co-model Structuring and Design Patterns** ..... 115  
Kenneth Pierce, Peter Gorm Larsen, and John Fitzgerald

6.1 Introduction ..... 115

6.2 Object-Orientation and Inheritance ..... 117

6.3 Interfaces for Sensors and Actuators ..... 119

6.4 Design Patterns ..... 121

6.5 Using Inheritance for Threads ..... 126

6.6 Structuring Constituent Models for Flexible Simulation ..... 131

6.7 Conclusion ..... 137

**Part II Methods and Applications: The Pragmatics of  
Co-modelling and Co-simulation**

**7 Case Studies in Co-modelling and Co-simulation** ..... 141  
Marcel Verhoef, Bert Bos, Kenneth Pierce, Carl Gamble,  
and Job van Amerongen

7.1 Introduction ..... 141

7.2 The R2-G2P Line-Following Robot ..... 142

7.3	The ChessWay Self-balancing Scooter .....	145
7.4	Conclusion .....	151
<b>8</b>	<b>Methods for Creating Co-models of Embedded Systems</b> .....	<b>153</b>
	Kenneth Pierce, Sune Wolff, and Marcel Verhoef	
8.1	Introduction .....	153
8.2	Paths to Co-models .....	154
8.3	Using SysML Initially .....	157
8.4	The CT-first Approach .....	164
8.5	The DE-first Approach .....	171
8.6	The Contract-first Approach .....	182
8.7	Conclusion .....	183
<b>9</b>	<b>Co-modelling of Faults and Fault Tolerance Mechanisms</b> .....	<b>185</b>
	Carl Gamble, Kenneth Pierce, John Fitzgerald, and Bert Bos	
9.1	Introduction .....	185
9.2	Fault Identification .....	186
9.3	Fault Selection .....	188
9.4	Fault Modelling .....	188
9.5	Fault Tolerance Coverage .....	190
9.6	Fault Tolerance Modelling .....	190
9.7	An Example Using the Line-Following Robot .....	190
9.8	An Example Using the ChessWay .....	194
9.9	Conclusion .....	197
<b>10</b>	<b>Design Space Exploration for Embedded Systems Using</b>	
	<b>Co-simulation</b> .....	<b>199</b>
	Carl Gamble and Kenneth Pierce	
10.1	Introduction .....	199
10.2	Using ACA .....	200
10.3	An Example Using the Line-Following Robot .....	202
10.4	Candidate Parameters .....	205
10.5	Experimental Design .....	207
10.6	Using Folder Launch Configuration .....	210
10.7	An Example Using the T1X Tractor .....	211
10.8	Ranking of Results .....	216
10.9	An Example Using the Line-Measuring Robot .....	218
10.10	Conclusion .....	222
<b>11</b>	<b>Industrial Application of Co-modelling</b>	
	<b>and Co-simulation Technology</b> .....	<b>223</b>
	Marcel Verhoef and Peter Gorm Larsen	
11.1	Introduction .....	223
11.2	A Dredging Excavator .....	225
11.3	A Document Handling System .....	237
11.4	The ChessWay Self-balancing Scooter .....	253
11.5	Conclusion .....	258

**Part III Advanced Topics**

**12 Deploying Co-modelling in Commercial Practice** ..... 263  
 Sune Wolff, Peter Gorm Larsen, and Marcel Verhoef

12.1 Introduction ..... 263

12.2 Company Introductions ..... 264

12.3 Traditional Development ..... 264

12.4 Integrating Co-modelling and Co-simulation  
 with Existing Processes ..... 265

12.5 Resources ..... 266

12.6 Challenges Encountered ..... 266

12.7 Key Benefits ..... 267

12.8 The Future of Co-modelling ..... 269

12.9 Conclusion ..... 270

**13 Semantics of Co-simulation** ..... 273  
 Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen

13.1 Introduction ..... 273

13.2 Structure of Co-simulation ..... 274

13.3 Co-simulation Semantics ..... 278

13.4 Adding Fault Injection Semantics to the Co-simulation ..... 283

13.5 Semantics of the CSL ..... 285

13.6 Conclusion ..... 291

**14 From Embedded to Cyber-Physical Systems:  
 Challenges and Future Directions** ..... 293  
 John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef

14.1 Introduction ..... 293

14.2 The Co-modelling and Co-simulation Landscape ..... 295

14.3 Co-modelling in the CPS Design Flow ..... 297

14.4 Enabling Collections of DE and CT Models to Be Combined ... 298

14.5 Open Co-simulation ..... 298

14.6 Ubiquitous and Distributed Computing ..... 299

14.7 An Open and Lively Research Field ..... 301

14.8 Conclusion ..... 302

**A 20-sim Summary** ..... 305  
 Christian Kleijn

A.1 Introduction ..... 305

A.2 Overview ..... 306

A.3 Graphical Models ..... 307

A.4 Equation Models ..... 307

A.5 Modelling Tools ..... 307

A.6 Simulation ..... 308

A.7 Analysis ..... 309

A.8 Scripting ..... 310

- A.9 Co-simulation ..... 310
- A.10 Code Generation ..... 310
- B VDM-RT Language Summary ..... 313**
  - Peter Gorm Larsen
  - B.1 Operators for Basic Types ..... 313
  - B.2 Operators for Set Types ..... 313
  - B.3 Operators for Sequence Types ..... 313
  - B.4 Operators for Mapping Types ..... 315
  - B.5 Record Types and Values in VDM ..... 315
  - B.6 Small VDM-RT Examples ..... 316
  - B.7 Threads and Synchronisation in VDM ..... 319
  - B.8 The System Class Concept in VDM-RT ..... 319
  - B.9 Example of Classes ..... 320
  - B.10 UML Diagrams ..... 321
- C Design Patterns for Use in Co-modelling ..... 323**
  - Carl Gamble, Kenneth Pierce, John Fitzgerald, Bert Bos,  
and Marcel Verhoef
  - C.1 Introduction ..... 323
  - C.2 Controller Patterns ..... 324
  - C.3 Fault Patterns ..... 341
  - C.4 Fault Tolerance Patterns ..... 347
- D Abstract Modelling of ChessWay Safety ..... 359**
  - Marcel Verhoef and Bert Bos
- References ..... 371**
- Glossary ..... 377**
- Index ..... 381**



# List of Contributors

**Bert Bos**, Chess iX, Haarlem, The Netherlands

**Jan Broenink**, University of Twente, Enschede, The Netherlands

**Joey Coleman**, Aarhus University, Aarhus, Denmark

**John Fitzgerald**, Newcastle University, Newcastle upon Tyne, UK

**Carl Gamble**, Newcastle University, Newcastle upon Tyne, UK

**Christian Kleijn**, Controllab Products, Enschede, The Netherlands

**Peter Gorm Larsen**, Aarhus University, Aarhus, Denmark

**Kenneth Lausdahl**, Aarhus University, Aarhus, Denmark

**Kenneth Pierce**, Newcastle University, Newcastle upon Tyne, UK

**Augusto Ribeiro**, d60, Aabyhoej, Denmark

**Job van Amerongen**, University of Twente, Enschede, The Netherlands

**Marcel Verhoef**, Chess WISE, Haarlem, The Netherlands

**Sune Wolff**, Aarhus University, Aarhus, Denmark

**Part I**  
**Co-modelling and Co-simulation:**  
**The Technical Basis**

# Chapter 1

## Collaborative Development of Embedded Systems

Marcel Verhoef, Kenneth Pierce, Carl Gamble, and Jan Broenink

### 1.1 Introduction

This chapter motivates our interest in collaborative modelling for embedded systems design by describing the challenges faced by developers of contemporary embedded systems. We set the scene in Sect. 1.2 indicating how ubiquitous embedded control systems are in our daily life, and identifying the issues faced by industry when producing solutions of sufficient quality in a timely manner. In Sect. 1.3, we list the main challenges that we see in the development of this kind of product. In Sect. 1.4, we introduce a small fictional case and investigate it from the perspective of the different disciplines, illustrating the motivation for collaborative development. In Sect. 1.5, we explain how the Crescendo technology can address the challenges posed by embedded systems design. Section 1.6 concludes the chapter with a small summary.

---

M. Verhoef (✉)  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

C. Gamble • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk); [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

J. Broenink  
University of Twente, Enschede, The Netherlands  
e-mail: [J.F.Broenink@utwente.nl](mailto:J.F.Broenink@utwente.nl)



## 1.2 Setting the Scene

Computers are all around us and we use them every day, sometimes even without giving it a second thought. The term “computer” often refers to the Personal Computer (PC), which is used to send e-mail and browse the Internet or perhaps a video game console that is used for entertainment. But computers are also part of the alarm clock, coffee machine, dishwasher, video recorder, DVD player, camera, television set and mobile telephone. This class of systems is often referred to as “embedded systems” and you can easily count up to a hundred embedded devices in an average family household nowadays [25].

We become more and more dependent on the proper operation of these embedded systems. Not only because they are efficient and convenient to use but also because they potentially affect the quality of life. Sangiovanni-Vincentelli already mentioned in his key-note presentation [88] at the 2006 Design Automation and Test in Europe (DATE) conference that a modern, high-end car contains 80 microprocessors executing several million lines of code and this trend has continued to grow over the last decade. These microprocessors are used to control not only the car radio and air conditioning, but also the air bag, cruise control, fuel injection, brakes and power steering. A failure in any one of those critical embedded systems may have severe consequences. But the general public is typically not aware of this because these computers are deeply embedded in the system, hidden well out of plain sight. Dependability issues are typically associated with the military, medical or aeronautical domains but not so much with consumer or capital goods. For example, no one asks about the code coverage statistics of the power steering unit (an embedded system that contains a microprocessor which executes possibly several thousand lines of code) when you buy a new car. In 2004, Deutsche Welle reported<sup>1</sup> that the reliability rating of German cars, which used to be unrivalled and universally acclaimed, has been steadily decreasing for several years in succession as compared to their main competitors. Analysts believe that this may very well be due to the increased complexity as outlined by Sangiovanni-Vincentelli.

The economic relevance of embedded systems is easily demonstrated. For example, take mobile telephony. Market analysts such as Informa Telecoms & Media<sup>2</sup> predicted in 2005 that the number of mobile handsets deployed worldwide would reach one billion early in 2007 which corresponds to roughly 15 % of the population on Earth. Moreover, this target was reached in just 15 years and the market is far from saturated. Growth is continuing, at the time of writing there are an estimated 6.8 billion active mobile phone accounts.<sup>3</sup>

These numbers are just staggering, and it is obvious that such a market potential generates an enormous amount of pressure on the companies that build these kinds

---

<sup>1</sup>See <http://www.dw-world.de/dw/article/0,2144,1400331,00.html>.

<sup>2</sup>See <http://www.informatm.com/>.

<sup>3</sup>See <http://www.itu.int/>.

of products. Production volumes are extremely high, profit margins are typically low which implies that you have to reach the market with a new product before your competitor, in order to be economically successful. This so-called “time-to-market” (TTM) pressure is therefore the beast to beat. Companies like Nokia and HTC were mobile handset market leaders in 2010, but their marketshare has dropped spectacularly since Apple and Samsung introduced their smartphones. This also caused a spectacular shift in the market for software ecosystems. Microsoft ruled the personal computers era with Windows for two decades, but is now struggling to keep its marketshare in mobile computing (for tablets, smartphones) as they are competing against iOS (Apple), Android (Google) and Linux.

Companies invest huge amounts of money and effort in order to reduce the production time and cost price of their products. This has created a secondary economy consisting of companies that deliver (half-) products and services to achieve those goals. For example, Gartner<sup>4</sup> reports that the revenues for Electronic Design Automation (EDA) experienced double-digit growth in 2006, reaching 4.5 billion US dollars. Over the last decade, the market has stabilised to 5.7 billion US dollars per year (2012 figures), but this stabilisation is mainly due to the economic downturn that started in 2010, which caused investments to be postponed. But do all these investments lead to good products? Unfortunately not. It seems that the well-known adage “Price, Time, Quality - Pick Any Two for Success”, as is shown in Fig. 1.1, is still a fact of life.

After the famous CHAOS report from the Standish Group<sup>5</sup> appeared in 1994, there have been numerous published examples of projects failing or products malfunctioning [49]. Although the way these numbers have been gathered and reported has been criticised by the academic community [33], the same trends have also been independently demonstrated by, for example, the Software Improvement Group and Coverity. The latter company produces a yearly report in which they publish measured software defect rates of open source software [26], including the Linux operating system, the Network Time Protocol (NTP) and the Xbox media center (XBMC). Coverity analysed 380 million lines of code from 250 proprietary code bases, with an average code base size of 1.5 million lines of code. The average defect density (of high- and medium risks) measured was 0.69 per thousand lines of code. The trend since 2009 was that defect density is going up rather than down, not because of deteriorated quality of the code analysed but due to the improved performance of the static analysis tools used! In 2012, roughly 5,800 defects were detected in the 7.4 million line Linux code base and approximately 5,200 of those were fixed within the same year. Since 2006, 15,000 defects were found in total and to date some 8,400 are fixed. Despite these numbers, Linux is considered to be one of the most robust pieces of software as the mindset of the developers is to continuously improve the code base by using static analysis tools, among others.

---

<sup>4</sup>See <http://www.gartner.com/>, Doc. Id. G00143619.

<sup>5</sup>See <http://www.projectsmart.co.uk/docs/chaos-report.pdf>.



**Fig. 1.1** Decision making at large: how to find the optimum?

Despite efforts to improve the quality of computerised systems, it remains difficult to make error-free systems, as the previous example demonstrates. Most surprisingly, end-users seem to have accepted that as a given fact. People are used to reboot their computer if a problem occurs. If it does not work, you just download the latest software from the web site. Updates and upgrades have become part of the business model of the product. Even more so, only limited warranties<sup>6</sup> are provided and companies typically do not accept any liability from the use of their products. Would you buy a car if you would have to sign such a legal document? Open source software comes with a so-called “*as-is*” disclaimer, without warranty of any kind. The GNU General Public License<sup>7</sup> actually contains the following sentence: “*The entire risk as to the quality and performance of the program is with you.*” Yet, open source software is often believed to be of higher quality than most commercial software because it is exposed to public scrutiny.

Quality is a major issue in embedded systems development, mainly because of the production volumes involved. Intel Corporation was forced to recall a substantial number of their early Pentium processors in 1994 because a problem was found in the floating point unit after the product release. Harrison reported at the 2005 ForTIA Industry Day [71] that Intel wrote off 475 million dollars because of the Pentium FDIV bug and suffered considerable damage to their reputation. But even in a low-volume market things can go spectacularly wrong with great consequences. On June 4, 1996, the inaugural flight of the Ariane-5 rocket failed. About 40 s

<sup>6</sup>See for example the End-User Licence Agreement at <http://www.microsoft.com/>.

<sup>7</sup>See <http://gplv3.fsf.org/>.

after initiation of the flight sequence, at an altitude of about 3,700 m, the launcher veered off its flight path, broke up and exploded. The Cluster mission, consisting of four identical scientific satellites, was lost during this event. Conservative estimates suggest that this accident costed the European tax payer in the order of 300 million Euro. The Inertial Reference System (abbreviated in French: SRI), which is used to determine the attitude of the launcher, shut down mid flight because an exception occurred in the software calculating the current flight path. Virtually the same system had been used to launch Ariane-4 rockets successfully for many years, but it was used outside its original specification in this particular case. The investigation showed that the system was never tested under flight conditions despite suggestions from the responsible engineers. In fact, the Ariane-5 Accident Report [67] states: “... it was jointly agreed not to include the Ariane-5 trajectory data in the SRI requirements and specification.” However, the report does *not* state *why* this decision was made. It is commonly believed that the TTM pressure, to have this new generation launcher operational as soon as possible, may have contributed to this decision, taking into account the excellent track-record of a similar system on Ariane-4. Johnson reports on similar problems at NASA in [48]. The “Faster, Better, Cheaper” initiative, which was announced in 1998, fostered a culture in which engineers took considerable risks to innovate with new designs in order to meet requirements. In hindsight, TTM is one of the contributing factors [51] to the loss of the Mars Polar Lander mission in 1999.

But the tide of acceptance seems to be turning, in particular in the area of security. Here, users are less willing to accept failure in applications where trustworthiness and reliability are paramount, such as authentication, digital identity, privacy and on-line payments. In combination with the explosive growth of the Internet, a myriad of exploits have been reported due to software vulnerabilities in browsers, for example. An extensive overview of these can be found in the digest of the Usenet newsgroup `comp.risks`. An entire industry, lead by Symantec and McAfee, has grown around providing anti-virus and anti-malware scanners and services. These issues seemed to be isolated to the personal computer, but now it is also entering the realm of embedded systems. In 2008, the Digital Security group at Radboud University Nijmegen demonstrated the feasibility of breaking the cypher used on the Dutch public transport NFC payment card, in real time, by eavesdropping on the encrypted communication between the card and the terminal.<sup>8</sup> This enabled them, with very simple means, to copy the contents of the NFC card on the fly so that one could in principle travel for free. In 2013, the same research group demonstrated that it was possible to use a similar strategy to hack the secure wireless connection for remote controlled car keys, not only giving you access to the car, but also the ability to start it. Both examples attracted a lot of media attention<sup>9</sup> but despite the public outcry, instead of fixing the problem, the researchers were ordered to refrain from publishing their results. The real problem here is that it cannot be fixed

---

<sup>8</sup>See <http://www.ru.nl/ds/research/rfid/>.

<sup>9</sup>See, for example, <http://www.bbc.co.uk/news/technology-23487928/>.

by a mere software upgrade as it requires replacement of (potentially millions of) physical devices in the field. It is fair to say that the latter two examples are not related to software problems, but are a case whereby a priori assumptions about the robustness of certain cryptographic solutions are overestimated against the ever-increasing power of modern computers. Nevertheless, with the advent of software only solutions, such as openssl<sup>10</sup> and bitcoin,<sup>11</sup> one could imagine the economical impact in case these could be broken due to software errors.

### 1.3 The Embedded Systems Design Challenge

One might argue that the examples mentioned above are dated and do not reflect the current state of practice. But ongoing research, such as [26, 49], has shown that the average likelihood of projects succeeding has only *marginally improved* over the last decades despite substantial investments in tools and processes. It is generally believed that the performance in the embedded systems domain is rather worse than better. Why is this the case? Looking at general trends there are a few potential reasons.

**The design gap problem.** According to Moore’s law [75], the performance of hardware is roughly doubled every 18 months. But recent advances in networking, packaging and integration technology have enabled the development of heterogeneous embedded computing platforms that show a potential exponential growth in performance and thus complexity.<sup>12</sup> These platforms are commonly referred to as System-On-Chip or Network-On-Chip and usually combine multiple and interconnected radio-frequency, analog and digital components on a single chip. However, the technology we use to design the applications for these new platforms cannot keep up with this tremendous growth in capabilities, primarily because they are currently focused on designing single, monolithic systems. In other words, the complexity of the problem grows much faster than the capabilities of today’s leading design tools. This is commonly referred to as the “design gap.”

**The moving target problem.** Rapidly evolving technology and the constant quest for reducing cost-price forces designers of embedded systems to operate on the edge of what is technically feasible. In order to stay competitive, they sometimes need to adopt novel technology even while a product is already under development. One of the key problems in embedded systems design is the *validation* of these design decisions. How much effort and time does it take to check that the intent of a design choice works out well in practice? Over-dimensioning is the usual approach to accommodate for uncertainty in the design, but this is typically not economically

---

<sup>10</sup>See <http://www.openssl.org/>.

<sup>11</sup>See <http://bitcoin.org/>.

<sup>12</sup>International Technology Roadmap for Semiconductors, see <http://public.itrs.net/>.

viable because it usually increases the cost price. Sometimes actual prototypes need to be built in order to assess the feasibility of some potential solution. Managing this process is regarded as the key to success, and it is often referred to as “shooting at a moving target.”

**The requirement versus design paradox.** Making design decisions in the early phases of the system life-cycle is notoriously difficult. In this stage, requirements are often unclear and under-specified, at best leading to a long list of properties that the system shall eventually satisfy. In the past, emphasis has been put on managing the requirements process, such that sufficient information is available at the time the design decisions are made. However, this is often not realistic, in particular in the domain of embedded systems. At the time when requirements are elaborated, the major architectural design decisions also need to be taken, primarily in order to meet the TTM target for the product. But how can one make these crucial decisions when there is still so much uncertainty? This is in particular true for performance criteria that the system must meet because they are in general surprisingly hard to quantify and evaluate. It is obvious that elaboration of the requirements is guided by the chosen architecture but in turn the definition of the architecture depends on clear and unambiguous requirements. System architects have to deal with this paradox, for example, by applying iterative development processes in order to close the design loop.

**Late closure of the design loop.** Distributed real-time embedded control systems are inherently complex and so is the associated design process. Many implementation choices need to be made, and the impact of each decision is difficult to assess due to this complexity. This makes the design process error prone and vulnerable to failure as other downstream design choices may be based on it, causing a cascade of potential problems. Moreover, it may take some time to realise that a decision is wrong because it requires feedback in the design process. Usually this happens late in the life-cycle, at system integration and testing or even product manufacturing. The repairs required to fix these problems cause significant project delays and cost overruns or sometimes even worse: product cancellation.

**Multidisciplinary design.** Systems are traditionally designed in a mono-disciplinary style usually with an organisational structure to reflect this (e.g. mechanical department, electronics department, software department and so on). While in the past systems were developed out-of-phase (mechanical design precedes electrical design which in turn precedes software design), nowadays concurrent engineering is applied in order to save development time. However, system-level requirements that cannot be assigned to a single discipline, such as performance, typically cause great problems during the integration phase because the responsibility to meet the requirement is shared among all disciplines. The root cause of this problem is the lack of cross-discipline design interaction. This problem cannot be solved by improving the internal organisation; the way (embedded) software is currently being developed is fundamentally different from, for example, mechanical and electrical design. These engineers basically speak a

different language, are concerned about different types of problems and use different techniques to address and solve these problems. This challenge is dominant in the embedded systems domain because the computer and the device it controls both lose their function if they were to be separated. Hence, they cannot be designed in isolation which makes the cross-discipline communication mandatory.

**Dependability and fault tolerance.** As embedded systems become more pervasive, they face ever more demanding and interdependent functional and non-functional requirements, including the need for reliability and fault tolerance, performance and interoperability. In addition, they are increasingly distributed in character, with multiple processors connected with different forms of network, introducing a wider range of alternative architectures and faults for controllers. In order to achieve dependability, the next generation of embedded systems applications must be engineered to provide predictable behaviour in the face of faults, including malfunctioning infrastructure, environmental hazards, malicious intentions, design defects and degraded component services, any of which can have very damaging consequences if fault tolerance mechanisms are not in place.

## 1.4 Embedded Systems Design: An Illustrative Story

To illustrate how some of the problems identified in the previous sections can affect a company producing embedded systems under strong market pressures, we present the story of such a development in Fig. 1.2. In this story, a fictional agricultural vehicle company attempts to adapt to a rapidly changing market. Their aim is to maintain their market share by producing a new version of their flagship tractor system, the T1X, while competing against a rival firm. Unfortunately, due to problems during the development, production is delayed significantly and this allows their rival to release a competing product sooner and gain the upper hand in the market.

Although this story is fictional, we have drawn on the experiences of the industrial partners who participated in the development of our method. In the following sections, we consider the different perspectives of the engineers and software designers in this failed development and from this motivate the need for collaborative development that underpins our approach.

### 1.4.1 *The Control Engineers' Perspective*

When given the task of building an autonomous GPS-controlled tractor, the control engineers at OPECorp (Fig. 1.2) focus on designing the control laws. They need to find suitable parameters that can cope with the range of different agricultural machines hauled by the tractor and they cannot rely on farmers updating the

OPECorp (Old Physics Engineering Corporation) is a long-established manufacturer of agricultural equipment. In particular they have a historical reputation for producing reliable tractors. While they used to hire mainly mechanical and electrical engineers, in more recent years with the addition of computer controllers to their tractors, they have established a strong software team.

While OPECorp was the market leader for many years, their market share has been eroded recently by competitors producing rival products, in particular Upstart Farm Technologies (UFT). Both OPECorp and UFT produce a range of agricultural machinery (e.g. plows, harrows, seeders) that can be hauled by their tractors. Their tractors provide both hydraulic power and digital data connections for more advanced accessories that sort of follow a standard. The management of OPECorp are keen to maintain their market position and aim to produce a new version of their flagship tractor, dubbed the T1X. The T1X will include a new GPS-controlled autonomous mode. OPECorp expect that UFT are also working on a new tractor on a similar time scale (18 months).

To begin, the control engineers design control laws for the new tractor, which must support the range of agricultural machines that can be hauled by the tractor. Once the control engineers are happy with their low-level control laws, the software team takes over the task of building the software of the T1X. Meanwhile, a GPS unit has been selected and work begins on fitting it to the T1 test rig that the company always uses for new products. With six months to production, UFT announce at an agricultural trade fair that their rival product will contain the latest GPS unit from market leader Loc8, with much higher accuracy than OPECorp's current choice for the T1X. In a surprise move, they also announce compatibility with third-party agricultural machinery from the Ploughman Group (PG). The management at OPECorp insist that these features are incorporated into the T1X, in order to keep up with their rivals. This means the engineers have to fit the Loc8 GPS unit to the prototype and discover that it has to be placed in a different position on the vehicle to the previous GPS unit. They also have to build an adapter for the digital interface used by the PG machines. The software team must now adapt their communication protocols to deal with the PG machines as well. Here it also turns out that the "standard" is not interpreted in the same way by all vendors so it is not as easy as expected.

Once the prototype reaches field trials, the deadlines for the start of full production are looming. The software is loaded onto the prototype machine and trials begin. Unfortunately things go wrong from the start. The tractor drifts steadily from its intended path in autonomous mode. When a PG machine is attached to the T1X, the prototype shuts down, going into safety mode. The trials have to be abandoned and a large amount of money and resources are wasted. It is discovered back at the factory that the new position of the Loc8 GPS unit was not communicated to the software team, which threw off the controllers calculations and caused the drift. The shut down when the PG machine was connected occurred because the software used a library to encode and decode packets used in OPECorp's proprietary communication protocol, which made assumptions about the packets that did not hold true for the PG system. So while the high-level messages were right for the PG machine, the library couldn't decode the low-level packets and didn't recognise messages from the PG machine, causing a safe-mode shutdown. It takes three months to fix these problems found and another month to set up a second set of trials, delaying the release of the T1X significantly and giving an advantage to UFT, whose tractor is released on-time to much praise.

**Fig. 1.2** A fictional story of a failed embedded systems development



software once the machine is in production. The different machines hauled by the tractor will affect the balance of the system as a whole, and they want to be sure that the system can be controlled safely and stably. This low-level control is their main area of expertise. To do this, the engineers build a Continuous-Time (CT) model of the T1X and run simulations to tune control parameters that suit the various types of machines.

### ***1.4.2 The Software Designers' Perspective***

The software engineers at OPECorp (Fig. 1.2) receive the low-level control parameters from the control engineers, but from experience they know this only forms 10% of their software for a tractor controller. They have experience in building controllers for the T1 family of tractors and their main focus is on supervisory behaviour. When tasked with adding autonomous control features to the T1X, their experience tells them that mode changes (e.g. changing to and from autonomous mode) and fault tolerance (e.g. dealing with problems caused by the environment that the T1X will find itself in) will form the majority of the control software. To handle this complexity, they build Discrete-Event (DE) models to capture and analyse this mode-changing behaviour.

### ***1.4.3 The Case for Collaborative Development***

Heemels and Muller [42] identified a key set of issues that seem to be root causes behind the problems outlined in previous sections:

1. Reasoning about system-level properties is difficult because a common language is lacking. Each engineering discipline uses its own method, vocabulary and style of reporting. This incompatibility causes confusion often leading to misunderstandings and wrong assumptions being made on the sub-designs of other disciplines. These inconsistencies are hard to spot because there is usually no structured system design reasoning process in place.
2. Many design choices are made implicitly, usually based on previous experience, intuition or even assumptions. System-level reasoning is made difficult if the rationale behind such a decision is not quantified. The reasons are sometimes kept hidden on purpose, for example, if strong personal preference or politics plays a role. This may perhaps lead to a local optimum in the system design but only rarely to a global optimum. It is therefore necessary to make design knowledge explicit in order to enable the dialogue at the system level.
3. Dynamic or time-dependent aspects of a system are complex to grasp, and moreover, there are not many methods and tools available to support reasoning about time varying aspects in design, in contrast to static or steady-state aspects.

The effects of these points are amplified by the complexity of the product under development (a typical high-tech system consists of tens of thousands of components and millions lines of code) and the complexity of the design process (number of people involved, organisational structure, out-of-phase or multi-site development, etc.) Our hypothesis is that lightweight and abstract models that capture the system-level behaviour and a reasoning method that indicates how and when to use them will reduce these tensions considerably. A good system engineering methodology will expose implicit or hidden design choices and replace “hand-waving” by design rationale based on objective, quantified and verifiable information.

## 1.5 A Solution: The Crescendo Approach

The aim of our approach is to bridge the gap between control engineers and software designers by providing tools and methods that allow them to collaborate in the design of distributed embedded control systems. Our approach allows them to explore and test their designs without resorting to expensive prototypes, thereby helping to manage the risks of embedded systems development. The essence of our approach is to use models in all phases of the design. These models describe the system in a more coarse way in the beginning and are gradually refined and extended in later phases. Each result of a step in the design cycle is a combination of design decisions and models, which are used for that decision, and are also the starting point for the next design step. We refer to this as *model-based design*.

A single modelling formalism may be effective for a simple non-distributed system, but it rarely scales. Continuous-time models are excellent for modelling physical system dynamics, but if a distributed architecture is needed or if the idealised assumptions do not hold, the model can become hard to maintain. Discrete-event models are excellent for expressing system logic, making it possible to incorporate a distributed system architecture and failure assumptions, but physical laws are difficult to model. Combining these two worlds, if done in a semantically sound way, can provide a collaborative modelling (*co-modelling*) framework in which it is possible to experiment with both discrete and continuous aspects of the whole system. This makes it possible to run a combined simulation (a *co-simulation*) of both continuous and discrete models, observing the consequences of, say, a change in process distribution or a sensor failure, on the full control system and the controlled process. Changes in models can be caused by changes in requirements or component capabilities, or by deliberate refactoring. In particular, changes in one model may have ramifications on others. The co-modelling approach that we discuss in this book could help address this because the impact of changes can be assessed immediately, prior to the integration of separately developed constituent parts.

To deal with the first five design challenges mentioned in Sect. 1.3, use of models and simulation needs to start from the beginning of the design process. The Crescendo methods and tooling support collaborative modelling and simulation

in all stages of design, including early stages, and so aim to support a concurrent engineering approach. To address the sixth challenge, relating to dependability and fault tolerance, we provide guidelines and examples to enhance initial co-models with realistic behaviour, faults and fault-tolerance solutions. Testing via co-simulation allows designers to explore several possible designs, allowing them to assess these trade-offs in the early phases of the design cycle. As such testing can be laborious, especially when the number of alternatives in combination with several fault situations can explode, the Crescendo automated testing tool can facilitate this work.

But perhaps the most important aspect of Crescendo is not the potential productivity improvement, but the ability to create abstract and multi-disciplinary system-level models that are *competent*. In other words, the predictions obtained from the models match, within some acceptable error margin, the behaviour of the real system that is under development. At the end of the day, this is what modelling is all about, since the overall aim is to raise the confidence in designs, especially in the early phases of the product development life-cycle.

## 1.6 Conclusion

There is a pressing need for collaborative approaches to modelling in embedded systems design, and presented our approach embodied in the Crescendo tools. In this chapter, we have reviewed the design challenges facing the embedded systems sector, including increased “TTM” pressures and the need for greater dependability. We identified critical problems that occur during developments, including the gap between existing design methods and rapidly changing technology, the challenge of volatile requirements and a need for late closure of the design loop, as well as the necessity for multiple disciplines to collaborate for a design to be successful.

Using a fictional account of an embedded systems design, we illustrated some of these challenges and presented the rather different perspectives of engineers and software designers. Although fictional, the story reflects the experiences of industrial practitioners who participated in the development of the methods and tools described in the remainder of this book. We explained the essential contribution of the Crescendo approach, stressing three key points:

- using a *model-based* approach from the beginning of design;
- allowing multidisciplinary, collaborative modelling; and
- supporting dependability from early in product development by fault modelling, fault injection and automated testing to check fault tolerance.

# Chapter 2

## Co-modelling and Co-simulation in Embedded Systems Design

John Fitzgerald and Kenneth Pierce

### 2.1 Introduction

This chapter introduces the first basic concepts of co-modelling and co-simulation, including notions of system, model and co-model, simulation and co-simulation, etc. It also describes the ways in which co-modelling and co-simulation can be integrated with established development processes such as IEEE 15288 (*Systems and Software Engineering—System Life Cycle Processes*, [45]) and IEEE 12207 (*Systems and Software Engineering—Software Life Cycle Processes*, [44]).

The collaborative development of an embedded system requires productive interaction between engineers from very different backgrounds. Control engineering and software engineering have matured over many decades, each with its own philosophy, methods and terminology, and so it is necessary to clarify the common ideas that underpin co-modelling and co-simulation. This chapter introduces these concepts, including the ideas of system (Sect. 2.2), model (Sect. 2.3), co-model (Sect. 2.4), co-simulation (Sect. 2.5) and design space exploration (DSE) (Sect. 2.6). Realising collaborative modelling and co-simulation within established development processes is considered in Sect. 2.7. Finally, Sect. 2.8 provides a summary of the chapter.

### 2.2 Systems and System Boundaries

We build models in order to assist in the design of *systems*. We regard a *system* as an entity that interacts with other entities, including hardware, software, humans and the physical world [6]. The system may itself be a group of interacting or

---

J. Fitzgerald (✉) • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk); [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

interdependent items forming a coherent whole [4]. The *system boundary* defines a frontier between the system and the entities that form its *environment*. The developer can exercise some choice over the design of entities within the boundary of a system of interest. By contrast, the laws governing the behaviour exhibited by the environment are beyond the developer's direct control.

In an embedded system, the entities within the system boundary may be digital computing elements or physical elements such as machines. The environment provides *stimuli* to the system, and the resulting behaviour of the system, visible at its boundary, is termed its *response*. Embedded control systems are typically thought of as being composed of a *controller* and *plant* ("that part of the system which is to be controlled" [43]). The controller contains the control laws and decision logic that affect the plant directly by means of *actuators* and receive feedback via *sensors*.

Experience suggests that, while control engineers and software engineers might broadly agree on these definitions, they will have a natural bias towards some aspects of a system. For example, software engineers may see the environment as everything outside of the computing part of the system, including the plant, whereas control engineers may focus mainly on the plant as the system. Communication is therefore required in a co-modelling project to ensure common understanding of where the boundaries of influence and responsibility lie in the design process.

## 2.3 Models

In this book, we focus on the use of *models* to describe designs during product development. The act of creating models is called *modelling*. A model is an abstract description of the reality of a putative system [4]. The model is abstract in the sense that it omits details that are not relevant to the *purpose* for which the model is constructed. For example, a model of an aircraft flight control system intended to ensure smooth response to pilot commands may omit details of the cockpit layout, but would instead focus on the commands that can be generated to the control surfaces. Models that are expressed with sufficient clarity and precision may be analysed to confirm or refute the presence of desirable characteristics or the absence of undesirable properties. This helps developers to control risk by providing assurance of design characteristics before expensive commitments are made to implementation in target software and hardware.

A model may contain representations of the system, environment and stimuli. We regard a model as being *competent* for a given analysis if it contains sufficient detail to permit that analysis. Models may be analysed by inspection or by formal mathematical analysis. Many models are also *executable* in that they may be performed as a sequence of instructions on a computer; such an execution is termed a *simulation* because the behaviour exhibited is intended to simulate that of the system of interest.

A *design parameter* is a property of a model that can be used to affect the model's behaviour, but which remains constant during a given simulation. A *variable* is part of a model that may change during a given simulation. We consider *code generation* to be the process of implementing a controller by automatically translating a model into some programming language, which can then be executed on the real computer hardware of the system.

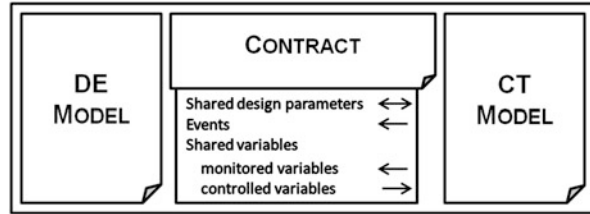
Embedded systems contain both computing and physical elements, and so we expect that the models describing these may be quite different in nature. In this book, we focus on two types of models: discrete-event and continuous-time. In a Discrete-Event (DE) model, “only the points in time at which the state of the system changes are represented” [83, p. 15]. Discrete-event modelling is typically used for digital hardware [5, 68, 93], communication systems [11] and embedded software [22]. In a *continuous-time simulation*, “the state of the system changes continuously through time” and the simulator “approximates continuous change by taking small discrete-time steps.” [83, p. 15]. By contrast, Continuous-Time (CT) modelling uses differential equations and iterative numerical integration methods to describe dynamic behaviour. Continuous-time modelling is typically used for analogue circuits and physical processes [68].

In this book, we set out to answer the question of whether DE and CT models can be brought together in a sound but practically useful way to enable the early-stage collaborative design of embedded systems. The principles and experience that we present can be applied to a wide range of notations and tools for CT and DE modelling. However, we have realised the approach using two particular formalisms: *bond graphs* [16] for CT models and Vienna Development Method (VDM) [37, 50] for DE models. VDM models can be constructed, animated and analysed using the tool *Overture* [56], which provides natural features for describing software structures and behaviour. In the same way, bond graphs can be supported by the tool *20-sim* [53] which allows the plant to be modelled in several ways, including the powerful bond graph [52] notation which permits domain-independent description of the dynamic behaviour of physical systems. *Overture* and *20-sim* are linked by a new tool *Crescendo*, which allows models expressed in the two formalisms to be developed and analysed together. *20-sim* and VDM are introduced in depth in Chaps. 3 and 4.

## 2.4 Co-models

Our approach focuses on system models that are composed of a DE model of a controller and a CT model of a plant (called “co-models”). The DE and CT models are referred to as *constituent models*. Interaction between the DE and CT models is achieved by executing them simultaneously and allowing information to be shared between them. This is termed a *co-simulation*. In a co-simulation, a *shared variable* is a variable that appears in and can be accessed from both the DE and CT

**Fig. 2.1** A co-model contains a DE model, contract and CT model, where a contract may define shared design parameters, events and shared variables



models. Design parameters that are common to both models are called *shared design parameters*.

An *event* is an action that is initiated in one model and leads to an action in another model. Events can be scheduled to occur at a specific time (*time events*) or can occur in response to a change in a model (*state events*). State events are described with predicates (Boolean expressions), where the changing of the local value of the predicate during a co-simulation triggers the event. In our approach, events are referred to by name and can be propagated from the CT model to the DE model within a co-model during co-simulation.

Shared variables, shared design parameters and events define the nature of the communication between constituent models. These elements are recorded in a *contract*. For each shared variable, only one constituent model (either the DE model or the CT model) can be assigned write access to it. In the control-system paradigm, shared variables written to by the DE constituent model are called *controlled* variables and those written to by the CT constituent model are called *monitored* variables. A co-model is a model comprising a DE model, a CT model and a contract. Note that a co-model is itself a model and that a co-simulation can therefore be described succinctly as the simulation of a co-model. Figure 2.1 shows a hierarchy of the concepts relating to a co-model.

For a co-model to produce simulation results that can be trusted, the DE and CT models must be *consistent* with each other. Consistency can be broken down into two parts. If the models agree on the identities and data types of the variables, parameters and events they share, then they can be said to be *syntactically consistent* with each other. Achieving syntactic consistency alone does not guarantee that the simulation will produce trustworthy results. For that, the models must also agree on the semantics of the variables, parameters and events they share. If this agreement is reached, then the models can be said to be *semantically consistent*. Only when the DE and CT models are both syntactically and semantically consistent can we say that the co-model is consistent and only then can we place trust in its results.

We suggest that at a minimum, the following should be recorded about each contract entry: the SI unit<sup>1</sup> or a simple description of the value, the range of acceptable values, the datum against which a value is measured, and the direction

<sup>1</sup>The international system of units, abbreviated SI from French: *le Système Internationale d'unité*.

of positive values or frame of reference. For events, the condition under which the event will be raised should be recorded.

## 2.5 Co-simulation

Simulation of a co-model is called *co-simulation*. During a co-simulation, the DE and CT simulator have responsibility over their own constituent models. Overall coordination and control of the co-simulation is the responsibility of a *co-simulation engine* that is responsible for the progress of time in the co-simulation and the propagation of information between the two constituent models. Crescendo acts as such an engine.

Figure 2.2 shows the co-simulation engine interacting with the DE and CT simulators. The thin arrows indicate inputs and outputs. The DE simulator and CT simulator take a DE model and a CT model as input, respectively. The contract and scenario are inputs to the co-simulation engine. The co-simulation engine outputs a set of results (representing the outcome of the co-simulation). The large arrows indicate data exchange between the co-simulation engine and the two simulators. Note that the simulators do not communicate directly.

### 2.5.1 The Co-simulation Engine

In order to allow coherent co-simulations to be performed, it is important to reconcile the semantics of two simulation tools from different domains. This is covered in detail in Chap. 13. At this stage however, it is useful to understand the basic operation of a co-simulation and of the co-simulation engine.

Figure 2.3 presents an abstract view of the synchronisation scheme underlying co-simulation between a DE simulation of a controller (top) and a CT simulation of the plant (bottom). The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and simulation time in both linked simulators (the co-simulation engine is not shown explicitly in Fig. 2.3).

Each simulator maintains its own local state and internal simulation time. At the start of a co-simulation step, the two simulators have a common simulation time. The granularity of the synchronisation time step is always determined by the DE simulator. The scheme does not require resource-intensive rollback of the simulation state in either of the simulators, though rollback may occur inside the CT simulator in order to catch the precise time requested, i.e., when a zero crossing is detected in an equation.

At the start of a co-simulation step ( $t_n$  in Fig. 2.3), the DE controller simulation sets the controlled variables and proposes a duration by which the CT simulation should, if possible, advance. The co-simulation engine communicates this to the CT



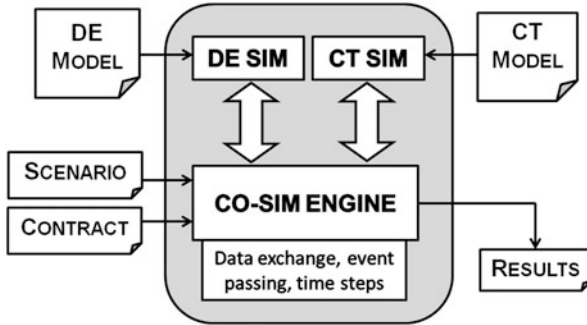


Fig. 2.2 Tool-oriented perspective of a co-model

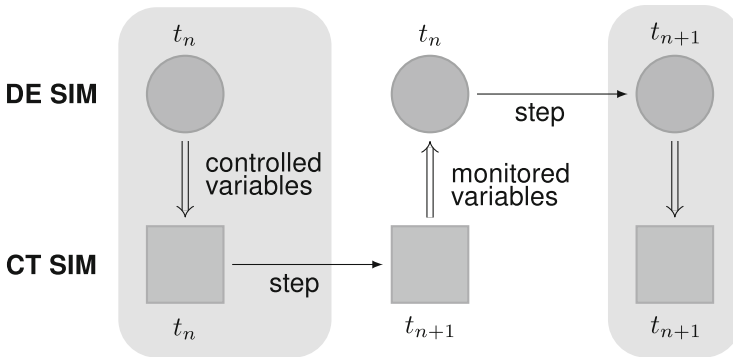


Fig. 2.3 Example of the synchronisation scheme for DE-CT co-simulation

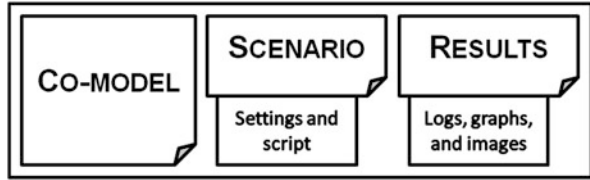
simulator. The CT simulator then tries to advance its simulation time. If an event occurs before the proposed step time is reached, the CT simulator stops early so that the DE simulator can be notified of the event. Once the CT simulator has paused (reaching internal time  $t_{n+1}$ ), the monitored variables and the actual time reached in the CT simulation are communicated back to the DE simulator. The DE simulation then advances so that both DE and CT are again synchronised at the same simulation time.

### 2.5.2 Scenarios

To predict a system’s behaviour using a co-model, it is often desirable to try out a number of *scenarios*, in which certain aspects are varied, including the setup of the modelled system, simulated user inputs and faulty behaviours. Scenarios are realised through two features: co-simulation *settings* and *scripts*.

The *settings* configure a co-simulation before it begins. Settings include: selection of alternative components from within the co-model, setting of design param-

**Fig. 2.4** A co-simulation run comprises a co-model, scenario and test results



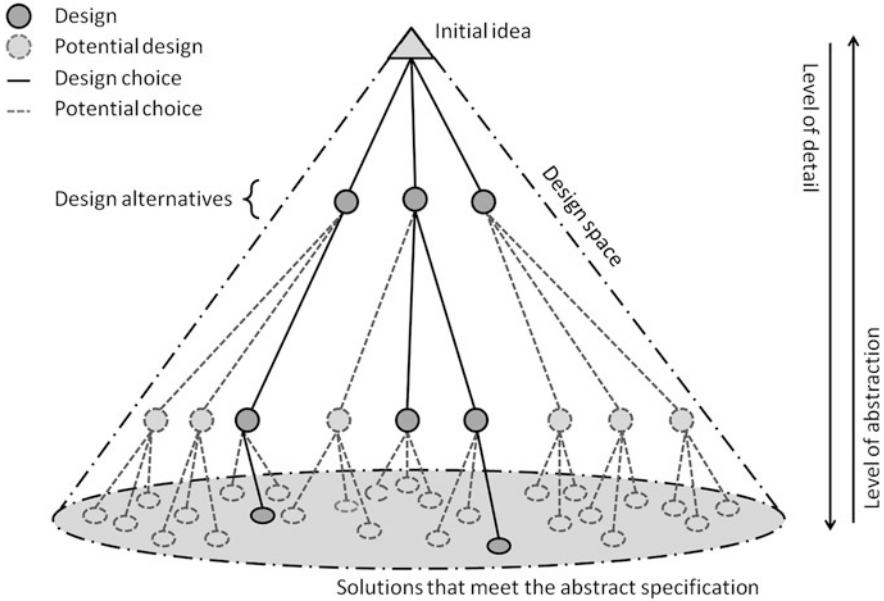
eters and various tool settings such as co-simulation duration and choice of integration method. A *script* may influence a co-simulation during execution by changing selected values in the co-model. Values can be changed at a given time or in response to a change in the state of the co-model. Scripts are defined using a simple, domain-specific language and are contained in a *script file*. Scripts can be used for fault activation and for mimicking user inputs.

The output from a co-simulation is a *test result* that may take a number of forms, including a *log* of data collected during execution for post-simulation analysis and 2D and 3D plots to allow the simulation state to be observed more immediately. The combination of a co-model, a scenario and corresponding test results is called a *co-simulation run*. Figure 2.4 shows the elements that make up a co-simulation run.

## 2.6 DSE and Automated Co-model Analysis

As with other model-based techniques, our approach can be used to test a range of solutions while creating a design. We view the *design space* as the set of possible solutions for a given design problem, and *DSE* is an activity undertaken by one or more engineers in which they build and evaluate co-models in order to reach a design from a set of requirements. Where two or more co-models represent different possible solutions to the same problem, these are called *design alternatives*. Each choice involves making a selection from alternatives on the basis of criteria that are important to the developer (e.g. cost, performance). The alternative selected at each point constrains the range of designs that may be viable next steps forward from the current position. Figure 2.5 illustrates the concept of DSE.

Crescendo aids DSE by supporting the selection of a single design from a set of design alternatives. Ranges of values for co-model settings can be defined before the tool then runs co-simulations for each combination of these settings. Results are stored for each simulation and can be analysed. We call this feature *Automated Co-model Analysis (ACA)*. One way to analyse these results is to define 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 can be applied to the test results, producing *analysis results* that contain the rank(s) for each design simulated.



**Fig. 2.5** A cone symbolising exploration of the design space and showing how a choice restricts further designs, inspired by [25]

## 2.7 Co-simulation in Practice

The approach proposed in this book may be applied in the concept definition phase in order to clarify the optimal system-wide requirements for the different parts of the system. However, if it is not just used at the very early stages, successful use of our approach may rest on integrating it with existing practices and design flows. This may involve identifying how our approach fits existing standards. In addition, decisions about how to bring co-simulation into a design flow where model-based design is already used will depend on the type of modelling done previously and the competencies of the team involved. These two aspects also influence the choice of how to build an initial co-model. We offer some insights into these points in the following sections.

### 2.7.1 Where Does Co-simulation Fit with Existing Practice?

To help with getting a feel for how our approach can fit into existing practices, we describe how our approach can map into the following existing standards: ISO/IEC and IEEE standards 15288 [45] and 12207 [44]; ECSS-E-40 [31] (Space Engineering—Software) and ECSS-Q-80 [32] (Space Product Assurance—

Software Product Assurance); and the Rational Unified Process (RUP) [82]. While these workflows differ in some ways, they have two key properties in common. First, none of them mandates a particular life cycle, but they do identify processes that form part of life cycles that can be implemented in specific projects and development organisations. Second, it is possible to identify a core progression of processes that holds across all of these frameworks.

The development process starts with something that needs to be designed. This is the *operational concept* in IEEE 12207 or the *vision* in the RUP. From here, each of the four workflows defines a set of ordered processes that occur in a development (described as *requirements for engineering* in ECSS-E-40, *technical processes* in IEEE 15288/12207 or *phases* in the RUP). All four workflows broadly adhere to the following pattern:

- Requirements definition
- Requirements analysis
- Architectural design
- Detailed design
- Implementation/integration
- Operations and maintenance

Collaborative modelling and co-simulation can have a role in several of these processes. In IEEE 12207 terms, Crescendo forms an “enabling system” supporting parts of the system life cycle, notably the more upstream technical processes. Relating to ECSS-E-40, Crescendo represents “tools and supporting environment”. We would expect to see applications of collaborative modelling and co-simulation as follows:

**Requirements definition:** During elicitation, requirements can be expressed in terms of a co-model or less formally. Defining the stakeholder requirements includes the development of representative activity sequences or use cases that help to elicit requirements that may not have been explicitly stated. Co-models and co-simulation can help subsequent analysis and maintenance of stakeholder requirements to identify areas of ambiguity or incompleteness and the communication back to the stakeholders of these deficiencies. A collaborative model allows system elements, continuous and discrete, to be expressed in the appropriate formalism, and this in turn may make the model easier to communicate to stakeholders.

**Requirements analysis:** A representation of a technical system (for example, a co-model) that meets the requirements is built. It involves the definition of a system boundary and of the services delivered at the boundary. Here, we expect co-models to be valuable in considering in depth alternative boundaries and functions. IEEE 15288 states that “System requirements depend heavily on abstract representations of proposed system characteristics and may employ multiple modelling techniques and perspectives to give a sufficiently complete description of the desired system requirements” (IEEE 15288, Clause 6.4.2.3).

**Architectural design:** This process involves the allocation of responsibilities to units in a solution architecture, each unit having defined internal or external interfaces. From the perspective of co-simulation, the key part of this process is the evaluation of alternative design solutions. Expressed as co-models, these alternatives can form the basis of trade-off and risk analyses.

**Detailed design:** Here, the design of the units in a solution architecture is built. By this stage, a single design should have been chosen from the set of alternatives. The constituent models of the co-model can then be used to explore the detailed design of the chosen solution and co-simulation used to test the evolving design.

### ***2.7.2 Developer Background and Legacy Models***

The choice of how to begin co-modelling can be influenced by the skills of the development team and whether or not legacy models exist. Legacy models are models that already exist and that relate to the system under design. These might include existing models of the system as a whole in a single formalism; models of a part of the system, such as a CT plant model; models of potential components of the system; or models of other systems or components that relate to the system under design. Legacy models, such as existing plant models, might be used directly or could simply be used as a reference. Another potential source of modelling information is in the form of prior art, existing implementations or other prototypes; these can provide valuable measurements or simply inspiration. It is useful to identify these models and sources before modelling begins.

The skill set of the co-model development team is another factor that can influence the way in which our approach is adopted. Perhaps, the “ideal” make-up for a team would be a group of experienced modellers from both the DE and CT domains who understand enough of the mindset within the other domain to communicate and collaborate effectively. Naturally, the real-world environment is unlikely to be so idyllic; therefore, it is a good idea to consider the skills of the team upfront. Software engineers with experience of object-oriented language should not find the move to VDM-RT difficult. Similarly, experience of other CT formalisms such as Matlab should permit a smooth transition to 20-sim for modellers. Note, however, than a team entirely composed of DE or CT experts should be careful not to be overly biased by their backgrounds.

### ***2.7.3 Paths to Co-modelling***

Building a first co-model is a big step towards adopting our approach. We define three “standard” paths to reach a first co-model, which are based on the structure of a co-model. Chapter 8 explores the following paths in much greater detail:

**DE-first:** Here, initial models are produced in the discrete-event formalism before introducing a CT model to form the initial co-model. The focus is on developing the DE controller first.

**CT-first:** In this approach, initial models are produced in the CT tool, with a DE model being introduced later to form a co-model. The focus is on modelling the dynamics of the plant.

**Contract-first:** In this third approach, a contract is defined initially. The constituent models are then developed separately but concurrently, following the respective DE-first and CT-first approaches. The contract acts as a guide and target for constituent model development. This allows for early testing of constituent models without reliance on a competent counterpart model. The constituent models are then integrated into a co-model.

## 2.8 Conclusion

In order to realise the potential of co-modelling and co-simulation technology, we need to take account of established modelling techniques and practices, rather than abandoning trusted approaches. In this chapter, we have outlined the concepts, semantics and pragmatics of co-modelling and co-simulation in our framework. After introducing the basic concepts, we briefly discussed the mechanics of co-simulation between DE and CT simulation engines. We indicated the potential of this approach as a means of exploring alternative designs and described ways in which this can be aligned with existing design flows, with reference to standard development processes including IEEE 15288 and 12207. We have only described the bare bones of the approach; the remaining chapters flesh it out by describing the DE and CT formalisms on which it has been realised, the tool support developed and the practical experience of several substantial industrial applications.

# Chapter 3

## Continuous-Time Modelling in 20-sim

Job van Amerongen, Christian Kleijn, and Carl Gamble

### 3.1 Introduction

This chapter provides an introduction to continuous-time modelling of physical systems, written with the DE domain expert in mind. Fundamental concepts such as bond graphs and differential equations are presented, together with graphical representations (block diagrams and iconic diagrams) used in the 20-sim tool. Control architectures in the form of feedforward and feedback controllers are briefly discussed, including sensors and actuators. Modern controllers are mostly implemented in computers. The sampled data controllers in the computer are coupled to the continuous-time plant by analogue-to-digital and digital-to-analogue converters. The roles of the sampling rate, the number of bits in the converters and the arithmetic as well as event handling are shortly discussed.

As a running example, we will use the experimental setup of Fig. 3.1. In this setup, 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. This model is representative for a large class of mechatronic systems.

When we try to model such a physical system, a first step is to identify subsystems in the system. The *causal relation diagram* sketched in Fig. 3.2 shows how we can recognise subsystems in the electrical and mechanical domain, coupled by means of a transducer in the form of an (electric) motor. All these components

---

J. van Amerongen (✉)  
University of Twente, Enschede, The Netherlands  
e-mail: [J.vanAmerongen@utwente.nl](mailto:J.vanAmerongen@utwente.nl)

C. Kleijn  
Controllab Products, Enschede, The Netherlands  
e-mail: [Christian.Kleijn@controllab.nl](mailto:Christian.Kleijn@controllab.nl)

C. Gamble  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk)

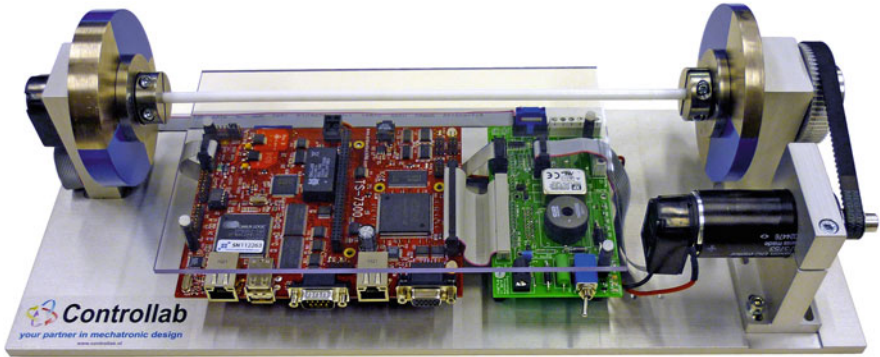


Fig. 3.1 A servo system with mechanical and electrical parts

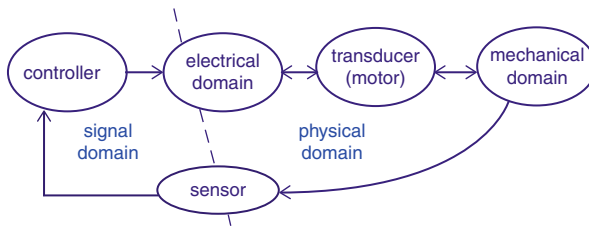


Fig. 3.2 Causal relation diagram of the system of Fig. 3.1

are physical components that interact with each other. When we want to control the angle or angular velocity, we need a sensor and a controller. The sensor produces a signal that is used in the controller to steer the electric motor.

We can further detail the different balloons in Fig. 3.2 and continue this process until we arrive at subsystems that cannot be split anymore. We call the models which describe these subsystems *elementary models*. The elementary models can be described by basic physical elements, e.g. a mass, a spring or friction. We refer to these elementary models as *ideal physical models* (IPMs). A real spring, as a component, not only has the property of being a spring, it also has a certain mass and damping. A *component* can be built from elementary, ideal models. In the next section, elementary phenomena in the mechanical, electrical and hydraulic domain will be described. For these elementary models, common icons in the mechanical and electrical domains will also be given.

Section 3.2 discusses the basic equations in electrical, mechanical and hydraulic systems and brings these equations in a generalised form. This emphasises that dynamical systems in different domains can be described by the same equations. Then Sect. 3.3 illustrates how iconic diagrams can be used to represent such generalised forms of equations. In Sect. 3.4, a domain-independent notation is introduced in the form of bond graphs. Bond graphs are used in 20-sim, either explicitly or hidden behind iconic diagrams. This so-called port-based modelling



enables the use of models which are very close to the physical reality and which can easily be extended or made more detailed with additional components.

Section 3.5 shows how these systems can be simulated in 20-sim. 20-sim is a tool that fully supports physical modelling by providing graphical notations, which are closely related to the physical world, such as iconic diagrams and bond graphs. In addition, block diagrams and equations can be used as input formats. 20-sim supports hierarchical modelling and has powerful integration algorithms as well as tools for symbolically or numerically solving algebraic loops.

Sensors and actuators form the link between the physical world and the signal-based computer world. Continuous-time and digital control systems are introduced in Sect. 3.6, while Sect. 3.7 provides additional information on the notation used in subsequent chapters. Finally, Sect. 3.8 provides a short summary of the chapter. For a more comprehensive description of the topics treated in this chapter, see [4].

## 3.2 Physical Systems

In this section, we will describe the most important elementary models in various domains. Each model is described by an icon and an equation.

### 3.2.1 Mechanical Systems (Translations)

The basic equations in the mechanical domain are given by Newton's law, Hooke's law and by the relation between force, velocity and (viscous) friction.

*Newton's law* describes the relation between a force,  $F$  and the acceleration,  $a$  (the derivative of the velocity,  $v$ ) and a *mass*  $m$ :



$$F = ma = m \frac{dv}{dt} \quad (3.1)$$

*Hooke's law* describes the relation between a force,  $F$ , and the extension,  $x$ , of a *spring* with *spring constant*,  $k$ , or *compliance*,  $c$ :



$$F = kx \text{ or } F = \frac{1}{c}x \quad (3.2)$$


In the case of a *damper* with *viscous friction*, the damping constant,  $d$ , determines the relation between the velocity,  $v$ , and the force,  $F$ :

 or 

$$F = dv \quad (3.3)$$


### 3.2.2 Mechanical Systems (Rotations)

The equations in the translation domain have their equivalents in the rotation domain. *Newton's law* for the relation between a torque,  $T$ , and the angular acceleration,  $\alpha$  (the derivative of the angular velocity,  $\omega$ ) and the *inertia*,  $J$ , is



$$T = J\alpha = J \frac{d\omega}{dt} \quad (3.4)$$

*Hooke's law* for the relation between a torque,  $T$ , and the extension,  $\varphi$ , of a *spring* with *spring constant*,  $k$ , or *compliance*,  $c$ , is



$$T = k\varphi \text{ or } T = \frac{1}{c}\varphi \quad (3.5)$$

In the case of a *damper* with *viscous friction*, the damping constant,  $d$ , determines the relation between the angular velocity,  $\omega$ , and the torque,  $T$ :

 or 

$$T = d\omega \quad (3.6)$$


### 3.2.3 Electrical Systems

The basic equations in the electrical domain are given by equations for inductance, capacitance and resistance. The relation between the electrical current,  $i$ , the voltage,  $u$ , and the *self-inductance*,  $L$ , of a coil is given by



$$u = L \frac{di}{dt} \quad (3.7)$$

The relation between the electrical current,  $i$ , the voltage,  $u$ , and the capacitance,  $C$ , of a *capacitor* is given by



$$i = C \frac{du}{dt} \quad (3.8)$$

*Ohm's law* describes the relation between the current,  $i$ , the *resistance*,  $R$ , and the voltage,  $u$ , of a *resistor*:



$$u = Ri \quad (3.9)$$

### 3.2.4 Hydraulic Systems

The basic equations in the hydraulic domain give the relation between pressure and flow. The relation between the fluid flow,  $\phi$  and the volume,  $V$ , for a *fluid storage* (e.g. a cylindrical tank with water) is given by

$$V = \int \phi dt \quad (3.10)$$

or, when we take the area of the tank and the properties of the fluid into account:

$$p = \frac{1}{c} \int \phi dt \quad (3.11)$$

where  $p$  is the pressure at the bottom of the tank and  $c$  is a constant, which determines the “capacity” of the tank.

For a restriction we find, similar to Ohm's law:

$$p = r\phi \quad (3.12)$$

where  $r$  is the *hydraulic resistance*, e.g. of a tap.

### 3.2.5 Equations in Integral Form

At first sight there is no relation at all between all these equations. However, we can rewrite these commonly used forms and bring the equations into a common framework. First of all we realise ourselves that for simulations in a computer integral forms (or integral causality) are preferred above derivative forms. In addition, we note that in a mechanical system the following relations hold:

$$v = \frac{dx}{dt} \text{ and } a = \frac{dv}{dt} \quad (3.13)$$

or in integral form:

$$x = \int v dt \text{ and } v = \int a dt \quad (3.14)$$

With (3.14) we can rewrite Newton's law (3.1) and Hooke's law (3.2):

$$F = ma \rightarrow F = m \frac{dv}{dt} \text{ or in integral form: } v = \frac{1}{m} \int F dt \quad (3.15)$$

and

$$F = kx \rightarrow \text{ or in integral form: } F = k \int v dt \text{ or } F = \frac{1}{c} \int v dt \quad (3.16)$$

In a similar way, we can rewrite the equations for the electrical domain (3.7) and (3.8):

$$u = L \frac{di}{dt} \rightarrow i = \frac{1}{L} \int u dt \quad (3.17)$$

and

$$i = C \frac{du}{dt} \rightarrow u = \frac{1}{C} \int i dt \quad (3.18)$$

Because the equations for the electrical and hydraulic resistance and for the mechanical friction are algebraic equations, there is no preference for one particular form. Therefore, we can write Eqs. (3.3), (3.9) and (3.12) in any of the forms below:

$$F = dvv = F/dF - dv = 0 \quad (3.19)$$


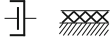

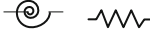








$$u = Rii = u/Ru - Ri = 0 \quad (3.20)$$

$$p = r\phi\phi = p/rp - r\phi = 0 \quad (3.21)$$

As a result of rewriting the equations, they can now be combined into a common framework as in Table 3.1. Table 3.1 suggests that we can consider the variables  $F$ ,  $u$  and  $p$  as analogue variables. Similarly the variables  $v$ ,  $i$  and  $\phi$  can be considered analogue variables. We could also have made the choice to consider  $F$  and  $i$  as analogue variables, but the so-called force-voltage analogy is most common.

The equation for the hydraulic inertia in the lower-right corner of Table 3.1 was not mentioned before. It follows in fact automatically from the symmetry in the rest of the table. This equation describes the inertia of a moving fluid mass. It is

**Table 3.1** Equations and icons of elementary submodels in various domains

Domain	Behaviours		
Mechanical (translation)	Spring:  $F = \frac{1}{c} \int v dt$	Damper/friction:  $F - dv = 0$	Mass:  $v = \frac{1}{m} \int F dt$
Mechanical (rotation)	Spring:  $T = \frac{1}{c} \int \omega dt$	Damper/friction:  $T - d\omega = 0$	Inertia:  $\omega = \frac{1}{J} \int T dt$
Electrical	Capacitor:  $u = \frac{1}{C} \int i dt$	Resistor:  $u - Ri = 0$	Inductance:  $i = \frac{1}{L} \int u dt$
Hydraulical	Tank:  $p = \frac{1}{c} \int \phi dt$	Resistance:  $p - r\phi = 0$	Inertia:  $\phi = \frac{1}{I} \int p dt$

responsible for a phenomenon known as water hammer: a sudden change in the flow of fluid, e.g. by closing a tap, will lead to large forces on the piping and may produce a sound as if the pipes were hit by a hammer.

### 3.2.6 Power
















Models of physical systems, like the system of Fig. 3.1, are often not restricted to a single domain. But there is a variable that all domains have in common. That is *power*. In the domains given in Table 3.1, power is the product of respectively  $F$  and  $v$ ,  $u$  and  $i$ , and  $p$  and  $\phi$ :

$$P = Fv \quad P = ui \quad P = p\phi \tag{3.22}$$

The fact that power is common in different domains enables us to couple these domains. Take for instance an elementary electrical motor, i.e. a motor, which converts electrical power into mechanical power, without any losses. At the electrical side power (in [kWh]) is the product of voltage  $u$  and current  $i$ . At the mechanical side, using SI units, power is expressed again in [kWh], now the product of the force  $F$  and the velocity  $v$ . To model an electrical motor, we need an element that relates the electrical variables  $u$  and  $i$  to the mechanical variables  $F$  and  $v$ , under the condition that

$$ui = P_{el} = P_{mech} = Fv \tag{3.23}$$

**Table 3.2** Summary of the basic icons in various domains

Mechanical translation				
	spring	damper	mass	fixed world
				
		friction		
Mechanical rotation				
	spring	damper	moment of inertia	fixed world
				
	spring	friction		
Electrical				
	Capacitor	Resistor	Inductance	Ground

In Sect. 3.4.2, we will see how power can be used as the basis for modelling of systems in various physical domains and introduce this elementary motor model.

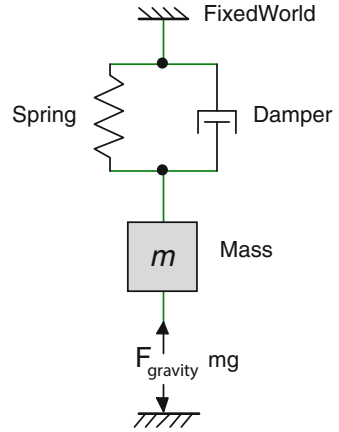
### 3.3 Icons and Iconic Diagrams

A causal relation diagram, like the one in Fig. 3.2, identifies the various subsystems, but is still far from an exact description of the dynamics of the system. The *elementary submodels* or *elements*, represented by the icons and formulas of Table 3.1, can be used to describe real *components* and complete systems. The various elements are summarised in Table 3.2. The icons of the references in the form of the fixed world and the electrical ground are given as well.

By connecting these elements (icons), we can make a model. We call such a model, consisting of interconnected ideal elements, an *IPM* or an *Iconic Diagram*. Because each of these elements is well described by the equations in Table 3.1, we can derive the equation(s) that describe the behaviour of the complete system. As an example of a mechanical system, we consider the *mass-spring-damper system* of Fig. 3.3.

This system can be seen as a mass that is connected to the ceiling by a spring-damper combination. Each of the icons represents an elementary model and is thus described by one of the equations of Table 3.1. The fixed world has a velocity equal to zero. The mass, as well as the lower ends of spring and damper, has a velocity  $v$ . When the system is at rest, the velocity and acceleration are both equal to zero and

**Fig. 3.3** Mass-spring-damper system



in this *static equilibrium* state the gravity force will be compensated by the force of the spring. When the system is not at rest, the sum of all forces is still equal to zero, but  $a$ ,  $v$  and  $x$  vary constantly. This is called a *dynamic equilibrium*. The equation which describes this “equilibrium” is called the equation of motion. In order to find this equation, we consider the forces related to the elements:

$$\begin{aligned}
 F_{\text{spring}} &= \frac{1}{c} \int v dt = \frac{1}{c} x \\
 F_{\text{damper}} &= dv \\
 v_{\text{mass}} &= \frac{1}{m} \int F_{\text{mass}} dt \text{ or } a = \frac{1}{m} F_{\text{mass}} \\
 F_{\text{gravity}} &= F_g = mg
 \end{aligned}
 \tag{3.24}$$

When we define a downward velocity or force as positive, then a positive velocity or position of the mass results in counteracting forces of the spring and the damper. This implies that in the equilibrium state holds

$$F_{\text{gravity}} = F_{\text{spring}} + F_{\text{damper}} + F_{\text{mass}} \tag{3.25}$$

or

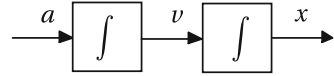
$$F_{\text{mass}} = F_{\text{gravity}} - F_{\text{spring}} - F_{\text{damper}} \tag{3.26}$$

This can be written as

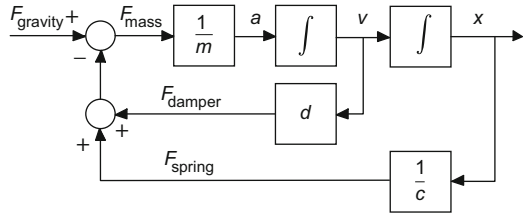
$$F_{\text{mass}} = F_{\text{gravity}} - \frac{1}{c} x - dv \tag{3.27}$$

We can make a graphical representation of this system in the form of a block diagram that can be used as a representation in the graphical editor of 20-sim.

**Fig. 3.4** Relation between  $a$ ,  $v$  and  $x$ , represented by two integrators



**Fig. 3.5** Second-order mass-spring-damper system represented by a block diagram



We start making the block diagram by drawing two integrator blocks, which give the relation between  $x = \int v dt$  (or  $v = \frac{dx}{dt}$ ) and  $v = \int a dt$  or ( $a = \frac{dv}{dt}$ ) (Fig. 3.4).

Adding the relations from (3.24) and (3.27) yields a block diagram of this system (Fig. 3.5).

Another well-known representation of a dynamic system is a differential equation. The differential equation of this system can be derived from the block diagram or by rewriting (3.27):

$$\begin{aligned}
 F_{\text{mass}} = ma &= F_{\text{gravity}} - \frac{1}{c}x - d v \\
 m \frac{d^2 x}{dt^2} &= F_{\text{gravity}} - \frac{1}{c}x - d \frac{dx}{dt}
 \end{aligned}
 \tag{3.28}$$

or

$$m \frac{d^2 x}{dt^2} + d \frac{dx}{dt} + \frac{1}{c}x = F_{\text{gravity}}
 \tag{3.29}$$

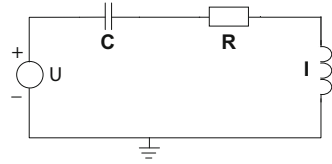
Note: In the case that the derivatives of  $x$  ( $v = \frac{dx}{dt}$  and  $a = \frac{d^2 x}{dt^2}$ ) are equal to zero, (3.25) describes the static equilibrium.

### 3.4 A Domain-Independent Description: Bond Graphs

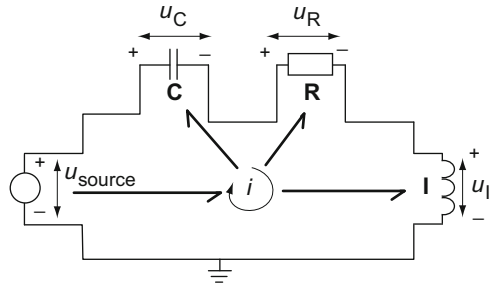
When systems become more complex, deriving equations (and block diagrams, which are just a graphical representation of the equations) requires more and more effort. Also, when only minor changes are made in the system, it may be required to derive a complete new set of equations. 20-sim is able to generate the equations automatically, either from a block diagram (like Fig. 3.5) or from an iconic diagram (like the one in Fig. 3.3). How this is done will be explained here. In 20-sim, all models of physical systems are based on bond graphs even when iconic diagrams are used. In that case, the underlying bond graphs are hidden for the user. It is



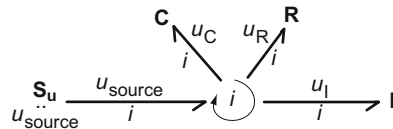
**Fig. 3.6** Electrical circuit with a voltage source



**Fig. 3.7** Electrical circuit: the *half arrows* indicate positive power flow



**Fig. 3.8** Electrical circuit: the *half arrows* indicate positive power flow



good to understand some basic properties of bond graphs because this helps to make *competent models* of systems that extend over various physical domains. Bond graphs are in fact domain-independent representations based on the analogies mentioned in Table 3.1. We will demonstrate this with an example of an electrical and a mechanical system. First we consider the electrical circuit of Fig. 3.6.

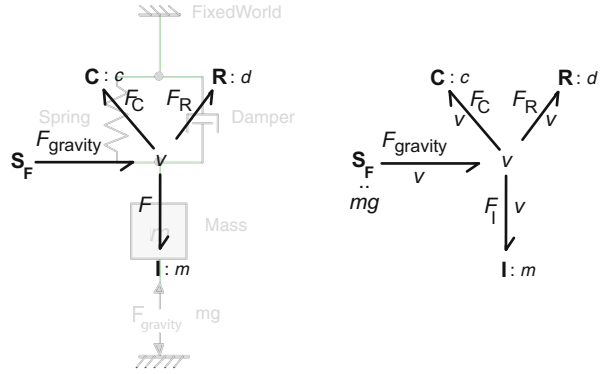
In this circuit, all elements share the same current. And when we add orientations to the voltages, the sum of all these voltages is equal to zero. This is made clear by redrawing Fig. 3.6 in Fig. 3.7. In Fig. 3.7 also an elementary bond graph is drawn. The half arrows are so-called *power bonds*. These bonds show how the power from the source ( $P_{\text{source}} = u_{\text{source}}i$ ) is distributed over the **C**, **R** and **I**-elements:

$$P_{\text{source}} = P_C + P_R + P_I = (U_C + U_R + U_I)i = U_{\text{source}}i \quad (3.30)$$

The diagram with the half arrows of Fig. 3.7 has been drawn separately in Fig. 3.8. Each bond represents power, the product of  $u$  and  $i$ . This is indicated by adding these two variables to each bond.

We can do a similar exercise with the iconic model of the mass-spring-damper system of Fig. 3.3. In this iconic diagram, we see that all the mechanical elements in Fig. 3.3 share the same velocity at the point where the mass, spring and damper are connected to each other. The sum of the forces in this point with common velocity is equal to zero. The power related to the gravity force is in this case distributed over the spring, damper and mass. If we indicate the spring as a **C**-element, the damper as an **R**-element and the mass as an **I**-element, this yields

**Fig. 3.9** Mass-spring-damper system



**Table 3.3** Effort and flow variables in various domains

Domain	Effort ( $e$ )	Flow ( $f$ )	Power ( $P = ef$ )
Mechanical (translation)	Force ( $F$ )	Velocity ( $v$ )	$P = Fv$
Mechanical (rotation)	Torque ( $T$ )	Angular velocity ( $\omega$ )	$P = T\omega$
Electrical	Voltage ( $u$ )	Current ( $i$ )	$P = ui$

$$P_{gravity} = P_C + P_R + P_I = (F_C + F_R + F_I)v = F_{gravity}v \tag{3.31}$$

This results in Fig. 3.9, which shows almost the same graph as Fig. 3.7.

In order to emphasise the common properties of the systems, we make a slightly more abstract representation. We saw already in Table 3.1 that when we consider  $F$ ,  $T$ ,  $u$  and  $p$  as analogue variables, we get a set of similar equations. Therefore, we generalise these variables the so-called *effort variables* ( $e$ ). Similarly we generalise the variables  $v$ ,  $\omega$ ,  $i$  and  $\phi$  to *flow variables* ( $f$ ). This is summarised in Table 3.3.

This leads to the generalised equations for the **C**, **I** and **R**-elements:

$$\begin{aligned}
 e &= \frac{1}{C} \int f dt \\
 f &= \frac{1}{I} \int e dt \\
 e - Rf &= 0
 \end{aligned}
 \tag{3.32}$$

In the bond graph, we could represent the variables  $i$  and  $v$  by the flow variable  $f$  and the effort variables  $u$  and  $F$  by the effort variable  $e$ . At the *junctions*, where the bonds come together we use the symbol **1** for a common flow and the symbol **0** for a common effort. The **1**-junction not only represents the current ( $i$ ) or the velocity ( $v$ ). It also indicates the point where the power from the effort source (**Se**) is distributed over the **C**, **I** and **R** elements:

$$P_{Se} = P_C + P_I + P_R \tag{3.33}$$

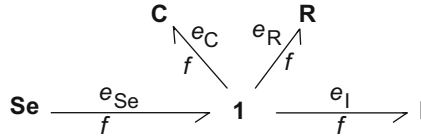


Fig. 3.10 Domain-independent bond graph representing the iconic diagrams of Figs. 3.3 and 3.6

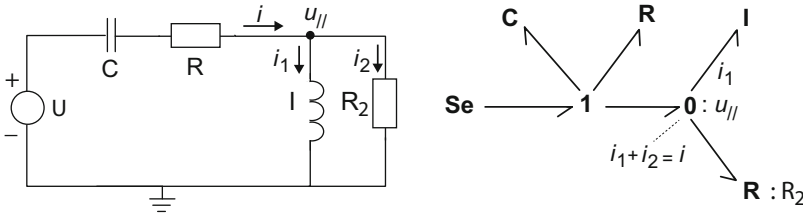


Fig. 3.11 Bond graph with a 0-junction (with two electrical elements in parallel)

or because all elements share the same flow:

$$e_{Se}f = e_C f + e_I f + e_R f \rightarrow e_{Se} - e_C - e_I - e_R = 0 \tag{3.34}$$

This leads to the domain-independent bond graph of Fig. 3.10.

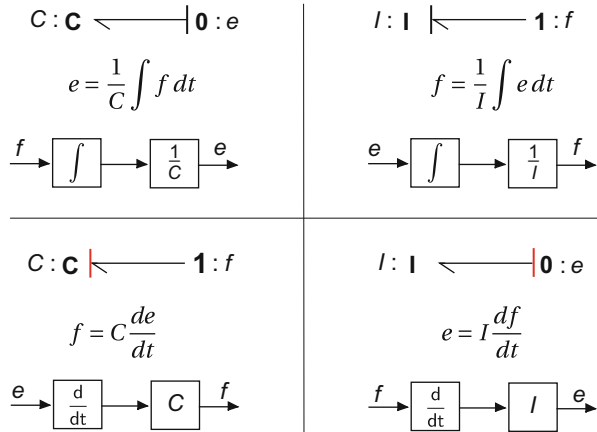
Thus at a **1**-junction the sum of the efforts (taking into account the direction of the half-arrows) is equal to zero. In the dual case, a **0**-junction not only represents an effort but also that the sum of all the flows (taking into account the direction of the half-arrows) is equal to zero.

An example of an electrical system with a **0**-junction is given in Fig. 3.11. The parallel circuit of the inductance (I) and the resistor  $R_2$  shares the same voltage  $u$  and it is thus represented by a **0**-junction.

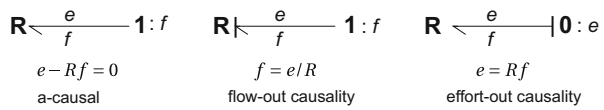
The iconic diagrams of Figs. 3.3, 3.6 and the bond graph representation in Fig. 3.10 represent the same information with respect to the dynamics of the system. In order to find the equations for these systems, the bond graph can be enhanced with the so-called *causal strokes*. With these causal strokes the bond graph can be directly translated into a set of equations, required to simulate the system. We have already seen that for simulations it is preferred to compute the equations for the different elements in integral form. In other words, we prefer integral causality. This implies that according to Eq. (3.32), for a **C**-element we prefer to compute effort as output with flow as input. This can be expressed as: **C**-elements have *preferred effort-out causality*. Similarly, **I**-elements have *preferred flow out causality*.

Causality can be indicated in the bond graph. Elements with effort-out causality have a so-called causal stroke at the end of the bond near the junction. Elements with flow-out causality have the causal stroke at the end of the bond away from the junction. A bond with a causal stroke can also be represented as an equation or a block diagram (Fig. 3.12).

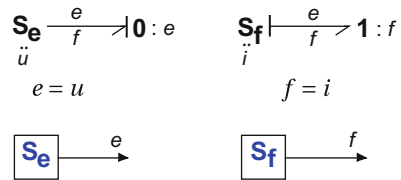
**Fig. 3.12** Preferred and non-preferred causality of **C**- and **I**-elements: bond graphs, equations and block diagrams



**Fig. 3.13** **R**-elements have indifferent causality



**Fig. 3.14** Fixed causality of effort and flow sources



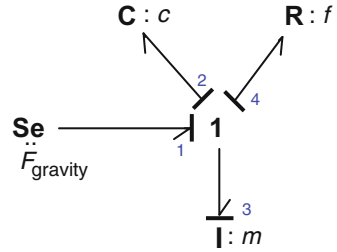
The bond in the upper-right corner of Fig. 3.12 shows that the flow of the **1**-junction is determined by integrating the effort of the **I**-element (according to Eq. (3.34)  $e_I$ : the sum of the efforts of all the other elements connected to this junction) and dividing this by  $I$ . In this case, we say that the **I**-element has the preferred flow-out causality. When the causal stroke is at the other end (lower-right corner of Fig. 3.12), the **I**-element has the non-preferred effort-out causality. In that case, the effort must be computed as the derivative of the flow.

Because **R**-elements are described by a static relation, there is no preference for effort-out or flow-out causality. **R**-elements have thus an indifferent causality (see Fig. 3.13).

Because an ideal voltage source always delivers a voltage (or effort), a voltage source has *fixed effort-out causality*. In the dual case, a current (flow) source has *fixed flow-out causality* (see Fig. 3.14).

In Figs. 3.12 and 3.13, we connected the bonds with effort-out causality always to a **0**-junction and the bonds with flow-out causality always to a **1**-junction. When there are more bonds connected to a junction, the junction can be either a **0**- or a **1**-junction.

**Fig. 3.15** Causal bond graph of the mass-spring-damper system



Junctions have a *causal constraint*. A **1**-junction represents a flow. This flow can only be determined by *one* of the elements connected to the junction. This implies that only one of the bonds connected to this junction has flow-out causality. The other bonds must have effort-out causality. In the dual case, a **0**-junction represents an effort. This effort can only be determined by *one* of the elements connected to the junction. This implies that only one of the bonds connected to a **0**-junction has effort-out causality.

### 3.4.1 Example

With these rules we can make the bond graph of Fig. 3.9 causal. We start with the fixed causality of the gravity force, represented by the effort source **Se** (1). Because there are no other elements with fixed causality, we assign the preferred causality to the **C**-element (2). Then we assign the preferred causality to the **I**-element (3). Of course we can interchange the order of step 2 and 3. The last (**R**)-element has indifferent causality, but the causal constraint of the **1**-junction tells us that there can be only one bond with effort out causality at the **1**-junction. This implies that the **R**-element must have effort-out causality (4) (Fig. 3.15).

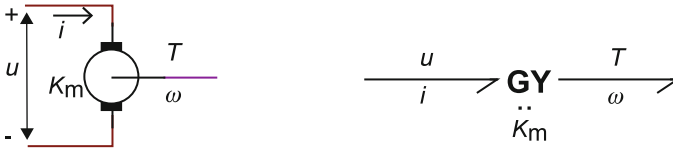
A causal bond graph can directly be converted into equations. Remember that in this mechanical system the flow of the **1**-junction stands for a velocity and the effort for a force. The set of equations ((3.35) and (3.36)) is equal to Eqs. (3.24)–(3.26):

The equation for the **1**-junction yields

$$e_{Se} - e_C - e_R - e_I = 0 \rightarrow F_{\text{mass}} = F_{\text{gravity}} - F_{\text{spring}} - F_{\text{damper}} \quad (3.35)$$

The equations for each bond yield

$$\begin{aligned} f &= \frac{1}{m} \int e_I dt & \rightarrow & \quad v = \frac{1}{m} \int F_{\text{mass}} dt \\ e_{Se} &= mg & \rightarrow & \quad F_{\text{gravity}} = mg \\ e_C &= \frac{1}{c} \int f dt & \rightarrow & \quad F_{\text{spring}} = \frac{1}{c} \int v dt \\ e_R &= df & \rightarrow & \quad F_{\text{damper}} = dv \end{aligned} \quad (3.36)$$



**Fig. 3.16** Elementary DC motor (*left*) and bond graph representation: gyrator (*right*)

Because the bond graphs of the RLC circuit of Fig. 3.6 and the one of the mass-spring-damper circuit of Fig. 3.3 are identical in terms of effort and flow variables, a similar set of equations is found for both systems.

### 3.4.2 Models in Different Domains

The elements presented so far are all one-port elements. They have one so-called *power port*. When we want to connect two different domains, we need a two-port element, where one port is connected to, for instance, the electrical domain and the other one to the mechanical domain. An example is the system of Fig. 3.1. The electric motor in this system converts electrical energy into mechanical energy and when it is used as a generator also vice versa. An elementary electric motor is described by the two equations at the left-hand side of (3.37) and the iconic diagram of Fig. 3.16. As a bond graph element, the elementary DC motor is represented by a *gyrator* (equations at the right-hand side of Eq. (3.37))

$$\begin{aligned} T &= K_m i & \text{or, more general:} & & e_2 &= n f_1 \\ u &= K_m \omega & & & e_1 &= n f_2 \end{aligned} \tag{3.37}$$

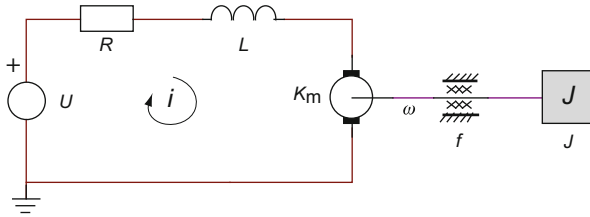
where  $K_m$  is the so-called *motor constant*. Note that these equations imply that the electrical power is equal to the mechanical power:

$$P_{\text{mech}} = T\omega = K_m i \frac{1}{K_m} u = ui = P_{\text{el}} \tag{3.38}$$

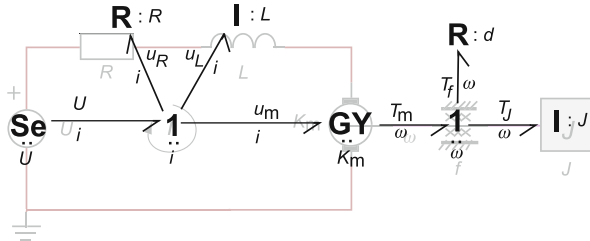
Figure 3.16 represents an *elementary DC motor*. A DC motor as a realistic *component* has additional electrical properties which can be represented by the elements resistance and self-inductance and mechanical properties which can be represented by the elements friction and inertia. This leads to Fig. 3.17.

In a similar way as we did in Figs. 3.7 and 3.9, we can draw the bond graph of Fig. 3.18.

In order to make this bond graph causal, we need to consider the causal constraints of the gyrator. Equation (3.37) implies that if one end of the gyrator has effort-out causality the other end must have effort-out causality as well. Similarly, if one end of the gyrator has flow-out causality, the other end must have flow-out causality as well. This is represented in Fig. 3.19.



**Fig. 3.17** IPM of a DC motor with electrical and mechanical properties connected to a voltage source



**Fig. 3.18** Power flow to the various elements in the DC motor model

$$\begin{array}{ccc}
 \mathbf{1} \left| \begin{array}{c} e_1 \\ f_1 \end{array} \right. \xrightarrow{\text{GY}} \left. \begin{array}{c} e_2 \\ f_2 \end{array} \right| \mathbf{1} & & \mathbf{1} \left| \begin{array}{c} e_1 \\ f_1 \end{array} \right. \xrightarrow{\text{GY}} \left. \begin{array}{c} e_2 \\ f_2 \end{array} \right| \mathbf{1} \\
 e_1 = n f_2 \quad e_2 = n f_1 & & f_1 = \frac{1}{n} e_2 \quad f_2 = \frac{1}{n} e_1
 \end{array}$$

**Fig. 3.19** Causal constraints of a gyrator

In a similar way as in the example of Fig. 3.15, we can now make the bond graph causal. We start again with the fixed effort-out causality of the voltage source (1). In the next step, we assign the preferred flow-out causality to the inductance (2). Because of the causal constraint of the **1**-junction, the other bonds get effort-out causality (3,4). The causal constraint of the gyrator results in effort-out causality at the mechanical side of the gyrator (5). The inertia has preferred flow-out causality (6) and the causal constraint of the mechanical **1**-junction gives effort-out causality to the friction (7) (Fig. 3.20).

From this causal bond graph, equations can easily be derived:

$$\begin{array}{ll}
 \text{left } \mathbf{1}\text{-junction:} & \text{right } \mathbf{1}\text{-junction:} \\
 u_L = u_{\text{source}} - u_R - u_{GY} & T_J = T_{GY} - T_d \\
 i = \frac{1}{L} \int u_L dt & \omega = \frac{1}{J} \int T_J dt \\
 u_R = iR & T_d = d\omega \\
 u_{GY} = K_m \omega & T_{GY} = K_m i
 \end{array} \tag{3.39}$$

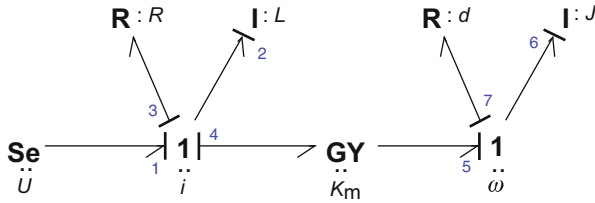


Fig. 3.20 Causal bond graph of a DC motor

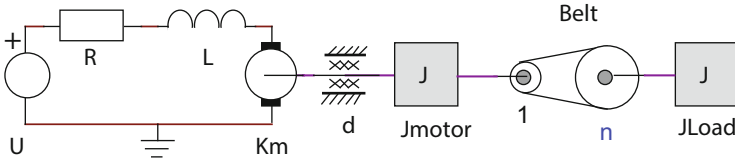


Fig. 3.21 Bond graph of a DC motor connected to a voltage source and a mechanical load via a transmission

$$\begin{array}{ccc}
 0 \text{---} \frac{e_1}{f_1} \text{---} \text{TF} \text{---} \frac{e_2}{f_2} \text{---} 1 & & 1 \text{---} \frac{e_1}{f_1} \text{---} \text{TF} \text{---} \frac{e_2}{f_2} \text{---} 0 \\
 e_1 = ne_2 & f_2 = nf_1 & f_1 = \frac{1}{n}f_2 & e_2 = \frac{1}{n}e_1
 \end{array}$$

Fig. 3.22 Causal constraints of a transformer

However, when using 20-sim, there is no need to derive the equations because both iconic diagrams and bond graphs can be used as input formats for the graphical editor of 20-sim. In addition, 20-sim does the causal analysis automatically. However, the causal analysis is not only useful for deriving equations. It also helps to make a proper model. This will become clear when we extend the model with the inertia of the load. The load is connected to the motor via a long rod and a transmission, a pulley-belt-pulley, as can be seen in the photo of the setup (Fig. 3.1). In order to keep the model simple, we model the rod as a rigid axis and disregard any flexibility in the belt. The transmission is added in the iconic diagram of Fig. 3.21.

In this case, the transmission corresponds to the elementary bond graph element transformer (TF), described by Eq. (3.40) and Fig. 3.22

$$\begin{array}{ccc}
 T_1 = nT_2 & \text{or, more general:} & e_1 = ne_2 \\
 \omega_2 = n\omega_1 & & f_2 = nf_1
 \end{array} \tag{3.40}$$

It follows from the equations that if one end of the transformer has effort-out causality the other end has flow-out causality.

When we add the transmission in the bond graph, we get the graph of Fig. 3.23. When we try to assign the causality to the elements in this graph, we notice that we cannot give the I-element  $J_2$ , the preferred causality. This implies that we can only



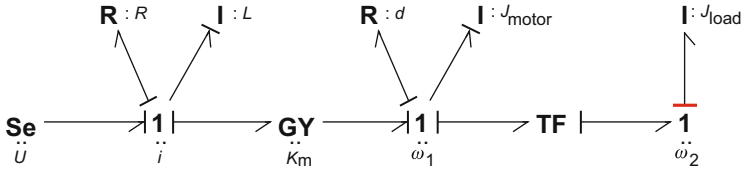


Fig. 3.23 Bond graph of a DC motor connected to a voltage source and a mechanical load via a rigid transmission

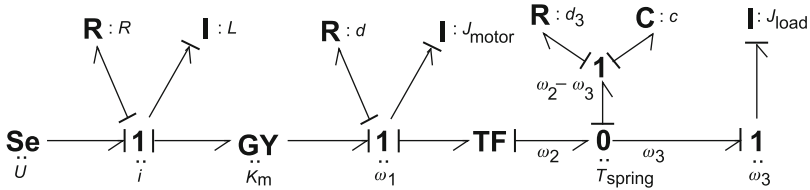


Fig. 3.24 Bond graph of a DC motor with a flexible axis

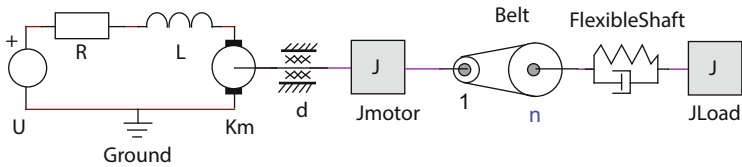


Fig. 3.25 IPM of a DC motor with a flexible axis

simulate the system with a simulation program (such as 20-sim) which is able to compute this element in derivative form. But it is not only a computational problem. Such a so-called *causal conflict* indicates that the model is probably either too simple or too complex. It means in this case that the two inertias are not independent of each other. Because they are rigidly connected they could be considered as one inertia (taking into account the presence of the transformer). This would simplify Fig. 3.23 to Fig. 3.20 with  $J_{total} = J_1 + n^2 J_2$  ( $n < 1$ ).

Another option is to look more closely to the setup and ask ourselves if we did not simplify the model too much. The long plastic rod, which connects the transmission with the load, is certainly not a rigid component. Therefore, it makes sense to model this flexibility as a rotation spring with some damping. The spring and the damper share the same velocity difference  $\Delta\omega = \omega_2 - \omega_3$  and the same torque  $T_{spring}$ . This can be expressed by adding a **1**-junction connected to a **0**-junction. This leads to the bond graph of Fig. 3.24. This bond graph can be translated into the iconic diagram of Fig. 3.25.

In this example, the transformer represented a pulley-belt-pulley which “transforms” a rotation into another rotation. Transformers also describe transformations from the rotation to the translation domain or, in the electrical domain, from one voltage (and current) to another voltage (and current).

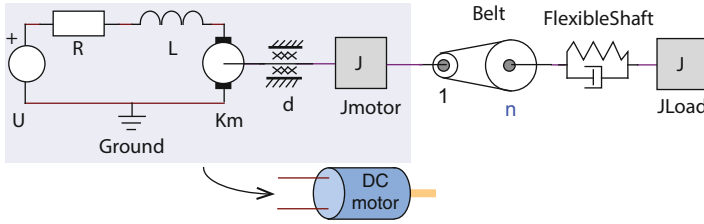


Fig. 3.26 Imploding the elements of the DC motor to a component

### 3.5 Simulating Physical Systems with 20-sim

20-sim supports all kinds of system representations as a basis for the simulation. The equation editor allows for the input of differential equations, either in integral or derivative form. The equations are, if possible, automatically converted into integral equations for the simulator. If this is impossible, the problems with derivative causality are solved symbolically or numerically. The graphical editor supports inputs in the form of block diagrams, iconic diagrams (or IPMs) and bond graphs. Block diagrams are in fact a graphical representation of equations. Iconic diagrams and bond graphs are basically a-causal. In order to simulate the system, 20-sim assigns the causality automatically. Causal conflicts as in Fig. 3.23 are indicated in the bond graph by an orange causal stroke. This indicates that the causal conflict will be solved symbolically or numerically in the simulator, but also alerts the user to think about the complexity of the model. In iconic diagrams, causal conflicts are reported by messages like

```
warning: Solved algebraic variables symbolically
{JMotorDisk\alpha_in}
The model has 0 errors and 1 warnings.
```

Although no action is required for such a warning, it makes sense to reconsider if the model is really adequate.

The different system representations may be used in one single simulation model. A number of *elementary models* can be combined into a *component* by using the *implode* option (Fig. 3.26). This option (and the inverse option *explode*, see Fig. 3.27), supports hierarchical modelling. In addition, the contents of the component can be made more complex or simpler. As long as the interfaces with the rest of the system remain the same, there is no need to change the rest of the model. The imploded model can also be inspected or altered. You can draw your own icon for the component or even use an image as icon.

When we take another look at Fig. 3.1, we see that between the belt and the flexible rod there is a another rather large metal disk, which can be modelled as an inertia. In order to make a model with components as close as possible to the real set-up, we add this inertia to the model (see Fig. 3.28). Because the belt is modelled as rigid, this leads again to a causal conflict. We have seen that 20-sim can solve this conflict symbolically.

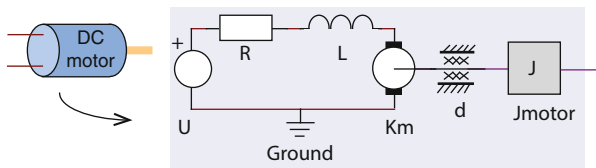


Fig. 3.27 Exploding or inspecting the component

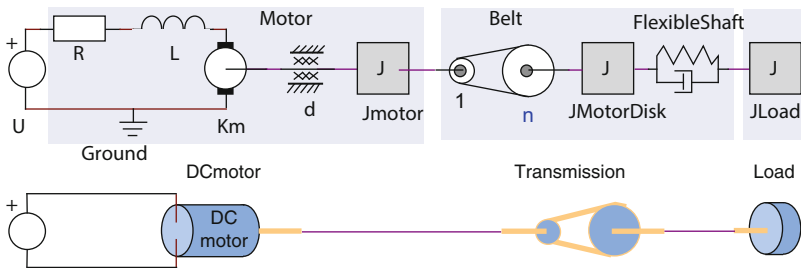
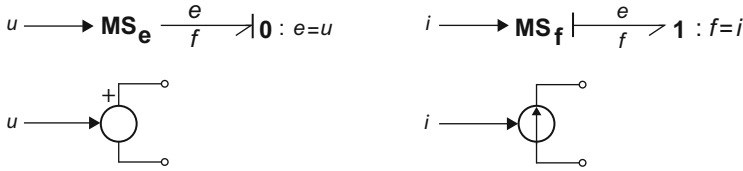


Fig. 3.28 Model including the inertia between the belt and the flexible shaft, imploded into components

### 3.5.1 Sensors and Actuators

The models in the previous sections have all in common that they describe physical systems where power is involved in the coupling between the different elements. Power is always the product of two variables, in general effort and flow. Effort and flow are not only physical variables, they can also be seen as signals, containing *information*. Block diagrams are pure signal based. The blocks are connected by single signals in contrast to the bidirectional bonds, which always represent the two conjugated variables  $e$  and  $f$ . A controller in a computer can always be expressed as a block diagram. In the computer, only the information of the signals is important. Power does not play a role (except for the power needed to run the computer itself). When we want to connect a computer to a physical system, we need devices which convert the signals from the computer to power-based *actuators* and *sensors* which extract relevant information-carrying signals from the physical model.

In the example of the DC motor with a mechanical load, we may want to control the angle or angular velocity of the load. In that case, we need a sensor that measures an angle in the form of a potentiometer or a code disc. For the angular velocity, a code disc or tachometer could be used. Although, for instance, a potentiometer delivers the angle in the form of an electric signal, only the information about the angle is relevant. In this example, the output of the controller will be a signal, representing the voltage needed to give the motor the desired motion. This low-power signal must be amplified by a power amplifier in order to connect it to the physical system. In 20-sim, this can be done by a so-called *modulated voltage*



**Fig. 3.29** Modulated sources. *Top:* Bond graphs. *Bottom:* IPMs

*source.* A modulated voltage source has a signal as input and power with a voltage proportional to the input signal, as output. In a bond graph, this is a modulated effort source or a modulated flow source. Figure 3.29 gives an example of a modulated voltage and a modulated current source.

### 3.5.2 A Brief Introduction to Pulse Width Modulation

In the lower half of Fig. 3.28, we see that the DC motor is connected to a voltage source and the speed of the motor can be controlled by adjusting the voltage applied to it. A common way of implementing a variable voltage/power source is through a method called Pulse Width Modulation (PWM). In PWM, the power to a device, such as a motor, is switched on and off very rapidly and the result is equivalent to connecting the device to a power source with a lower voltage. Figure 3.30 shows three PWM signals, each one exhibiting a different *duty cycle*, where the duty cycle represent the percentage of time the power is switched on. The upper signal has a 10% duty cycle where the power is on for 10% and off for 90% of the time, the middle signal represents a 50% duty cycle and the lower signal a 90% duty cycle. If the voltage in the on state is 5 V, then these PWM signals would be equivalent to 0.5, 2.5 and 4.5 V power sources, respectively. It is important that the length of one *cycle* is short in comparison to the dynamics of the device being driven. Otherwise fluctuations will be observed in the output of the device, such as the speed in the example of a motor. So a lamp dimmer might use a cycle time in the order of  $10^{-2}$  s while a motor controller cycle time might be in the order of  $10^{-3}$  to  $10^{-4}$  s.

It is possible to implement PWM in a co-model and have a DE controller switching the power on and off at the CT side, but this would require the CT and DE models to synchronise at twice the cycle time frequency (once for an on signal and once for an off). This can result in a simulation progressing slower than is necessary, especially if other monitored and controlled variables are updated at much slower frequencies. It is generally the case then that an abstract PWM is modelled by sending a value representing the PWM duty cycle from the controller to the power source, and then using a modulated power source, as shown in Fig. 3.29, in the plant model. This method for modelling PWM can be seen in the line-following robot examples throughout the book.

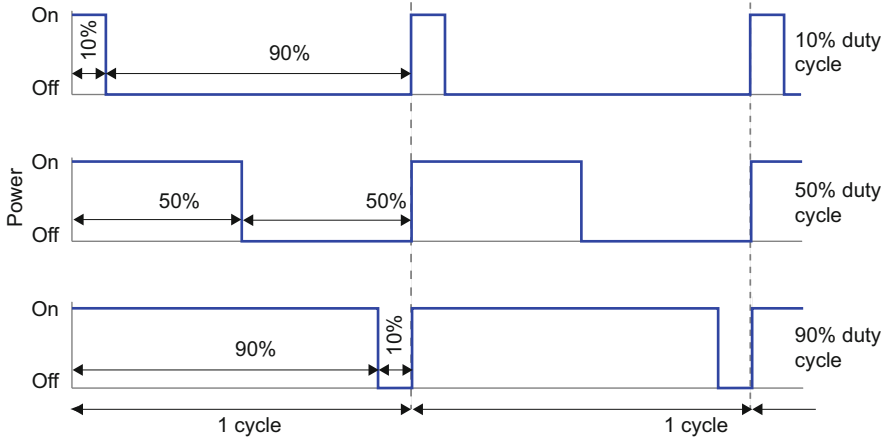


Fig. 3.30 Pulse Width Modulation (PWM) power signals

### 3.6 Control Systems

The torsion bar represents 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. Examples are, for instance, the arm of a robot, a printer head or a wafer stepper used in the semiconductor industry. We can connect the physical system, in this case the torsion bar to a controller by means of one or more sensors and an actuator. This is represented in Fig. 3.31. Here we consider position control. It can be shown that in order to get a stable control system with a relatively simple controller, the position of the motor and its derivative, the motor velocity, should be used in the controller. We replace the voltage source of Fig. 3.28 with an amplifier and a modulated voltage source and use an encoder to measure the (angular) position of the motor (Fig. 3.31).

Most controllers are *feedback controllers*. A feedback controller is part of the control loop. It generates the voltage for the motor to bring or keep the load in the desired position, by making the error between the output of the *setpoint generator* and the measured position of the motor as small as possible (see Fig. 3.31).

The advantage of a feedback controller is that it can compensate for unknown disturbances, uncertainties in the model of the system and parameter variations. A disadvantage is that feedback controllers may become unstable. Therefore, they must be designed carefully. The best performance is obtained when the controller has a high gain, but a high gain may make the system unstable. When a good model of the physical system is available, a *feedforward controller* can increase the performance of the system. The feedforward controller could compensate for the dynamics of the physical system and generate a proper input signal for this system before any error signal can be measured. Of course, feedforward and feedback control can be applied simultaneously (Fig. 3.32).

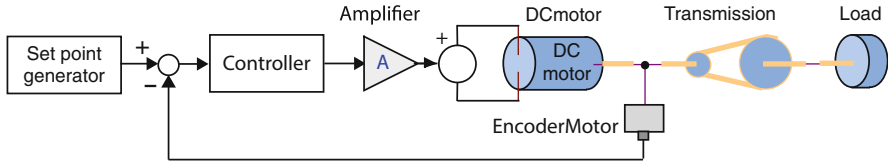


Fig. 3.31 Closing the loop with a sensor, a controller and an actuator

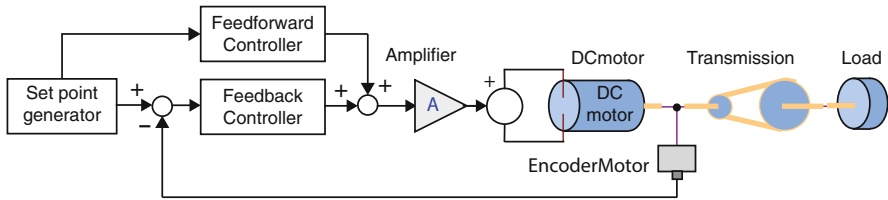


Fig. 3.32 Feedforward and feedback control

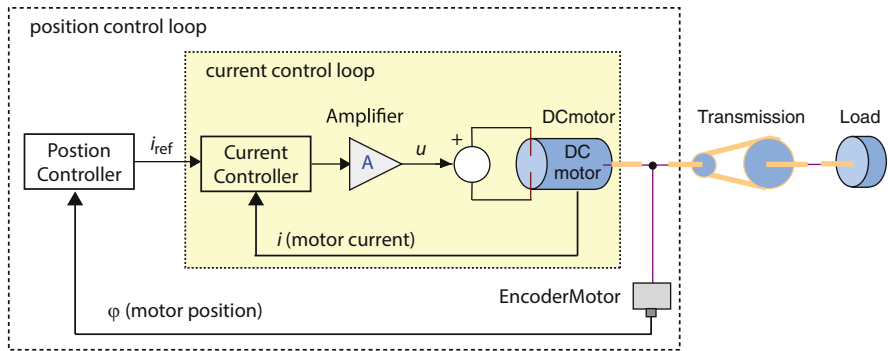


Fig. 3.33 Cascade control

Sometimes controllers are applied in a cascade (Fig. 3.33). In the example of the torsion bar, it would make sense to apply current control instead of voltage control of the motor. This can be realised by adding a *current controller* which makes the current ( $i$ ) out of the voltage source equal to the desired current ( $i_{ref}$ ), generated by the *position controller*. As a result the dynamics of the electrical part of the motor are made so fast, that they can be disregarded. In addition, the current-controlled voltage source becomes in fact a current source. This situation where the current-control loop is placed inside the position-control loop is called *cascade control*. See Fig. 3.33, where the output of the position-controller, the signal  $i_{ref}$ , is the setpoint for the current-control loop.

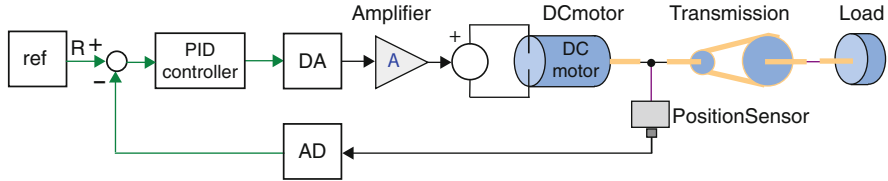


Fig. 3.34 Digital control of a servo system

### 3.6.1 Digital Control Systems

Physical systems are *continuous-time systems*. Computers are *discrete-time systems*. When we want to connect these two we need two additional converters: an Analog-to-Digital (AD-)converter and a Digital-to-Analog (DA-)converter. An AD-converter samples a continuous-time signal and converts it into a discrete-time integer signal with a limited accuracy. DA-converters do the opposite. Typical AD-converters have a resolution of 10–14 bits. DA-converters may have a little bit less bits. They convert the discrete-time integers from the computer into continuous-time signals. A simulation program must be able to handle all these signals simultaneously.

Figure 3.34 shows an example of a continuous-time system coupled to a digital control system. The DA-converter translates the digital controller output into an analogue input of the power amplifier. The power amplifier is connected to the DC motor, which drives the load through the transmission. A position sensor measures the rotation of the motor. The continuous-time sensor signal is transformed into a discrete-time signal by the AD-converter. A *PID-controller* (Proportional plus Integral plus Derivative Controller; the most widely used controller) closes the loop by comparing the sensor signal with the *setpoint* or *reference* and generates a proper output when the sensor signal deviates from the setpoint.

Proper scaling of the signals before the AD- and DA-conversion is important because otherwise the system behaviour is deteriorated due to a rough discretisation or saturation of the signals. At the input of a 12-bit AD-converter with a range of  $\pm 5$  V, there should never be signals larger than 5 V. On the other hand, if the maximum signals are very small, the number of effective bits decreases, leading to a loss of performance. Similar considerations hold for DA-converters.

In 20-sim, all models between an AD- and a DA-converter are automatically simulated as discrete-time elements by calculating them at a fixed rate. This rate is commonly known as the *sampling rate*. In 20-sim, you can set the sampling rate to any desired value. During a simulation, 20-sim will calculate, at every sample instant, the set of discrete-time models, starting from the AD-converter up to the DA-converter. The controller output will be calculated instantaneously or with a delay of a discrete number of samples. As a rule of thumb, the *sampling frequency* should be chosen a factor 10 higher than the bandwidth of the system. Figure 3.35 shows a simulation result where you can clearly distinguish the continuous-time

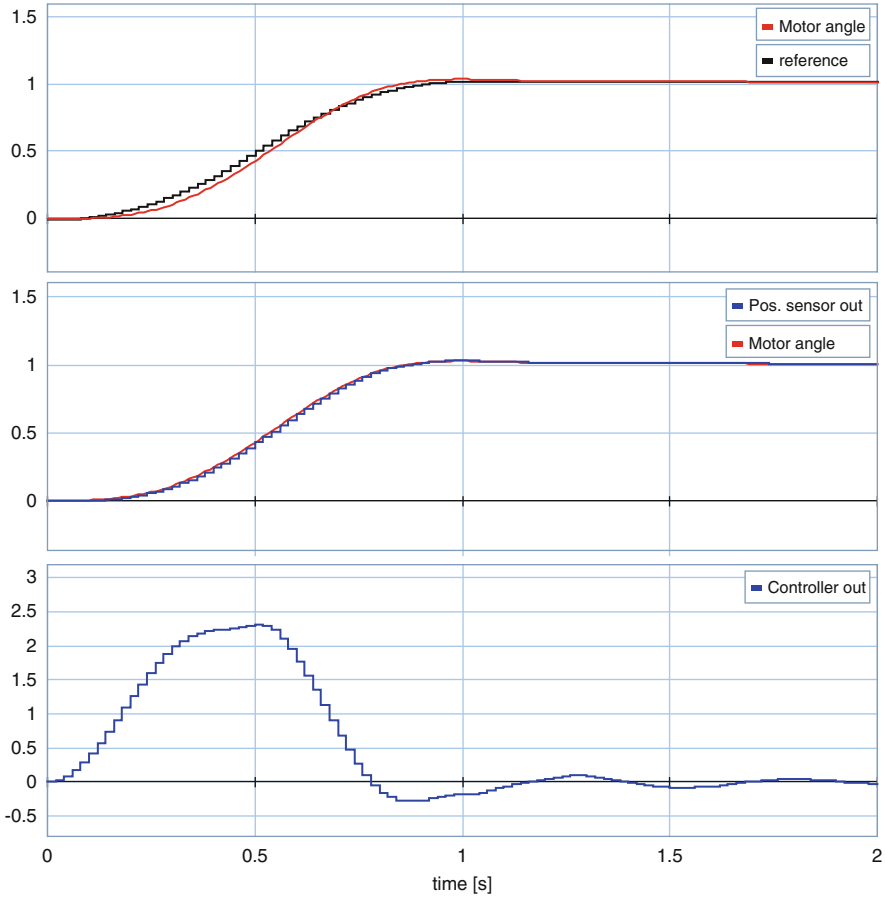


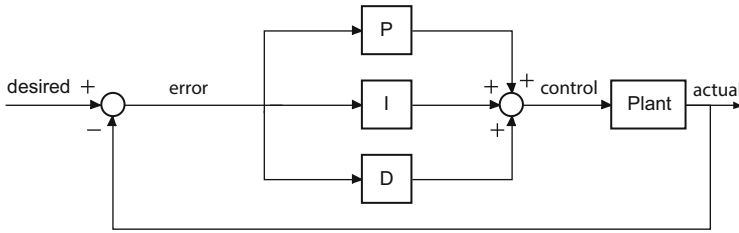
Fig. 3.35 Responses of the PID-controlled torsion bar

signal (the actual motor angle) and discrete-time signals. The top plot shows the discrete-time reference signal and the continuous-time motor angle, the middle plot shows the continuous-time motor angle and the discrete-time version of this signal after AD-conversion. The plot at the bottom shows the discrete-time output of the controller. The sampling frequency in this example is 50 Hz. All calculations in 20-sim are performed using floating-point arithmetic. Fixed-point calculations can be emulated but is seldom used in practice.

### 3.6.2 PID Control

Proportional Integral Derivative (PID) controllers are the most widely used controllers in industrial control systems. A PID controller takes as inputs, the desired





**Fig. 3.36** Feedback PID controller shown as a block diagram

value of some property of the system (setpoint) and the current value of that property (measured variable) and computes a control value (output) that will steer the system towards the setpoint. The output is found by summing three weighted terms, where all terms are based upon the error ( $\epsilon$ ) between the measured variable and the setpoint (Fig. 3.36). The first term is found by multiplying the current error value by a weighting (gain),  $P$ . The second term considers what has happened in the past and is found by integrating the error values and then multiplying this by a weighting (gain),  $I$ . The final term considers the rate of change of errors. It is found by differentiating the error and multiplying this by a weighting (gain),  $D$ . As a result the control signal,  $u$  is given by

$$u = P\epsilon + I \int (\epsilon dt) + D \frac{d\epsilon}{dt} \quad (3.41)$$

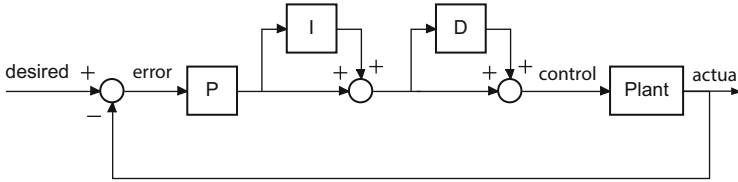
Of course, in digital control systems, a discrete version of this formula must be used:

$$u(kT) = P\epsilon(kT) + I \sum_{n=0}^k \epsilon(nT) + D \frac{\epsilon(kT) - \epsilon((k-1)T)}{T} \quad (3.42)$$

The integral term of the PID controller makes it possible that the controller output is unequal to zero even when the error and its rate of change are both equal to zero. Therefore, it can, among others, account for external disturbances on the system that would prevent it from reaching the desired setpoint value. Likewise, the differentiating term, by considering the rate of change of the errors, may reduce the chance of overshooting the setpoint. It is not always necessary to use all three terms and so  $P$ ,  $I$ ,  $PI$  and  $PD$  control are all potential strategies that may be used.

Practical implementations of PID for digital systems mostly use the “series” form (Fig. 3.37) because tuning, that is, finding the proper PID parameters, is easier for this form.

The  $I$  parameter is mostly replaced by its reciprocal value  $1/\tau_i$  and  $D$  by  $\tau_d$ .  $1/\tau_i$  is called the integral time, it indicates how much time is needed to create an output that is equal to the error. The smaller  $1/\tau_i$ , the quicker the response of the controller.  $\tau_d$  is called the derivative time. It determines how forceful the controller



**Fig. 3.37** Practical implementation of a PID controller shown as a block diagram

reacts on sudden changes in the setpoint or in the external disturbances and may prevent overshoot because it takes the rate of change of the error into account.

The mentioned influences of the controller parameters  $I$  and  $D$  are true for many systems, but certainly not for all systems. Therefore, proper tuning of a PID controller can only be done in combination with (a model of) the physical system.

### 3.6.3 DE Systems

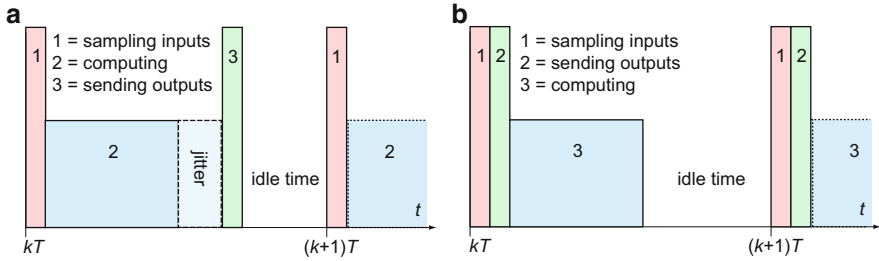
Digital control systems will run on a processor with some kind of operating system. The operating system is required to load the controller code and execute it at a specific rate. Most operating systems are not able to guarantee a proper real-time behaviour. Real-time Linux operating systems are closest to proper real-time control.

The “real-time performance” of a teller machine means in fact *fast enough*. This type of real-time behaviour is called *soft real time*. From the perspective of control engineering soft real time is not enough. Computers used for control must be able to communicate with the continuous-time process at equidistant time intervals, as will be explained in the next section. Furthermore, this communication must be fast enough with respect to the dynamics of continuous-time system. The reason for this is that otherwise there may be stability and accuracy issues. We call this behaviour, where the computer interacts with the process fast enough and at fixed time intervals, *hard real time*.

### 3.6.4 Sampling

A *hard real-time* computer observes the environment at fixed time intervals, uses these observations to perform computations and finally delivers its results to the environment. The most natural way to this is as indicated in Fig. 3.38a:

1. sample the data from the process (this yields the input signal for the computer),
2. perform the computations, and
3. send the computation results to the process (the output signal of the computer is the input signal for the process).



**Fig. 3.38** Timing in a real-time control system. *Left*: Sending outputs after completion of the computations. *Right*: Sending outputs in the next sampling interval

Both the input and output of data should take place at fixed moments in time. Because the duration of the computations needed for computing the output signal varies, this order of the different tasks causes *jitter*, that is, a variation in the duration of the computations in task 2 and thus a variation in the distance between the tasks 1 and 3 and between the distances of the tasks 3 in different sampling intervals. Therefore, this order is only acceptable when the jitter is small.

An alternative to perform this theoretically correct is the following (see Fig. 3.38b):

1. sample at  $t = kT$  the data from the process
2. send the computation results of the former sample, at  $t = (k-1)T$ , to the process
3. perform the computations: the input(s) measured at  $t = kT$  are used to compute the outputs for  $t = (k+1)T$ .

In this case, variations in the duration of the computations do not affect the sampling process. There must always be some idle time because otherwise it cannot be guaranteed that the computations are ready on time. If there is not enough time for the necessary computations, unpredictable results may occur. The alternative order of the different tasks gives the best guarantee that the AD- and DA-conversions take place at equidistant time intervals. In addition, in order to guarantee proper equidistant sampling, special measures can be taken at the hardware level of the interface, for example, by adding an independent hardware clock implemented in an FPGA.

The solution of Fig. 3.38b introduces one sample delay between the sampling of the inputs (task 1 at  $t = kT$ ) and the sending of the outputs (task 2 at  $t = (k+1)T$ ). In all cases, the use of a digital computer in a control loop leads to some time delay. Time delays are harmful to the stability and therefore digital control systems are in general less stable than their continuous-time counterparts. By choosing a small enough sampling interval, the effects of the sampling on the behaviour of the controlled system and its stability can be minimised.

Because the computer only sees the outside world through the sampling process, it only sees values at discrete moments in time:  $t = T, 2T, 3T, \dots, kT$ . For the computer, the outside world is a discrete world. This has as a direct consequence

that the controller should be designed for the discrete system. However, the process itself remains in most cases a continuous-time process, where the variables may change during the sampling intervals.

Working with a fixed sample-rate and floating-point arithmetic will allow control engineers to design stable and robust controllers. This is, however, always an abstraction of reality. Digital control systems

- use fixed-point arithmetic in many cases,
- do not exactly run at a fixed sample rate,
- may be event based,
- have a more complex architecture in practice, and
- introduce a delay equal to the sampling interval in the control loop.

### 3.6.5 Events

Events are actions initiated outside the scope of a program that have to be handled synchronous with the program flow. Two types of events may have a direct influence on the execution time of the controller code.

Events that affect the operating system may force the operating system to temporarily halt the execution of the controller code. *Real-time operating systems* can guarantee a maximum time delay for handling events. Generally operating systems cannot guarantee such a maximum delay. The operating system therefore has a crucial influence on timing of the controller execution.

Events that affect the controller itself may directly influence the execution of the controller. Consider, for example, the controller for a valve in a water tank. In normal operation, the valve should allow a fixed flow of water into the tank. When the water level reaches a maximum, an event is triggered, forcing the controller to close the valve. Depending on the controller architecture, events may be handled immediately or at the next sample time.

Events like the water reaching a maximum level are called *state events*. State events are events that are not known in advance. Events that are known in advance are called *time events*. 20-sim is perfectly capable of generating events. The program can detect state events at the exact time when they occur, even if it is between sampling times. However, 20-sim is not able to simulate controllers that immediately respond to events. In 20-sim, the discrete-time controller is always coupled to the continuous-time process by means of AD- and DA-convertors, which are only active at fixed sampling intervals. If state events should be simulated as well, this could be done by selecting a small sampling interval in the main control loop or by adding an extra discrete-time loop with a (very) small sampling interval.

Figure 3.39 shows a state event. The maximum value is reached in between the sampling time  $3T$  and  $4T$ . It is exactly determined by additional computations in 20-sim.

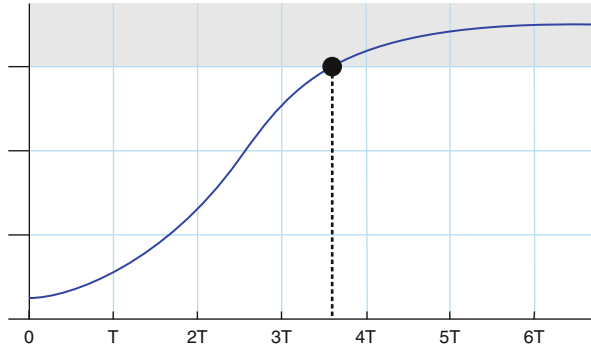


Fig. 3.39 State event: detecting when the level reaches a maximum

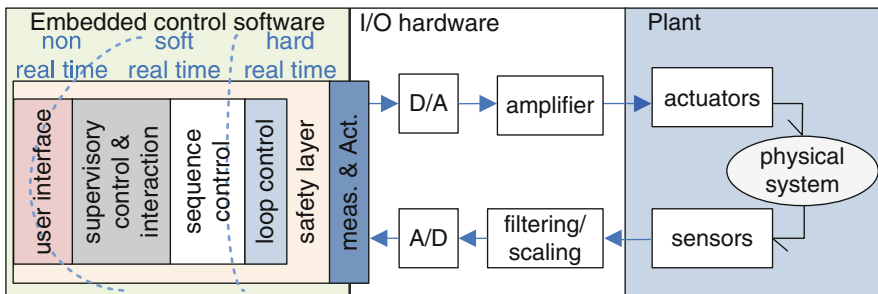
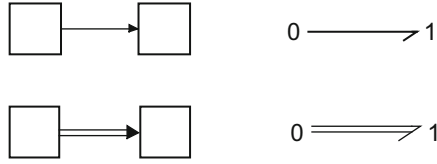


Fig. 3.40 A general architecture for controllers [17]

### 3.6.6 Controller Architecture

The controllers described so far close the loop between the sensor measurement and actuator output. These loop controllers have to ensure that a system will follow a given setpoint, while providing stability and robustness. In general, controllers will be more complex (see Fig. 3.40) and contain several layers.

At the lowest level, we find the loop controller, which translates measured sensor inputs into actuator outputs, based on a given setpoint. In practice, multiple loop controllers may be used, for example, for safe operation, normal operation, starting up, etc. A sequence controller handles the generation of the setpoints and the type of loop controller that is used. Sequence controllers may vary from generators of a simple set of points to complex state machines generating a sequence of interconnected profiles for varying loop controllers on multiple actuator systems. A supervisory controller is responsible for the correct operation of one or more sequence controllers and will handle communication with the user interface and the network. To ensure a safe operation of the system, most controllers will have an independently operating safety layer.



**Fig. 3.41** Examples of connecting arrows sharing: a single variable (*top row*); multiple variables (*bottom row*); bond graphs (*right column*); non-bond graphs (*left column*)

The loop controller has to run at a fixed sample rate to ensure stability. The loop controller code therefore has to be executed with hard real-time guarantees. Depending on the implementation, the other levels of the controller have less strict requirements and can run at a slower pace. The safety layer has special demands ensuring its stable operation even if the other levels fail. For safety critical systems, this may force the safety system to run on a different processor or even on a different computer system.

### 3.6.7 Co-simulation

In general, controllers consist of multiple levels, running with varying timing demands on different processors, where each level has a different coding architecture. In fact, the code for the higher levels of control for products that needs to be tolerant against potential faults are typically significant larger than the loop controllers. A discrete event modelling and simulation tool like Overture/VDM is far more capable of properly modelling these controllers than continuous-time modelling programs like 20-sim. Therefore, it makes sense to couple 20-sim with Overture/VDM using co-simulation.

## 3.7 A Small Note on Notation

In this chapter, the blocks and icons in all models are connected using single-lined arrows, each of these lines represents a single variable (in the case of block diagrams) or a single conjugated pair of variables (in the case of bond graphs). Later on in this book, in Chaps. 8 and 11, you will see the use of double-lined arrows to connect blocks and icons in some models, similar to those in Fig. 3.41.

Double lines in 20-sim mean that multiple variables are being shared between the icons they connect rather than just one and their principal goal is to maintain clarity of the diagram.

### 3.8 Conclusion

In the space available, only an introduction to modelling and control of physical systems could be given. With the information provided here, it should be possible to make simple models of physical systems and apply an elementary controller. To design more sophisticated (digital) controllers, a more thorough study of control-system design is necessary, we refer for example to [4]. In the final realisation of a control system, the controller itself may be only a minor part of the code. Safety software and start-up and shut-down procedures may require many more lines of code, as will be shown in the next chapter.

# Chapter 4

## Discrete-Event Modelling in VDM

Peter Gorm Larsen, John Fitzgerald, Marcel Verhoef, and Kenneth Pierce

### 4.1 Introduction

As indicated in Chap. 3, the design of real software controllers demands notations and tools that have the features needed to describe systems that evolve via a series of discrete events and so express the structure and logic of layered architectures. VDM is one such formalism and is introduced in this chapter. VDM is a general-purpose formalism for describing discrete event systems, with many more features than are needed here. This chapter covers the core of the language so that a reader with limited experience of DE modelling and software design will be able to understand the DE controller models used in the first examples and begin to develop new controllers of their own.<sup>1</sup> More sophisticated features, especially those needed to describe a controller structure, are introduced in later chapters. Readers requiring comprehensive coverage of VDM for controllers are directed to Appendix B, the manuals available at the Overture web site<sup>2</sup> or to introductory texts [35, 36].

---

<sup>1</sup>VDM has three dialects: the ISO-standardised VDM Specification Language (VDM-SL), its object-oriented extension (VDM++) and a further extension for describing real-time systems (VDM-RT). The latter dialect is used in most of this book.

<sup>2</sup><http://www.overturetool.org>.

P.G. Larsen (✉)  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

J. Fitzgerald • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk) [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

M. Verhoef  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)



Complexity is a critical problem in the design of modern control software. Software's discontinuous behaviour means that model-based descriptions must deal with large and complex spaces of system states in which it is not possible to safely draw conclusions from tests on selected sample values. DE modellers typically deploy several techniques to manage complexity, and modelling languages such as VDM support these. The first is *abstraction*, which is the deliberate suppression of detail that is not relevant to a model's purpose. Second, *structuring techniques* are used to organise models so that the system can be understood as the composition of units that can be modelled and analysed separately. Third, a high degree of rigour in modelling allows the *systematic analysis* of models and identification of flaws; given a sufficiently formal modelling language, some analyses are so systematic that they can be performed automatically. Conventional programming languages have the benefit of a structuring mechanism and some degree of rigour, but it is the capacity for abstraction that makes a DE *modelling* language such as VDM radically different and able to provide early-stage analysis of controllers before they are implemented in code on specific devices.

In this chapter, we introduce models of successively more sophisticated controllers (full versions of all the controller models can be found on the book's web site). First, we introduce basic abstractions in data and functionality (Sect. 4.2). This is enough to allow us to present a simple controller model in VDM (Sect. 4.3), based on the PID controller for the torsion bar described in Sect. 3.1. However, the real benefits of DE modelling lie in the ability to describe the logic of supervisory control, and this requires a richer collection of data models than just numbers. In Sect. 4.4, we introduce these models in VDM and in Sect. 4.5 show how they enable us to model an enhanced torsion bar controller that visits a series of defined waypoints. This ability to model supervisory control is particularly valuable in ensuring safety; we illustrate this with an example model of a torsion bar controller that avoids specified regions (Sect. 4.6).

As we consider more sophisticated controllers, it becomes necessary to use structuring mechanisms, with separate component parts linked in an architecture and functioning concurrently. In Sect. 4.7, we introduce *object-oriented* abstractions for modelling software structure, and in Sect. 4.8, we introduce the *thread* concept for concurrency. We present a fully structured model of the torsion bar controller by way of illustration. The final step in the DE modelling journey is from the controller software to the digital system that runs it: the CPU on which software runs determines its practical performance. In Sect. 4.9, we introduce the features of VDM that allow us to model the deployment of a software process onto processors with specified characteristics. Together, these features—data, functionality, structuring, concurrency and deployment—constitute the DE part of a co-model. Having introduced them all, we are then ready to present the first co-simulation in Chap. 5. Finally, Sect. 4.10 summarises modelling concepts introduced in this chapter.

## 4.2 Basic Elements: Data and Functionality

Controllers typically take decisions and act on *data* that are sensed or come from setpoints determined by users, and so it is natural to base models of controller software around descriptions of data. The computations that are available to be performed on data are referred to as the *functionality* of a computing system. The functionality of a controller is exposed at its interface with the environment and controlled plant and includes the main control functions, as well as the interactions with external users, for example.

A model of a controller needs to contain representations of data and functionality. Data include the *monitored* data coming into the controller from sensor devices and the *controlled* data passed from the controller to actuators in order to effect change. The core functionality of the controller is the *control step*: a sequence of instructions that interprets the monitored values and sets the controlled data (sending the desired values to the actuators). The control step can, for example, be executed periodically, at a frequency that is determined by the control characteristics required of the application. The periodic execution of the control step in a loop means that this structure is referred to as a *loop controller* (see Sect. 3.6.6). In this section, we introduce the elements of a basic loop controller model in VDM.

### 4.2.1 Data

In a control setting, one naturally thinks first of data that is numeric or quantitative, for example, the angular velocity of a wheel, but a controller may also need to know which one of a number of modes it is in (Starting, Cruising, Shutdown, etc.), or which mode it must move to next. Data therefore includes both qualitative and quantitative values of interest drawn from collections of values that we term *types*. Data are not only single values, but a supervisory or sequence controller may require more sophisticated structures to represent collections of values such as sequences of recorded values, tabular mappings between operating modes, etc. We therefore consider types of structured data as well as basic values.

The basic types in the VDM notation are summarised in Table 4.1. Note the abstraction in these types: there is neither a built-in upper limit on the natural numbers nor a lower one on the integers, just as there is no predefined precision on reals. These additional restrictions, if desired, are explicitly declared.

Controllers are configured around specific values that represent design parameters. Software engineers tend to refer to such values as *constants* because they do not change as the controller operates. In VDM, such constants are defined in a section of text that begins with the keyword “**values**”. For example, consider the torsion bar controller introduced in Sect. 3.1. In describing a controller design

**Table 4.1** Basic numeric types in VDM

Type	Notation	Example values
Natural numbers incl. 0	<b>nat</b>	0, 1, 2, ...
Natural numbers excl. 0	<b>nat1</b>	1, 2, 3, ...
Integers	<b>int</b>	..., -2, -1, 0, 1, 2, ...
Reals	<b>real</b>	...-3.7, ..., 0, $\pi$ , ..., 5.8372, ...

in VDM, we might define some conversion constants such as the encoder resolution and the belt ratio as follows:

```

values
  -- conversion constants used
  ENC_RESOLUTION : real = 2000.0;
  BELT_RATIO : real = 3.75;

```

Each definition gives the name of the value, followed by its type and specific value. Observe that we separate definitions by semicolons and that the formal text is annotated with useful notes as comments on lines prefixed “-”.

Data that vary during the execution of the software are referred to by means of *instance variables*. Unlike mathematical variables which typically represent single values, instance variables are names given to memory locations that can hold values which change during a computation. Thus, a reference to an instance variable  $x$  at one point in a calculation does not necessarily refer to the same value as  $x$  at another point. The values held in instance variables are set by means of assignments, written “*variable name* := *value*”. For example, the assignment “ $x := 3$ ” places the value 3 into the memory location referred to by the name  $x$  independent of what  $x$  was bound to before. Note that this is in contrast to the = sign used in value definitions where the values will never change.

Instance variables are used for many purposes in discrete event modelling of controllers. Here we consider two main uses. First, variables are used to record the monitored and controlled data. For example, a loop controller for the torsion bar might contain variable definitions to represent the monitored data from the motor encoder. Note that this data may be modified by the sensor—our controller software only monitors it. In VDM, variables are defined as follows:

```

instance variables
  -- monitored variable
  enc_motor: real := 0.0

```

For each variable, we give the type and define the initial value in the variable by means of an *assignment* denoted by the `:=` symbol. Our controller can set the motor (pulse width modulated) setting, and this is again modelled as a variable. The data that is sent out to the motor may be modified by the controller and is defined as follows:

```
instance variables
  -- controlled variable
  pwm_motor: real := 0.0
```

The second use of a variable is to store data that are local to the controller itself and that may be both read and modified by the controller internally. For example, a discrete event controller typically calculates the actuator settings by performing a calculation periodically, based on the current and previous values of the error. The controller must therefore remember the previous error and use it in the next cycle. We would therefore declare a variable to hold this data:

```
instance variables
  prev_err: real := 0.0
```

#### 4.2.1.1 Expressions

Controller models will typically contain formulae that describe complex values to be calculated or properties that must be checked. Such formulae are just representations of values, and though they may read variables, they do not write to variables (we say that they have no *side effects* when evaluated). Straightforward numbers, the simplest forms of numerical expression, have already been shown. More complex expressions can be constructed using the full range of numeric operators (see Appendix B). For example, the encoder signal is in the variable `enc_motor`. The rotation computed from the encoder signal is given by the following expression, which uses numeric division (`/`):

```
enc_motor / ENC_RESOLUTION / BELT_RATIO
```

where `ENC_RESOLUTION` and `BELT_RATIO` are defined as constant values as indicated above.

Certain operators on numeric values yield Boolean results (the values `true` or `false`). These are typically comparators such as `=`, `<`, `>=` (greater than or equal to) and `<=` (less than or equal to). As a consequence, some expressions such as “`x`

`<= min`” yield Boolean values. A range of operators is available for constructing Boolean formulae, including logical operators such as **not**, **and** and **or** (see Appendix B). For example, the expression:

```
x >= min and x <= max
```

records the assertion that `x` is between the limits `min` and `max`. Generalisations of such Boolean expressions will be presented in Sect. 4.4. Typically, `min` and `max` would be defined elsewhere, whereas `x` typically is a variable. We call Boolean expressions that depend on such variables *predicates*.

Experience shows that the most complicated logic operator to understand is the *implication*, written as `A => B`. In essence, this means that if `A` is true, then `B` shall also be true (in case `A` is false, the entire implication expression is true). This is reflected in the truth table for implication. The left-hand side of the implication is called the *antecedent*, and the right-hand side is called the *consequent*.

A	B	A => B
true	true	true
true	false	false
false	true	true
false	false	true

In Sect. 4.4, you will see this in action. Boolean expressions are used extensively. For example, a conditional expression is of the form

```
if EB
then E1
else E2
```

If the Boolean expression `EB` evaluates to **true**, the conditional expression overall is `E1`, otherwise `E2`. As an example, suppose that a controller has to limit the value of an output `x` to a maximum `max` and minimum `min`. The output value would be

```
if x < min then min
elseif x > max then max
else x
```

Note that it is also possible to use an **elseif** keyword if the logic requires more than two options. Many other forms of expression are defined in VDM and will be introduced as they are required. For a full explanation of expression forms, see the Overture VDM Language Reference Manual [59].

### 4.2.1.2 Data Types and Invariants

As the examples above suggest, an important application of Boolean expressions is in defining restrictions on the values held in variables. Such restrictions are represented in VDM as data type *invariants*. Invariants are Boolean properties that must be respected by all the values within a specified data type. For example, suppose we require that the value held in the variable  $x$  must always be between 1 and 12. In order to capture this, we need to introduce the notion of an *invariant*. In order to do this, we can define a type as

```
types
  Limited = real
  inv lim == lim >= 1 and lim <= 12
```

This definition declares a data type that contains only the real numbers between 1 and 12 inclusive. The **inv** keyword is followed by a name (`lim` in this case) of an arbitrary element from the type, and the invariant is defined after the `==` symbol. Suppose we declare an instance variable of this type:

```
instance variables
  x: Limited
```

Any attempt to assign an invalid value such as 0 to the variable would yield a runtime error. An operator is said to be *partial* if its result is undefined for some argument values. Perhaps the best known example is numeric division, in which the result is undefined if the denominator evaluates to zero. We seek to protect the application of a partial operator by applying a guard that ensures the undefined condition does not arise. For example, the following expression is undefined if  $y$  has the value  $-10$ :

```
v/(y+10)
```

If we wished to guard this, we might use an **if** expression, returning a 0 in the case where the division is undefined.

```
if y <> -10
then v/(y+10)
else 0
```

Note that, given the invariant on the type `Limited` in our small example, the following expression is guaranteed to be defined:

```
v/(x+10)
```

## 4.2.2 Functionality

We have considered so far the definition of data—values, variables, types and invariants—but we have not considered the modelling of computations that work on data. A computation operates on input data and manipulates it in order to generate output data. We distinguish two kinds of computation: *functions* and *operations*. Functions work on input *values* and calculate a result based only on those input values. Functions are pure and side effect free, which implies that calling a function with the same input parameters twice will always yield the same result. Operations, however, may also read from and write to the *instance variables* in the model. This enables operations to have a sense of history over multiple invocations, which potentially allows them to return different results with the same input parameters.

### 4.2.2.1 Function Definitions

Let us begin with a simple example of a function that computes the rotation from our encoder signal. In VDM, we model this as follows:

```

functions
  -- function to compute rotation from encoder signal
  enc2rot: real -> real
  enc2rot (penc) == penc / ENC_RESOLUTION / BELT_RATIO;

```

The **functions** keyword marks the beginning of a section of the model in which one or more function definitions are given. A function definition always has two parts: a *signature* and a *body*. The signature (here the line “enc2rot: **real** -> **real**”) declares the name of the function, the types of its inputs and the type of its result. The body consists of two parts to the left and right of the “==” respectively. To the left, we give the function name and the name of the input argument; to the right, an expression that defines the result. Note that the type of the defining expression must match the return type defined in the signature.

Defined functions may be used in other expressions. For example, we may write an expression using `enc2rot` elsewhere in the model. We might wish to represent the error value as the difference between the current setpoint and the angular position calculated from the encoder reading. This is given by the expression

```

setpoint - enc2rot(enc_motor)

```

Functions may have multiple input arguments. For example, if we were to provide a limiting function that restricts a value to a range, we might define it thus (with three input arguments):

```

functions
-- limit x between min and max
limit: real * real * real -> real
limit(x, min, max) ==
  if x < min then min
  elseif x > max then max
  else x;

```

If you have a complicated expression that you would like to make use of multiple times, VDM has a concept called a *let expression* that is useful. In this way, a new local constant can be declared and used subsequently. The syntax for this expression is

```

let name = some complex expression
in
  expr that makes use of name

```

Such an expression can be thought of as first calculating the complex expression and then afterwards binding the result of that calculation to the name which will be usable in the expression after the **in** keyword. Whenever name occurs, it will mean the value that was bound to it. In our torsion bar example, we could use this in a `get_setpoint` function that simulates a cycloid<sup>3</sup> curve:

```

functions
-- get setpoint for the given time (cycloid signal)
get_setpoint: real -> real
get_setpoint(call) ==
  let delta = 2 * MATH`pi * (t - START_TIME) /
    (STOP_TIME - START_TIME),
    cycle = AMPLITUDE * (delta - MATH`sin(delta)) /
    (2 * MATH`pi)
  in if delta < 0 then 0
  elseif delta > 2 * MATH`pi then AMPLITUDE
  else cycle
pre STOP_TIME > START_TIME;

```

Here `delta` is defined as a local constant based on a complex expression (in the same way, the constant `cycle` is defined to illustrate how multiple such constants can be made in one `let-expression`). In this definition, there are also references to `pi` and `sin`, which are standard mathematical concepts that are not built into VDM themselves. Instead, they are placed in a library called `MATH` that you can choose to include whenever needed. Other libraries will be illustrated later.

<sup>3</sup>See <http://en.wikipedia.org/wiki/Cycloid>.



Finally, the definition of `get_setpoint` also has a so-called *precondition* indicated by the “**pre** STOP\_TIME > START\_TIME” part at the end of the function definition. A precondition indicates that in order to call a function, you must ensure that a given condition is satisfied. Here the condition is meant to ensure that the partial division operator will not be called outside its defined area.

#### 4.2.2.2 Operation Definitions

Functions deliver a result based only on their input arguments and any global constants defined in the model. Operations, by contrast, may read and modify the instance variables as well. Consider a trivially simple example. Suppose that we have instance variables containing the current sensed angular position for the torsion bar controller, the setpoint and the current motor setting:

```
instance variables
  enc_motor: real := 0.0;
  setpoint: real := 0.0;
  pwm_motor: real := 0.0
```

We might wish to define a simple operation, named `CalcP`, that determines the new motor setting on the basis of the current error, and returns a Boolean value saying whether the error was within a specified tolerance.

```
operations
  -- calculate response for the current error
  CalcP: real ==> bool
  CalcP(tolerance) ==
    ( pwm_motor := Kp * (setpoint - enc2rot(enc_motor));
      return abs(setpoint - sensed) <= tolerance )
```

There are some important notational differences between the operation and function. The signature is distinguished from that of a function by the use of a “==>” arrow instead of the “->”. Notice that only the input and output types are shown in the signature—not those of the instance variables. However, the main distinction is in the body of the operation. Rather than containing a single expression that gives the result, the operation body is a series of individual calculation *statements*, each of which may read from or write to the instance variables. A single statement multiplies the error by a constant  $K_p$  to determine the motor setting.<sup>4</sup> A special **return** statement contains the expression that gives the output value. In our example, the

---

<sup>4</sup>This multiplication of the error by a constant is a form of *proportional* control,  $K_p$  is called the *proportional constant* and hence the name `CalcP` for the operation.

output value is true for regular sensor values and false for unusually large sensor values.

Just as there are several forms of expression available for the body of a function or an invariant, VDM offers several forms of statements for use in the bodies of operations. Again, here we just mention the most basic; others are introduced as they become required. For a full explanation of statement forms, see the Overture VDM Language Reference Manual [59].

Statements that are to be executed sequentially can be gathered into a *block* delimited by parentheses. Sequential compositions of statements are separated by a semicolon (and, by convention, line breaks). As an example, suppose that we wish to model a more sophisticated PID controller of the kind introduced in Sect. 3.6.2 using the cumulative error and the rate of change of the error. The following operation illustrates one method for determining the PID response and shows the use of a block to group statements:

```

operations
  -- calculate PID response for the given error
  CalcPID: real ==> real
  CalcPID(err) ==
    ( uI := uI + (err * SAMPLETIME);
      uD := (err - prev_err) / SAMPLETIME;
      prev_err := err;
      return (Kp * err) + (Ki * uI) + (Kd * uD) )

```

The operation calculates the controller output to be applied: the body of the operation has a return statement that computes the control output from the proportional, integral and derivative terms. It is preceded by three sequential assignment statements. The first two calculate the integral and derivative factors which are multiplied by tuning constants  $K_i$  and  $K_d$  in the return statement. The final assignment sets the current error to the variable `prev_err` for the next control cycle. The PID algorithm requires the following instance variables to be defined in the controller:

```

instance variables
  -- PID variables
  uD: real := 0.0;
  uI: real := 0.0;
  prev_err: real := 0.0;

```

This is a far-from-perfect controller, but our purpose here is to introduce the modelling concepts rather than present a sophisticated control algorithm.

### 4.3 Example: A Basic Controller Model

We have covered enough of the basic elements of VDM to show a discrete-event model of a simple software controller for the torsion bar introduced in Sect. 3.1. The aim of this first example is simply to show the structure of a controller. With only numbers, simple expressions and assignments in our modelling language, we cannot describe much more than basic loop control. However, later in the chapter, we introduce the features of a DE-specific modelling method like VDM that enable this. Although simple, the torsion bar system has the main elements (e.g. controller, sensor, amplifier, electric motor, transmission) that we also see in more complex mechatronic systems such as robots.

We have so far presented some extracts from a model called `TorsionBar1-Minimal`; here we add the remaining elements. In order to achieve periodic execution of a control algorithm, a periodic thread is used (features for defining these will be introduced in detail in Sect. 4.8.1). It is specified as follows:

```
thread
  -- define periodic thread (nanoseconds)
  periodic (SAMPLETIME * 1e9, 0, 0, 0) (Step)
```

where `Step` is the name of the operation to be executed in each iteration, and the time length (i.e., the period) between its execution and the next will be `SAMPLETIME * 1e9`. VDM uses nanoseconds as the core time unit, hence the `1e9` factor. The periodic operation `Step` is defined as follows:

```
operations
  -- periodic operation
  Step: () ==> ()
  Step() == (
    -- write held value to actuator
    pwm_motor := hold_pwm;

    -- calculate new hold value
    let err = get_setpoint(time/1e9) - enc2rot(enc_motor)
    in
      hold_pwm := limit(CalcPID(err), -1, 1)
  )
```

where the “`()`” in the signature indicates that no input parameters are required and no results are returned, that is, the operation affects only the instance variables. The instance variable `hold_pwm` is declared as

```
instance variables
  -- store sensor and hold values
  hold_pwm: real := 0.0
```

This will hold the pwm value calculated in the previous iteration. The actual calculation makes use of functions and operations that have already been presented in the text. The error (`err`) is calculated and used with a PID controller to calculate the best correction for the motor. The only remaining parts are the constant value definitions used in calculations within the model. These are

```

values
  -- thread period in seconds (0.02 = 50Hz)
  SAMPLETIME : real = 0.02;

  -- conversion constants used
  ENC_RESOLUTION : real = 2000.0;
  BELT_RATIO : real = 3.75;

  -- cycloid signal parameters
  START_TIME : int = 0;
  STOP_TIME : int = 1;
  AMPLITUDE : int = 1;

  -- PID controller parameters
  Kp : real = 2.4;
  Ki : int = 12;
  Kd : real = 0.12;

```

## 4.4 Modelling with Structured Data

Very basic control algorithms can be modelled using numbers alone, but sophisticated supervisory control requires a richer vocabulary. In this section, we introduce a wider range of types that can be used to represent non-numeric data (Sect. 4.4.1) and describe the important types that model structured collections of values: records, sets, sequences and mappings (Sect. 4.4.2). These abstractions will be illustrated using an extended controller model called `TorsionBar2-Visit`. In this model, we require the load disk to move through a set of positions (angles between 0 and  $2\pi$ ) by ordering them into a path that minimises total travel distance. Similar requirements often arise in robotics, for example, in robots that have to follow a track of points for spot welding.

### 4.4.1 Nonnumeric Data

#### 4.4.1.1 Characters

VDM contains a nonnumeric basic type called `char` containing the ordinary characters. This can, for example, be used in strings that can be represented as sequences of characters (we come back to sequences in general in Sect. 4.4.2):

```
types
  String = seq of char
```

Strings can be used to generate descriptive messages using the standard IO library.

#### 4.4.1.2 Union Types and Quote Types

Supervisory controllers often have to change mode, since different requirements may apply during different phases of operation such as start-up, shutdown or error recovery. In such circumstances, a modelling abstraction is needed to indicate the current mode. While it may be tempting to encode these as numbers or characters, this is certainly not in the spirit of abstraction, and the meaning of a model is made more intuitive if it is possible to introduce simple labels that can be used in this sort of situation. In VDM, *quote types* serve this purpose. A quote type is simply a label; for example, to record the current direction of travel for a robot, one might simply want to introduce labels for “left” or “right” motion. Each of these would be a quote type. For example, <LEFT> is a quote type: it contains just one *quote value* (also called <LEFT>).

If a variable is required to assume one of a range of distinct values that are not numbers or characters, it is often appropriate to define a type that represents the union of several quote values to represent an enumeration. For example, if our robot’s direction can be left or right or unknown (perhaps because of a sensor failure), it would be worth defining a type `Direction` to include the range of possibilities:

```
types
  Direction = <LEFT> | <RIGHT> | <UNKNOWN>
```

where the `Direction` type contains three values listed to the right of the “=” sign and separated by the “|”. A variable of type `Direction` will always have exactly one of the three distinct values shown. We may wish to model shutdown behaviour in the case of sensor failure. In this case, we may need to interrogate an instance variable of type `Direction`:

```
if d = <UNKNOWN> then Shutdown -- call some shutdown operation
else ...
```

The “|” symbol in the type definition represents the union of types, accumulating the types on either side of the vertical bar into a single larger type. This type union operator can be used for any type, but it is often used either between quote types or to form the kind of enumeration of possible values that we have in this example.

## 4.4.2 Structured Collections: Records, Sets, Sequences and Mappings

Some of the most powerful abstractions in VDM permit the modelling of the highly structured data required to manage decision-making in supervisory controllers. In this section, we introduce record structures that allow several related data items to be treated as a single unit. We go on to introduce collection types (sets, sequences and mappings) that allow whole groups of data to be treated as one. These abstractions allow complex data to be represented in such a way that avoids the introduction of unnecessary detail into the model. For example, mappings can be used to abstract away from the detail of pointer structures, when all that matters for the model is the representation of relationships between data values. Of course, details such as pointer structures can be represented using these abstractions if they are necessary for the model's purpose.

### 4.4.2.1 Records

It is often convenient to be able to treat data items that relate to the same subject as though they were a single item. Such compound structures are termed *records* in VDM. A simple example might be data relating to each user of an information system, for example, their name, user identifier and access rights. Such items might typically be treated together in an application and so may be grouped into a single item. In our torsion bar example, suppose that we require to control the process of moving from one setpoint to the next within a specified time interval. Here it makes sense to gather information of different types together in a common structure that represents the change in position:

```
types
-- represents setpoint change time / value
SetpointChange ::      setpoint : real
                      travel_time : Time
                      wait_time : Time
```

This defines a *record type* called `SetpointChange` with three components (called *fields*): the first is a real number representing the setpoint, the second and third represent travel and wait times. The latter are of type `Time`, and we assume that this type is defined elsewhere.

Operators are required to build records from their constituent data items and to select the individual data from within them. If a record type `T` is defined as shown above, an operator `mk_T` (called a *constructor*) becomes available to build records of that type. In the torsion bar example, the constructor is called

`mk_SetpointChange`, so one particular change value could be written as follows:

```
mk_SetpointChange(0,200,50)
```

indicating a setpoint at 0 rad, time to change to this setpoint 200 ns and a wait time of 50 ns.<sup>5</sup>

We will often need to select the values held in individual fields of a record. We use a simple dot notation for this. For a record value `r` containing a field called `f`, the expression `r.f` is the data value in the `f` field of the record value `r`. In our example above, if we have a value `ch` of type `SetpointChange`, the expression

```
ch.travel_time
```

is the travel time component of `ch`. To take another example combining the constructor and selector, the following Boolean expression yields `true`:

```
mk_SetpointChange(0,200,50).wait_time = 50
```

#### 4.4.2.2 Sets

Sets are one of the key abstractions used for describing collections of values. A set is a finite unordered collection: there is no sense of one value in a set being before or after another. Furthermore, there is no concept of a value occurring several times in a set: a value is either in the set or not. We say that a value is *an element* of a set if it occurs in the set. Sets are written with curly braces around the elements, which are separated by commas. The empty set, which has no elements, is written `{ }`. The fact that elements are unordered means that the set `{ 9, 2 }` is exactly the same as the set `{ 2, 9 }`. We use the operator `card` to get the number of *distinct* elements in a set (for example, `card { 1, 4, 2, 4 }` yields the value 3).

In the `TorsionBar2-Visit` model, the controller must maintain a collection of the desired angles which the torsion bar must turn to at specific times. One such possible collection is a set, so below we will illustrate some of the set operators using such angles. If we have two sets of angles, these can be combined using the `add_angles` function:

---

<sup>5</sup>For the interested reader, the mechanism at work here is termed *tagging*. Record definitions typically use the “`::`” instead of the equality symbol used in other type definitions. This indicates that all values in the type carry a *tag* holding the name of the type, so that given a record, we can tell which record type it belongs to, even if we have two record types with the same kinds of field. For a tagged type `T :: ...`, the constructor is `mk_T`, where `T` is the tag.

```

functions
-- join two sets of angles together
add_angles: set of Angle * set of Angle -> set of Angle
add_angles(s1,s2) == s1 union s2;

```

where the **union** operator coalesces two sets. Conversely, we can extract the set of values that are in two sets (the *intersection*) using the **inter** keyword. In order to assert that a value *a* is in a set *s*, we simply write

```
a in set s
```

where the **in set** keyword will yield **true** if *a* is indeed one of the members of *s* and **false** otherwise. It is usually impractical to enumerate the elements of sets in real applications. Instead, sets are much more often constructed by *set comprehension*.

```
{ value-expression | binding & predicate }
```

The *binding* binds one or more variables to a type or set. The *predicate* is a logical expression using the bound variables. The *value-expression* is also an expression using the bound variables, but this expression defines a typical element of the set being constructed. The comprehension represents the set of all values of the value expression for each possible assignment of values to the bound variables for which the predicate is **true**. For example, the comprehension

```
{x**2 | x:nat & x < 5}
```

represents the set of all values of *x* to the power of 2, where *x* is a natural number such that *x* is less than 5, that is,

```
{x**2 | x:nat & x < 5} = {0**2,1**2,2**2,3**2,4**2}
                        = {0,1,4,9,16}
```

The operators for sets are summarised in Table [B.2](#).

#### 4.4.2.3 Sequences

Sequences are finite collections of values in which there is an ordering among elements. Instead of a value simply occurring or not, it may be repeated several times, and this repetition may be significant. Sequences are enumerated presented in square brackets, and the empty sequence is written as `[]`. There are many basic operators that are used on sequences; an overview is given in Table [B.3](#).



For the torsion bar case study, suppose that we need to visit a collection of setpoints in order (so the order is significant). One design for doing this is to hold a queue of setpoint changes modelled as a sequence (initially empty), so that the values in the queue simply need to be read and applied in order. The queue would be declared as follows:

```
types
  -- setpoints to visit
  queue: seq of SetpointChange := [];
```

where `SetPointChange` is the record type defined above. Several useful operators are defined on sequences. The length of a sequence `queue` is given by “`len queue`”, whereby the length of `[]` is zero. For a non-empty sequence, `hd` gets the “head” or first element, and `tl` gets the “tail”; the sequence remaining after the head is removed. These features allow us to define the `Step` operation in the torsion bar controller. Here the setpoint change at the head of the queue is selected for application, and the queue is updated so that the head has been removed:

```
Step: () ==> ()
...
if ... and len queue > 0
then let next_pos = hd queue in (
  queue := tl queue;
  ...
```

The remainder of the `Step` operation goes on to use the `next_pos` value.

Sequences can be joined together by the concatenation operator (“`^`”); for example, the expression `queue ^ setpoints` represents a sequence formed by the elements of `queue` followed by the elements of the `setpoints` sequence, in order. We can retrieve elements of a sequence by means of their index numbers. For example, for a sequence `queue`, the third element is represented by `queue (3)`.

Other operators allow us to extract the sets of elements (**elems**) or indices (**inds**) of a sequence. These operators return sets, thus losing information about ordering and duplicates. Notice that some operators are partial in that they are undefined if applied improperly. For example, the head of a sequence is undefined if the sequence is of length zero (hence the guarding condition on the “`if`” line in the `Step` definition. Accessing the `n`th element of a sequence `queue` only makes sense if the expression `n in set inds queue` is **true**.

Sequences are often defined using a *comprehension* construction similar to that for sets. For example, suppose that we have a set `s` of angles that should be visited, but we wish to restrict the set to those that are within the range `0, ldots, pi/2`. We might express this restricted set by means of a comprehension:

```
{a | a in set s & a <= MATH`pi/2}
```

Suppose we want these angles to be in increasing order so that we can visit them in sequence without reversing. The lack of ordering information in a set makes it a poor choice here; a sequence of angles would be preferable. But how should the sequence be ordered? In VDM, sequence comprehension provides a default ordering of increasing numeric values. For example, consider the following expression:

```
[a | a in set s & a <= MATH`pi/2]
```

This takes all the values for  $a$  in the set  $s$  that satisfy the condition and returns them in increasing order. In fact, comprehensions can be more sophisticated, allowing us to build sequences of complex values. For example, the following expression returns a sequence of values formed by halving the angles in the desired range:

```
[a/2 | a in set s & a <= MATH`pi/2]
```

In sequence comprehensions, we require that the bound variable (here,  $a$ ) comes from a numerical type, so we have an order in which to construct the elements of the sequence. The result is evaluated for each value of the bound variable that satisfies the predicate, in numerical order.

In the `TorsionBar` example, the `Visit` operation manages the visiting of specified angle positions by first sorting a set of angles into a sequence (`sorted`). This is ordered and converted to a sequence of rotational positions (`setpoints`) by means of a sequence comprehension, and then it loops through all the positions in the queue, converting them to `SetPointChange` records:

```
operations
-- visit a set of angles
Visit: set of Angle ==> ()
Visit(s) == (
  let sorted = Sort(s),
      setpoints = [angle / (2 * MATH`pi)
                  | angle in set elems sorted]
  in for sp in setpoints ^ [0] do
    queue := queue ^
             [mk_SetpointChange(sp, TRAVEL_TIME, WAIT_TIME)]
  )
pre s <> {};
```

In this operation, we used a *sequence for loop* (“**for identifier in sequence expression do statement**”) to iterate through the `setpoints` in the sorted list. Notice our use of a precondition in the operation. This records the assumption that the `Visit` operation will only be called with non-empty sets of angles.

The dual of a precondition is a *post-condition*. A post-condition is used to record the guarantee that a function or operation makes to its users, provided its assumption (precondition) is satisfied. It characterises the result and “after” state of

the function/operation, giving its essential properties, and so effectively frees the modeller from the need to give a specific algorithm. Consider an example in which we define the `Sort` operation used above:

```
operations
-- sort angles in ascending order
Sort: set of Angle ==> seq of Angle
Sort(s) == ( ... )
```

How should we describe the sorting process? There are many possible algorithms, and we may not wish to bias the developer of the controller towards one rather than another: an algorithm that is satisfactory for modelling may not have adequate performance when implemented in real software. If we wanted merely to characterise the sorting algorithm, we might do so by means of a post-condition:

```
operations
-- sort angles in ascending order
Sort: set of Angle ==> seq of Angle
Sort(s) == ( ... )
post elems RESULT = s and
    len RESULT = card s and
    forall i in set inds RESULT &
        i <> len RESULT => RESULT(i) <= RESULT(i+1);
```

Note that the **RESULT** keyword just refers to the output produced by the operation. The post-condition uses a *universal quantified expression* (**forall** ... & ...) to quantify over all the indices of the result sequence, simply saying that each element is less than its successor in the sorted sequence. This post-condition captures the true requirement for the operation. We can then add an algorithm, with the lesser risk of biasing subsequent development:

```
operations
-- sort angles in ascending order
Sort: set of Angle ==> seq of Angle
Sort(s) ==
    ( dcl sorted: seq of Angle := [];
      -- insert each element into the right position
      for all a in set s do sorted := insert(a, sorted);
      return sorted )
post elems RESULT = s and
    len RESULT = card s and
    forall i in set inds RESULT &
        i <> len RESULT => RESULT(i) <= RESULT(i+1);
```

Note that the body of `Sort` uses a `dcl` declaration to declare a local variable that is visible only within the block enclosed by parentheses in which it is declared. In this case, the declared variable just holds the sorted sequence as it is accumulated. The algorithm is also described using a *set loop statement* (“`for all identifier in set set expression do statement`”), which allows us to iterate through the values in a set.

#### 4.4.2.4 Mappings

We have shown the benefit of data structures that deal with collections of values. In many supervisory control applications, however, we also need to record *relationships* between values; this is done using mappings. A mapping is a collection of pairs of values, each of which is called a *maplet*. For example, if we wished to record a series of “speed limits” for different regions of rotation, we might want to record a mapping from angles to angular velocities:

```
instance variables
LIMITS: map real to Speed := {MATH`pi/2  |-> <FAST>,
                             MATH`pi    |-> <SLOW>,
                             3*MATH`pi/2 |-> <FAST>}
```

A mapping is a directional relationship: the type of a mapping (here `map real to Speed`) gives us the types of the “from” and “to” sides of the relationship; these are called the *domain* and *range* types, respectively. Each maplet “`x |-> y`” connects a domain element to a range element. Mappings do not have to cover the whole of the domain type: the values that are connected to range values form a set called the *domain* of the mapping, and the values mapped to are called the *range*. Mappings can be arbitrarily large, but they are always finite in size. The empty mapping (without any elements) is written as `{ |-> }`. Operators for mappings are presented in Table B.4.

## 4.5 Example: Supervisory Control

We have already used extracts from the updated `TorsionBar2-Visit` model, which shows how supervisory control can be modelled in VDM. Below we will introduce the remaining elements of the model. The `Angle` type represents the desired angle in radians as a real number restricted by an invariant, as introduced in Sect. 4.2.1.2:

```

types
  -- represents an angle in radians
  Angle = real
  inv a == 0 <= a and a <= (2 * MATH`pi)

```

The same holds for the Time type:

```

types
  -- represents a time, must be positive
  Time = real
  inv t == t >= 0

```

The periodic control thread introduced in Sect. 4.3 is unchanged, but the Step operation has been updated to take the setpoints in the queue into account:

```

operations
Step: () ==> ()
Step() == (
  -- write held value to actuator
  pwm_motor := hold_pwm;

  -- change setpoint if necessary
  if len queue > 0 then (
    let now = time/1e9,
    mk_SetpointChange(sp, t_time, w_time) = hd queue
  in (
    if now >= next_time then (
      queue := tl queue;

      -- update setpoint generation variables
      base := enc2rot(enc_motor);
      amplitude := sp - enc2rot(enc_motor);
      start_time := now;
      stop_time := now + t_time;

      -- set next change time
      next_time := now + t_time + w_time
    );
  );
);

-- calculate new hold value
let err = get_setpoint(time/1e9, base, amplitude,
  start_time, stop_time) - enc2rot(enc_motor)
in hold_pwm := limit(CalcPID(err), -1, 1);
)

```

where **time** refers to the total simulation time in nanoseconds. The `mk_Setpoint Change (sp, time, wime)` on the left of the equality in the `let` expression is a pattern that assigns the elements in the head of the queue to the `sp`, `t_time` and `w_time` of the `SetpointChange`. The signature for the `get_setpoint` function has been changed; the new version is

```

functions
get_setpoint: Time * real * real * Time * Time -> real
get_setpoint(t, base, amplitude, start_time, stop_time) ==
  if stop_time - start_time = 0 then 0 else
    let delta = 2 * MATH`pi * (t - start_time) /
              (stop_time - start_time),
        cycle = amplitude * (delta - MATH`sin(delta)) /
              (2 * MATH`pi)
    in if delta < 0 then base
        elseif delta > 2*MATH`pi then amplitude + base
        else cycle + base
  pre stop_time >= start_time

```

The difference from the `TorsionBar1-Minimal` model is that values that were previously constants are now variables. Such instance variables cannot be seen inside a function definition, and thus additional arguments have been introduced. Thus, to complete the `TorsionBar2-Visit` controller, we have new instance variables:

```

instance variables
-- track time between setpoint changes
next_time: real := 0;

-- variables for setpoint generation
base: real := 0;
amplitude: real := 0;
start_time: real := 0;
stop_time: real := 0;

```

Finally, the remaining setpoint time change constants need to be set:

```

values
-- setpoint time changes
TRAVEL_TIME = 1;
WAIT_TIME = 0.5;

```

## 4.6 Example: Controlling for Safety

Systems often have safety requirements, and parts of these may be delegated to software as a safety function. In the `TorsionBar3-Monitor` model, we have, for example, required that the controller meets a safety constraint that the load disk must not enter a specified region. Let us now add an end stop safety constraint: the load disk must not enter a region at the end of the turn, for example, in the range  $(2\pi/3, 2\pi)$ , and must go slowly in a region leading up to this, for example,  $(5\pi/6, 2\pi/3)$ . In VDM, these constants are modelled as

```

values
  -- no go region
  NO_GO_MIN = 5*MATH`pi / 3;
  NO_GO_MAX = 11*MATH`pi / 6;

  -- slow region
  SLOW_MIN = 3*MATH`pi / 2;
  SLOW_MAX = 2*MATH`pi

```

The `NO_GO_MIN` and `NO_GO_MAX` must be incorporated in the pre-condition for the `Visit` operation:

```

operations
  -- visit a set of angles
  Visit: set of Angle ==> ()
  Visit(s) ==
    body unchanged
  pre s <> {} and
    forall a in set s & (a < NO_GO_MIN or a > NO_GO_MAX);

```

In real machines, the speed is mostly linearly limited from maximum when entering the speed limit region to zero when entering the no-go region. This serves to avoid sudden changes or “bumps” when crossing a region’s borders. The speed limit can be modelled in VDM as another constant:

```

values
  -- speed limit in slow region {rad/s}
  SPEED_LIMIT = 1;

```

We will introduce a safety monitor that stops the controller and returns it to 0 degrees if the speed limit or no-go region is entered. This is of course just one strategy; another approach might impose the speed limit and ignore requests to

enter the no-go region. In order to estimate the time, we need to record the previous encoder values. This can be done as an instance variable as

```
instance variables
  -- encoder sample
  prev_encl: real := 0.0;
```

This can be used in a new CheckMonitor operation:

```
operations
  CheckMonitor: () ==> ()
  CheckMonitor() == (
    -- read sensor value, calculate speed
    let encl = (enc_load / ENC_RESOLUTION) * 2 * MATH`pi,
        speed = (encl - prev_encl) / SAMPLETIME
    in (if encl >= NO_GO_MIN and encl <= NO_GO_MAX
        then -- inside no-go region
            EmergencyStop()
        elseif encl >= SLOW_MIN and encl <= SLOW_MAX
        then -- over speed limit in slow region
            if speed > SPEED_LIMIT then EmergencyStop();

    -- record sensor sample
    prev_encl := encl
  )
)
```

where a new EmergencyStop operation is used. This is defined as

```
operations
  -- stop if speed limit or no-go region are violated
  EmergencyStop: () ==> ()
  EmergencyStop() == (
    -- stop movement; clear setpoint and queue
    hold_pwm := 0;
    start_time := 0;
    stop_time := 0;
    base := enc2rot(enc_motor);
    queue := [];
  )
)
```

which essentially sets everything to initial values and stops the output to the actuator (the motor). Finally, the safety constraints need to be taken into account in an updated version of the Step operation (where the CheckMonitor test is carried out before sending output to the actuator):



```

operations
Step: () ==> ()
Step() == (
  -- write held value to actuator
  pwm_motor := hold_pwm;

  -- check for no-go and speed violations
  CheckMonitor();

  -- change setpoint if necessary
if len queue > 0 then (
  let now = time/1e9,
      mk_SetpointChange(sp, t_time, w_time) = hd queue
  in (
    if now >= next_time then (
      dcl travel_time: real := t_time;
      let next_angle = 2*MATH`pi * sp in
        if next_angle = 0 or
          (next_angle >= SLOW_MIN and
           next_angle <= SLOW_MAX) then
          -- increase travel time to limit speed
          let distance = abs (next_angle -
                             (2*MATH`pi * enc2rot(enc_motor)))
          in travel_time := distance / 0.5;

      -- update queue
      queue := tl queue;

      -- update setpoint generation variables
      base := enc2rot(enc_motor);
      amplitude := sp - enc2rot(enc_motor);
      start_time := now;
      stop_time := now + travel_time;

      -- set next change time
      next_time := now + t_time + w_time
    );
  );
);

-- calculate new hold value
let err = get_setpoint(time/1e9, base, amplitude,
                      start_time, stop_time) - enc2rot(enc_motor)
in hold_pwm := limit(CalcPID(err), -1, 1);
);

```

Note how the SLOW\_MIN and SLOW\_MAX constants also are used to limit the speed by increasing the travel time.

## 4.7 Object-Oriented Structuring

So far, we have shown how the abstraction features of DE modelling can be used to support the description of supervisory control. Although these features can do a great deal to help master complexity, models of real systems can still be complex. The structuring features included in our DE modelling framework help to manage this complexity. In the remainder of this chapter, we will discuss basic model structuring features using the torsion bar again, presenting a new model (*TorsionBarBaseline*) which has no more functionality than the models shown so far, but which does show how a model can be structured as a group of units that can be understood individually, and together supply the necessary control.

VDM supports a form of structuring based on *object-orientation*, a philosophy in which systems are treated as groups of individual entities termed *objects*, which each provide functionality that other objects may use. An object is a unit that can be understood on its own: it contains data and functionality, some of which is offered publicly and some of which remains private to the object. This notion of privacy allows objects' internals to be modified safely without compromising the overall system's capability, so long as the interfaces offered to other objects are preserved.

In a given system, there may be many objects of the same kind. For example, we might define a system composed of an array of smart sensors. These sensor objects will have similar interfaces, and so it makes sense, rather than repeating their definitions, to define a template for this kind of object. Such a "kind" of object is termed a *class*, and a template for objects of a class is termed a *class definition*. An object is often referred to as an *instance* of a class. In VDM, objects are dynamic in that they may be created, interact and be destroyed during the life of a system. By contrast, classes in VDM are static structures that do not change during a system's life: they are just templates used to create instances.

VDM objects have local state data in the form of *instance variables*. Within the class definition for a given object, the full range of types and operators introduced so far can be used to define local constant values, new data types, functions and operations.

Objects can be either passive or active. Passive objects are typically used as common stores of data for other objects, whereas an active object can be thought of as a separate machine, performing its own *thread* of control once it has been started: our smart sensors would probably be considered active objects if they monitor and raise an alarm if a condition is exceeded, for example. If a system has several active objects operating concurrently, it is necessary to have features for managing their execution. VDM provides a logic-based mechanism termed *permission predicates* for this purpose.

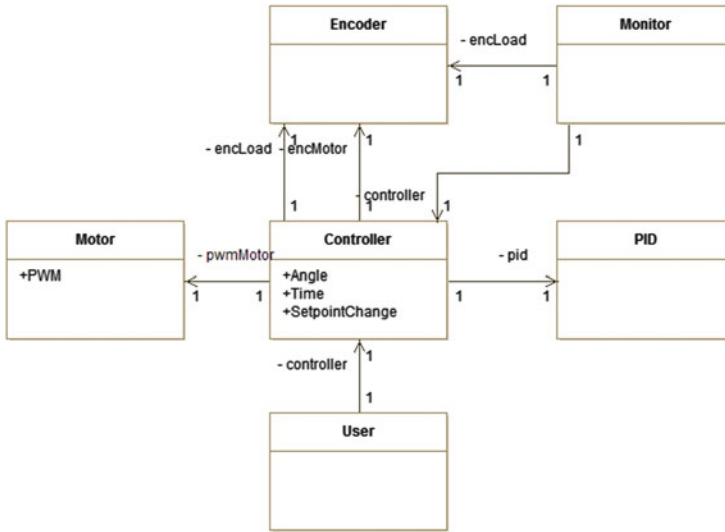


Fig. 4.1 UML class diagram with connections to/from the Controller class

#### 4.7.1 Structure of the TorsionBarBaseline Model

A graphical representation of the classes in a model is called a *class diagram*. In the remainder of the book, we will use a style of object-oriented description based on the Unified Modeling Language (UML) [15]. To introduce this, we consider a structure for the TorsionBarBaseline DE model.

The model has five user-defined classes, a *system class* (which we will come back to in Sect. 4.9) and five *library classes* (like the MATH and IO classes used above). An overview of the relationship between the user-defined classes and the PID library class is given in Fig. 4.1 in the form of a UML *class diagram*.

In a class diagram, classes are represented by boxes with two parts: the upper part holds the name of the class, and the lower part gives the type definitions inside the class. It is also possible to have parts that show the attributes (i.e., the values and instance variables defined in a class) and the functionality provided by the class (i.e., its function and operation definitions), but these and some of the libraries have been omitted from Fig. 4.1. Each class in a UML class diagram corresponds to a class in a VDM model.

Objects exist in relationships with one another. For example, we expect to have a single controller object in our model, but this will be linked to a specific object representing the motor, an object for the motor encoder, an object for the load encoder and so on. These relationships between objects of certain kinds are termed *associations* and are shown by labelled arrows in a class diagram, as in Fig. 4.1.

The underlying idea here is that, if we look at a controller object, we should be able to follow the associations to find the motor and encoders to which it is connected. An association in a UML class diagram corresponds to an instance variable in the VDM, which records the object at the target end of the link. For example, the four links from the controller in our example are represented as the following instance variables in the VDM:

```

class Controller

instance variables
  -- sensors (two encoders)
  private encMotor: Encoder;
  private encLoad: Encoder;

  -- actuators (one motor)
  private pwmMotor: Motor;

  -- PID control object
  private pid: PID;

  ...
end Controller

```

In this example, the associations are all one-to-one in that each object of the “from” end of the association points to exactly one object of the target class, and each object of the target class is associated with exactly one from the source class. This need not necessarily be the case, and the numbers on the association lines show the *multiplicity* of each association. It is possible for one end to show a range of values, including zero.

Information hiding and visibility is a key feature of object-oriented modelling, and so modellers are able to control the access rights to attributes. The **private** keyword preceding a definition indicates that the defined attribute can only be accessed by objects of this class directly; the **public** keyword indicates public access. It is sometimes convenient to access definitions without requiring an instance of the class in which the definition has been placed: these are indicated by the **static** keyword.

### 4.7.2 Instances of Classes and Constructors

Recall that classes are static, but their instances are dynamic, being created, living and dying within the running system. Given a class definition, we require a mechanism for creating an instance. This is done through the *new* expression. The creation of an instance of the `Motor` class would be modelled as follows:

**Fig. 4.2** UML class diagram illustrating different association multiplicities



```
new Motor()
```

where the list of arguments is empty, as indicated by the “()” part. If data are required in order to create the new instance, we define a *constructor* for the class with a number of arguments. Constructors are operations that create new instances of that class. They have the same name as the class itself, and their return type must be the same as the class. For example, the Controller class has a constructor defined as follows:

```
public Controller: nat1 * Encoder * Encoder * Motor ==> Controller
Controller(f, encm, encl, motor) == (
  -- set sampletime / period
  sampletime := 1/f;
  period := floor (sampletime * 1E9);

  -- set instance variables
  encMotor := encm;
  encLoad := encl;
  pwmMotor := motor;

  -- initialise instance variables
  pid := new PID(k, tauI, tauD);
  pid.SetSampleTime(sampletime);
);
```

The constructor is executed when a new instance of the class is created. For example, the creation of an instance of Controller might be expressed as follows:

```
new Controller(50, encMotor, encLoad, pwmMotor)
```

### 4.7.3 Optional Types and Association Multiplicities

As indicated in Sect. 4.7.1, associations can have many different multiplicities. For example, inside the Controller class, a reference to the CycloidGenerator

library is used, as shown in Fig. 4.2. This is optional: we expect there to be at most one cycloid generator or none at all, as indicated by the  $0 \dots 1$  in Fig. 4.2. In VDM, this is modelled by the use of an *optional type* for the target reference. A type is made optional by adding an extra value `nil`, which is frequently used to indicate a special error value or the absence of a normal value. For example, while the type `nat` contains all the natural numbers, the type `[nat]` represents all the natural numbers plus `nil`. In our cycloid generator example, we use an optional type for the instance variable representing the association

```
instance variables
  -- signal generator
  sp_gen: [CycloidGenerator] := nil;
```

Although we do not illustrate them here, a full range of multiplicities can be described using the collection types and invariants.

## 4.8 Concurrency

As we come to model more complex controller structures, it becomes important to have a natural way of modelling the simultaneous or *concurrent* execution of several processes that may occasionally *synchronise* with one another. In VDM, concurrent computations are modelled as *threads*, as described in Sect. 4.8.1. Synchronisation is achieved using a *permission predicate* approach described in Sect. 4.8.2.

### 4.8.1 Threads in VDM

Threads can be *procedural* or *periodic*. Procedural threads are simply described as a *statement* as in the body of an explicitly defined operation. Periodic threads describe computations that occur repeatedly. In order to specify such a thread, we define the operation that is to be invoked and the time period that should elapse between invocations. In the torsion bar `Controller` class, we would want to set the `Step` operation going with a fixed period:

```
thread
  -- define periodic thread
  periodic (period, 0, 0, 0) (Step)
```

The `Step` operation is called every `period` nanoseconds. Besides the period, the other parameters of a thread specification are, respectively, the jitter (the amount of variance allowed around the invocations of the operation), the delay (minimum

permitted time between two occurrences of the operation) and offset (the absolute time of the first occurrence of the event). In our example, we simply specify that `Step` is to be performed exactly at the times `0`, `period`, `2*period` etcetera. Threads are started explicitly by a `start` statement, often from an instance of a special class (conventionally called `World`), which sets a simulation going, as in our example:

```
class World

operations
  -- run a simulation
  public run: () ==> ()
  run() == (
    start(TorsionBar`ctrl);
    start(TorsionBar`monitor);
    start(TorsionBar`user);
    block();
  );
end World
```

Note that the `monitor` and the `user` also have threads of their own that are started here.

#### 4.8.2 Synchronisation of Threads in VDM

If we permit multiple threads, we also have to manage their synchronisation. In VDM, this is done by means of *permission predicates*, which provide guards that must be satisfied for an operation to be permitted to run. These are not the same as preconditions: failure of a permission predicate causes the thread to be blocked until the predicate is satisfied, whereas failure of a precondition means that the operation's outcome is undefined. Permission predicates can be logical assertions over instance variables, but may also be assertions about the history of activations and completions of the operations or the queues of operations waiting to execute. One of the most common types of permission predicate ensures mutual exclusion of running operation invocations. Such **mutex** constraints may be included in the Controller class:

```
sync
  mutex(Visit, Step)
  mutex(Visit)
  mutex(Step)
```

so at any point, at most one invocation of any of these operations will ever be allowed to be started. In a rather technical example, in the `World` class, a permission predicate is used to suspend the execution of the `block` operation:

```

operations
  -- wait for simulation to finish
  block: () ==> ()
  block() == skip;

sync
  per block => false;

```

This is used to block the execution on purpose. Note that the `skip` statement simply indicates that no action is required.

## 4.9 Modelling Systems

We have examined at some length the facilities for modelling data and functionality, structuring complex models and describing and constraining concurrency. Experience suggests that, if modelling a controller at a very high level of abstraction, relatively few of these features are required. However, as we move closer to a detailed description of a potentially complex, moded and multiprocessor controller, more of the features of the language come into play. This has been reflected in the increasing capability and complexity of the examples.

At some point in detailed design, we may wish to model the split between the computer hardware element of the digital controller package and the software, for example, in order to explore a design space of different processor characteristics or consider the potential of different controller software structures on specific computers. In VDM, we can model a computing hardware architecture consisting of one or more CPUs (processors) and connected by communications BUSES. This enables us to experiment more readily with alternative control architectures on the DE side than is the case using CT formalisms alone.

At the top level of the DE model in VDM, we describe the whole computing system by means of a special system definition which is similar to a class definition, except that it may additionally use the CPU and BUS abstractions to model the computing infrastructure and describe the deployment of elements of the control model onto CPUs. For example, in the `TorsionBarBaseline` model, we can construct a `TorsionBar` system class, within which the architectural components of the system are defined:



```

system TorsionBar

instance variables
  -- sensors (two encoders)
  public static encMotor: Encoder := new Encoder();
  public static encLoad: Encoder := new Encoder();

  -- actuators (one motor)
  public static pwmMotor: Motor := new Motor();

  -- controller object ~ 50Hz
  public static ctrl: Controller :=
    new Controller(50, encMotor, encLoad, pwmMotor);

  -- monitor object ~ 60Hz
  public static monitor: Monitor :=
    new Monitor(60, ctrl, encLoad);

  -- user object ~ 10Hz
  public static user: User := new User(ctrl);

```

Declaring these objects as public means that they can be accessed from anywhere in the model; for example, the controller object is accessed as `TorsionBar`ctrl`. These objects must be declared **static** as it is not possible to create instances of the **system** class. Only one can exist in the model, and it is implicitly instantiated at start-up. Having defined the architectural elements of the system, we can introduce the CPUs and communications infrastructure. In our example, we consider two CPUs connected by a single communications BUS. The definitions are given as follows:

```

instance variables
  cpu1 : CPU := new CPU(<FP>, 200E6);
  cpu2 : CPU := new CPU(<FP>, 200E6);
  bus1 : BUS := new BUS(<FCFS>, 115E2, {cpu1, cpu2});

```

Each CPU is configured with parameters indicating its scheduling policy (Fixed Priority or First-Come-First-Served) and its capacity in instructions per second. Each BUS is configured with a scheduling policy, its capacity (bandwidth in bytes per second) and the set of connected CPUs. In order to allow some freedom over which parts of a DE model are deployed to CPUs and which continue to be executed in a “perfect” environment, we assume a further *virtual CPU* with infinite capacity (i.e., executions consume no simulated time) which is connected with all other CPUs by a *virtual BUS* over which communication also takes no simulated time.

At any point in a model, the current simulated time can be accessed (by the **time** keyword). When any construct is executed, increments are made to this time. Default values for the increments can be overridden by a *duration statement*, which enables time to be incremented by a fixed amount (in nanoseconds) or a *cycles statement*, which enables the increment to be given in terms of processor cycles, so

that the actual increment depends on the CPU speed onto which the functionality is deployed.

The deployment of the system elements to CPUs is done in the system's constructor. Suppose that, for the `TorsionBar` system, we wish to deploy the controller to one CPU and the monitor to another:

```

operations
public TorsionBar: () ==> TorsionBar
  TorsionBar() == (
    -- deploy controller
    cpu1.deploy(ctrl, "TorsionBarController");
    cpu2.deploy(monitor, "Monitor")
  )

```

The elements not deployed to specific CPUs are allocated automatically to the virtual CPU and so are assumed to compute in zero time for the purposes of this model. This is useful for those elements of the model that represent the environment of the system.

A concurrent computation is said to be an *asynchronous* computation if the caller need not stop to await a response after issuing a call to another operation. Instead, the called operation will proceed in another thread. This is particularly relevant if the call goes over a bus because the called operation resides in an object allocated to a different CPU from that of the caller. We can define an operation as asynchronous in VDM using the **async** keyword before the operation name; such operations cannot return a value.

## 4.10 Conclusion

We have introduced the many forms of abstraction available to the DE modeller who wishes to design a controller. These include basic data that can be restricted by data type invariants, facilities for defining new types that are meaningful in the context of the control application, operations that work on persistent data and facilities for both implicit (pre/post-condition) and explicit (algorithmic) specification of control functionality. These enable basic control to be modelled, as shown in `TorsionBar1-Minimal`. Structured types such as records and collections such as sets, mappings and sequences enable more sophisticated supervisory control to be modelled, as demonstrated in the `TorsionBar2-Visit` model. Structuring facilities for more complex controllers enable the description of architectural solutions to problems like safety control, as presented in the `TorsionBar3-Monitor`.

In Chap. 6, we will see the further use of object-oriented structuring concepts to facilitate the reuse of designs and patterns of design, especially in the DE side. All of the techniques introduced here have been applied in the case studies described in Parts II and III, and further technical details will be introduced as required. All the models used as examples are part of the distribution of the Crescendo tool.

# Chapter 5

## Support for Co-modelling and Co-simulation: The Crescendo Tool

Peter Gorm Larsen, Carl Gamble, Kenneth Pierce, Augusto Ribeiro,  
and Kenneth Lausdahl

### 5.1 Introduction

Having seen how to construct CT and DE models in the two previous chapters, we may now bring them together to form a co-model as described in Sect. 2.4. This will be illustrated by reusing the torsion bar example used in the previous two chapters. This chapter introduces the Crescendo tool and its support for co-modelling and co-simulation. It focuses on illustrating the main tool functionality, rather than acting as a user manual. More detailed documentation for the Crescendo tool, as well as the individual CT and DE tools, can be accessed online for readers wanting further explanation.

The Crescendo tool can be downloaded from the website supporting this book, [www.crescendotool.org](http://www.crescendotool.org), along with additional material such as a user manual [60]. We recommended that you download and install the tool to gain the most from this chapter. The website also contains versions of all the examples introduced in this book. This chapter uses the *TorsionBar4-Baseline* co-model as an example, and so we suggest you import that so that you can better follow the information presented in this chapter.

---

P.G. Larsen (✉) • K. Lausdahl  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk); [lausdahl@eng.au.dk](mailto:lausdahl@eng.au.dk)

C. Gamble • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk); [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

A. Ribeiro  
d60, Aabyhoej, Denmark  
e-mail: [ars@d60.dk](mailto:ars@d60.dk)

The chapter begins with a demonstration of how to import the `TorsionBar4Baseline` model in Sect. 5.2. Then Sect. 5.3 explains how the CT and DE models are combined with a contract and linked to form a co-model. Afterwards Sects. 5.4 and 5.5 explains how co-simulation of the co-model can be carried out with and without additional scripts aimed at providing inputs dynamically during the co-simulation. Finally, Sect. 5.6 briefly demonstrates how to make changes to the CT and DE models from the torsion bar example. Finally, Sect. 5.7 provides a summary of the chapter.

## 5.2 Importing the Torsion Bar Co-model

Once the Crescendo tool has started and the *Welcome* screen has been closed, the overall layout of the screen shown looks like Fig. 5.1. Essentially it is divided into a menu bar at the top and five separate areas, called *views*. In the *Explorer* view, an overview of the different projects is shown (one co-model per project). The *Editor* view is used to view and edit files and includes syntax highlighting of keywords. If the file shown in the editor view contains structured information (e.g. VDM source text), the *Outline* view will provide a navigable outline of the contents, using icons to indicate the nature of the different definitions. The last two views (the *Simulation Engine* and *Console* views) are related to co-simulation, and we will come back to these in Sect. 5.4.

To import the `TorsionBar4-Baseline` co-model, click on the *File* menu entry and then selecting the *Import...* item in the menu and the import window will appear. Select the *Crescendo* item and then *Crescendo Book Examples*, selecting the `TorsionBar4-Baseline` co-model from the list. This is shown in Fig. 5.2. The imported model contains the CT model introduced in Chap. 3, the final version of the DE model introduced in Chap. 4, and the contract between them.

When the import has been completed, the `TorsionBar4-Baseline` project will appear in the Crescendo explorer with the structure illustrated in Fig. 5.3. The following sections explain the contents of these folders such that afterwards you will be able to create your own co-model projects.

## 5.3 Crescendo Contracts

As explained in Sect. 2.4, a contract consists of:

- Shared design parameters (SDPs)
- Shared (controlled) variables (written by DE, read by CT)
- Shared (monitored) variables (written by CT, read by DE)
- Events (only considered in Chap. 11)

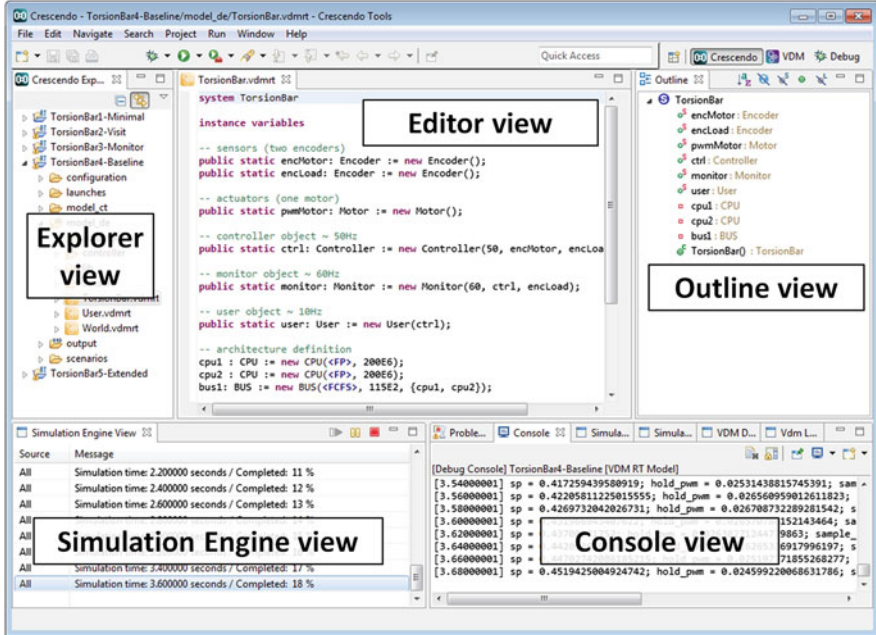


Fig. 5.1 The main Crescendo window

For the `TorsionBar4-Baseline` co-model, the controller needs to monitor the encoders for the motor (`enc_motor`) and the load (`enc_load`), and control the motor via a pulse-width modulated signal to an amplifier (`pwm_amp`, see also Sect. 3.5.2). This is recorded in the contract as:

```
-- monitored variables
monitored real enc_motor;
monitored real enc_load;

-- controlled variables
controlled real pwm_amp;
```

Note that lines prefixed by a “-” sign are comments to help the reader. This contract information can be found in the `contract.csc` file under the `configuration` folder. All values exchanged between DE and CT simulators are numbers (integers and reals), so if the model requires limitations on these, invariants should be defined on the DE side. It is also possible to exchange one- or two-dimensional matrices of numbers using a `matrix` keyword. This is not necessary in the current example, but more information can be found in the user manual [60].

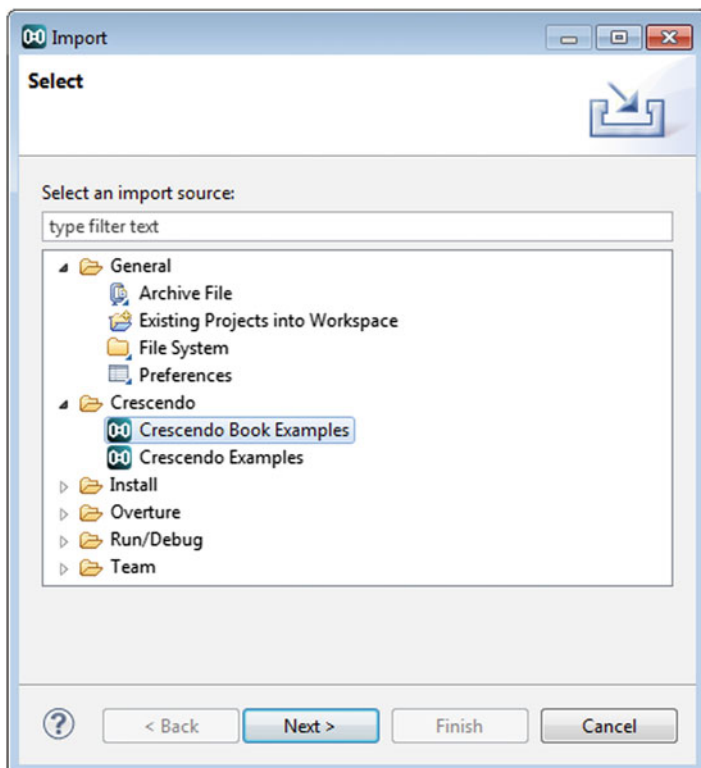


Fig. 5.2 Importing the TorsionBar4-Baseline co-model

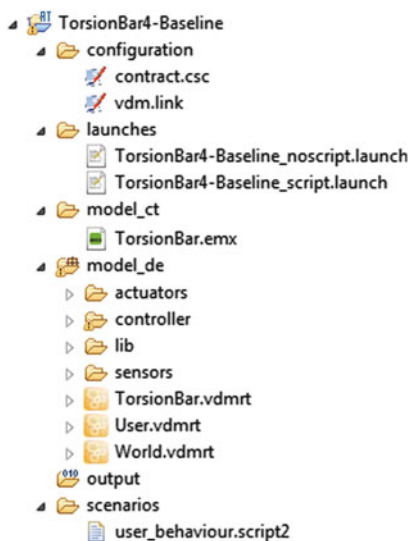


Fig. 5.3 TorsionBar4-Baseline co-model project structure

The contract also contains a SDP called `ENC_COUNTS`. Recall from Chap. 2 that SDPs are common to both models and are used when a parameter must be the same in both constituent models. SDPs are commonly used to define tunable parameters in a co-model. In the case of the torsion bar example, the SDP called `ENC_COUNTS` is used to indicate the number of pulses per revolution generated by the encoders. This parameter is used in the encoder blocks of the CT model to calculate the encoder count. This parameter is also needed by the controller in order to decode the count to determine the positions of the load and motor discs.

```
-- shared design parameters
sdp real ENC_COUNTS;
```

### 5.3.1 Introduction to the VDM Link File

In addition to the information provided above, it is necessary to connect these shared variables to elements in the DE and CT models. However, this is done differently in these two cases. For the DE side, there needs to be a VDM link file that makes the connection from the contract elements to their statically declared counterparts in the DE model. The syntax for these connections looks like:

```
input name = location
output name = location
```

The *name* refers to the name of the variable in the previous part of the contract and the *location* needs to be an object field path from the `system` class defined in the DE model. Thus, the *location* starts with the name of the `system` class and dots to instance variables derived from here. For the torsion bar example, this is done as follows:

```
-- linking of the monitored variables
input enc_motor = TorsionBar.encMotor.val;
input enc_load = TorsionBar.encLoad.val;

-- linking of the controlled variables
output pwm_amp = TorsionBar.pwmMotor.val;
```

This linking information resides in the `vdm.link` file which is also in the configuration directory. Note how each *monitored* variable is considered an input while *controlled* variables is considered an output. The syntax used here indicates the direction first, followed by the name of the variable. After that an equal sign is used to link it to the variable it will be mapped to inside the DE model. Here, `TorsionBar` is the **system** class (see Sect. 4.9) and inside that `encMotor` and `encLoad` are both statically declared encoder objects (of the `Encoder` class). Inside the `Encoder` class an instance variable called `val` of type **real** is declared and that is what is referred to here. The same applies to the **output** where the `TorsionBar` **system** has a statically declared instance variable called `pwmMotor` as an instance of the `Motor` class. Inside the `Motor` class there is also an instance variable called `val` of type **real** declared and that is what is referred to here. Thus, the right-hand-side of the equality sign shows the path from the top-level system down to the variables that needs to be exchanged with the CT side. We will illustrate how the linkage is done from the CT side in Sect. 5.3.2 below.

A similar connection must be made for the SDP. In the example, the SDP called `ENC_COUNTS` is mapped to a value of the same name in the `World` class of the DE model.

```
-- linking of shared design parameters
sdp ENC_COUNTS = World.ENC_COUNTS;
```

A fourth keyword can also be used in link files. The **model** keyword allows a script to modify an instance variable on the DE side during co-simulation. As above, only instance variables that are present statically (typically from the objects declared in the **system** class) can be used here. In this case, the name `input_trigger` is mapped to the `trigger` instance variable inside the `user` declared in our `TorsionBar` system. This looks like:

```
-- allow script to access trigger variable in User
model input_trigger = TorsionBar.user.trigger;
```

This instance variable allows a script to initiate user behaviour in a co-model. We will return to the use of `input_trigger` in Sect. 5.5.

### 5.3.2 Global Variables in the CT Model

To open the CT model located in the `model_ct` folder, double-click its file name (in this case `TorsionBar.emx`). This will automatically open the model in



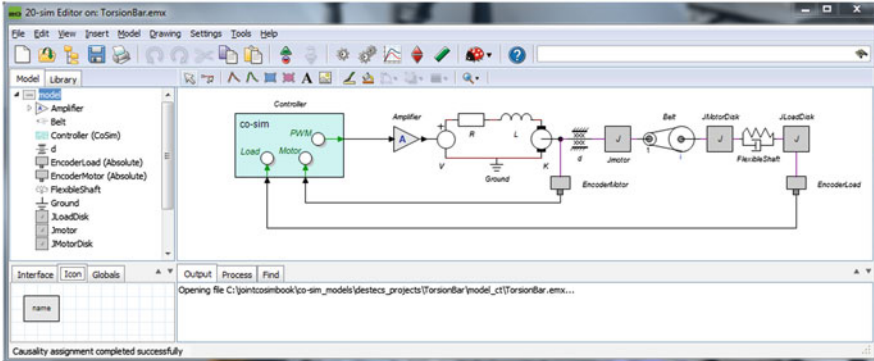


Fig. 5.4 The CT model of TorsionBar4-Baseline inside 20-sim

20-sim externally to the Crescendo tool as shown in Fig. 5.4. Note how the left-hand-side of 20-sim provides a navigable outline of the components that are included in the example. The main canvas shows the top level of the model, as presented back in Fig. 3.34.

For the CT side, the linkage to the elements included in the contract is done differently. Here one needs to declare the variables under the **externals** heading. Each variable also needs to be declared as being **global** in the CT model, and then we indicate for each of them whether they are being imported into or exported from the CT model.<sup>1</sup> Note that variables exported from the CT model will be input to the DE model and vice versa. Double-clicking on the Controller block (see Fig. 5.4) will reveal the linkage statements:

```
externals
  real global import pwm_amp;
  real global export enc_motor;
  real global export enc_load;
```

The names in the CT model must be an exact match with the names from the contract. The names must also be declared **global**, which allows them to be accessed without using the name of the block in which they are declared (e.g. the 20-sim internal name Controller/pwm\_amp is represented as pwm\_amp).

The SDPs need to be declared in a similar fashion. They must be declared under the **parameters** heading. In addition, SDPs need to be declared **global** and marked as being ( 'shared' ). This is done for the torsion bar example as follows:

<sup>1</sup>The 20-sim tool uses this principle to interface with other tools as well.

```

parameters
  real global ENC_COUNTS ('shared');

```

In order to connect the input ports of the `Controller` block in 20-sim to the external variables, it is necessary to write this in either a `code` or an `equations` section. For example:

```

code
  enc_motor = EncoderMotor; // export encoder values
  enc_load = EncoderLoad;

  PWM = pwm_amp; // import pwm

```

Note that `EncoderMotor`, `EncoderLoad` and `PWM` are the names of ports in the `Controller` block in 20-sim. These ports are used to connect the shared variables to the rest of the CT model.

## 5.4 Starting a Co-simulation

In order to start a co-simulation, the `Debug Configurations` window has to be opened. This is achieved by clicking the small arrow next to the debug button (the one with the small bug on it as shown in Fig. 5.5) at the top menu line and then selecting it. In a `Crescendo` setting, debug configurations are also called *launches*.

When `TorsionBar4-Baseline_noscript` is selected, the window shown in Fig. 5.6 should appear.

For a debug configuration, the following aspects need to be configured:

- A project must be selected for the co-simulation (in the case of Fig. 5.6 this is `TorsionBar4-Baseline`);
- The paths to the DE and the CT models must be set. The DE path will already be filled out, but the 20-sim file you wish to use must be selected (in this case `TorsionBar.emx`). This allows multiple CT models to be present in one `Crescendo` project;
- A script may be selected. Scripts are covered in the next section; and
- The simulation time (in seconds) must be set.

Once the above settings are configured in the `Main` tab, there is enough information to launch a co-simulation.

The debug configuration also contains six other tabs to allow many other options to be set. The `Shared Design Parameters` tab allows the values of the SDPs

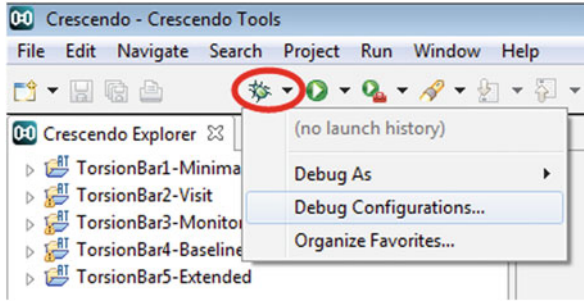


Fig. 5.5 Debug icon highlighted

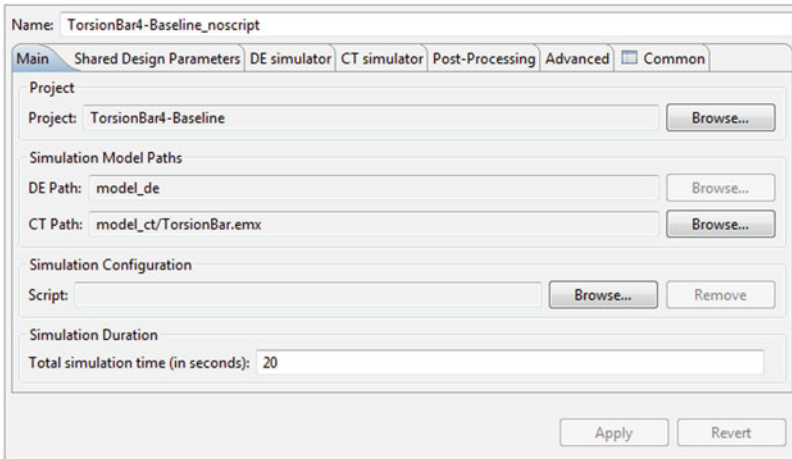


Fig. 5.6 Configuring a co-simulation

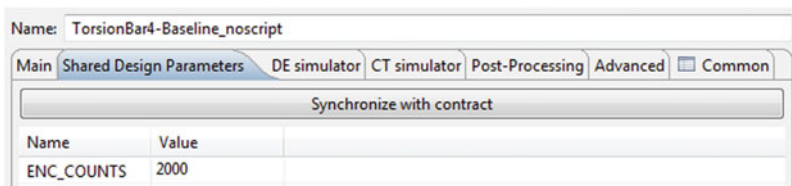


Fig. 5.7 Setting a shared design parameter

defined in the contract to be set before co-simulation. In the case of the torsion bar example, the ENC\_COUNTS parameter should be set to 2000, as seen in Fig. 5.7. The other five will not be explained here.

To start a co-simulation, press the Debug button. Now the Crescendo co-simulation engine will start executing the co-model with the DE and CT simulators taking turns. In the Simulation Engine view, progress of the

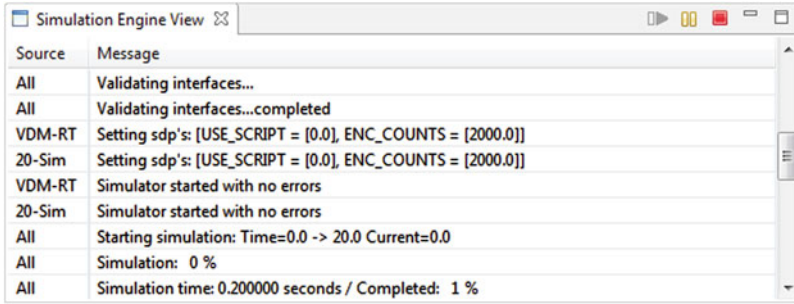


Fig. 5.8 The Simulation Engine view, including the square stop button (*top right*)

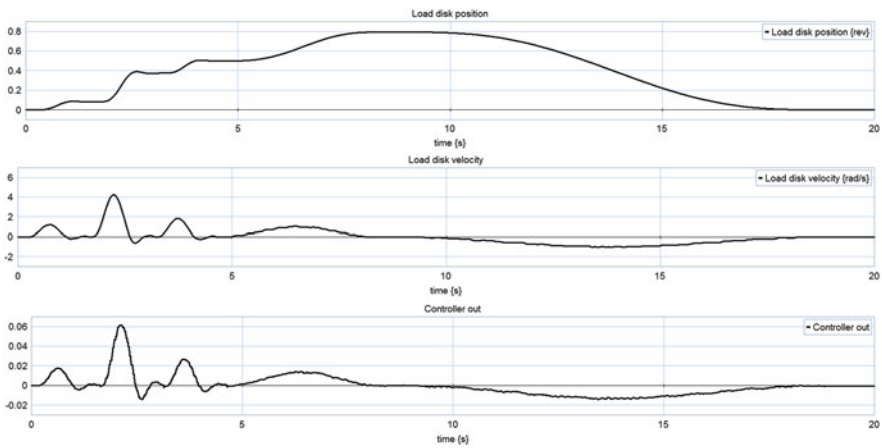


Fig. 5.9 Plot of selected variables against time (from TorsionBar4-Baseline)

co-simulation is reported as a percentage of simulation time passed. If you for some reason wish to abort a co-simulation, note that after starting it you can always press the stop button (square button in the upper part of the Simulation Engine view as shown in Fig. 5.8). Next to the stop button there are two parallel bars which act as a pause button. To the left of that there is a small arrow pointing to the right (grayed out in Fig. 5.8 but if it can be activated it would be green) which is a play button to be pressed again after a paused co-simulation.

Both the DE and the CT simulators have their own ways for a model to report progress of a simulation. On the DE side, progress is typically made using traditional output to the standard console, but it is also possible to develop project-specific Java Graphical User Interfaces (GUIs) that can be used to demonstrate the progress of a simulation. On the CT side, progress of a simulation can be shown using graphs such as Fig. 5.9, but in many cases the most valuable way of showing progress is using a 3D visualisation of the simulation as shown in Fig. 5.10. In particular, for stakeholders who are less technically skilled, this kind of animation can be

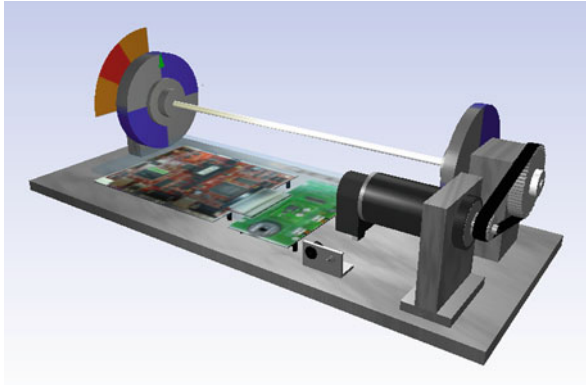


Fig. 5.10 3D animation of the torsion bar during simulation

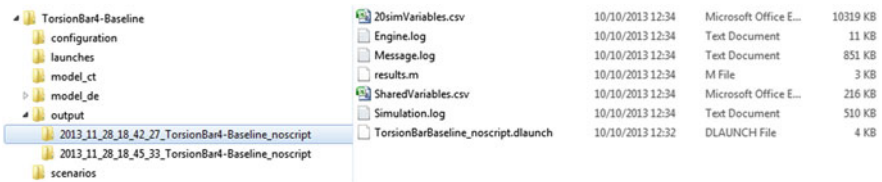


Fig. 5.11 Example structure of the results from a single co-simulation

very valuable in understanding the consequences of different design decisions. One particular benefit of the 3D visualisation is that it can be replayed after the simulation (real time, sped up or slowed down) or saved as a video without having to re-run the simulation.

In addition to the live outputs from the DE and CT tools, Crescendo also saves results from every simulation run within the `output` directory of the co-model project. The directory name includes both the date and the time that the simulation was run, in addition to the name of the debug configuration used (see Fig. 5.11). Three of the files saved are of particular interest. In the CT Simulator tab of a debug configuration, it is possible to select variables within the CT model that are to be logged. If this is done, then they are saved in the `20SimVariables.csv`. It is also possible to log the variables shared via the co-model contract. This is selected by ticking `Enable Logging` in the `Advanced` tab of the debug configuration. This data is saved in `SharedVariables.csv`. A `.dlaunch` file is also added to this directory, allowing the simulation to be repeated with the same settings if required.

## 5.5 Using Scripts and SDPs

When performing a co-simulation, it may be convenient to influence its behaviour “from the outside”. This might be done to disturb the system under test to observe the results or to explore different scenarios. Creation of these scenarios is possible with scripts. The Crescendo tool supports a scripting language called the Crescendo Scripting Language (CSL).

In general, a script consists of a collection of *condition* and *action* pairs. The conditions are written as logical expressions using keywords such as **time** as well as the variables exchanged between the DE and CT simulators. All the actions get reevaluated every time control is shifted from the DE to the CT side and vice versa (see Sect. 13.4 for more detail on the semantics). Thus, since there are many such shifts, the time taken to perform the co-simulation of the co-model is significantly increased when using scripts. As an alternative to the **when** keyword, we could equally well have used the **once** keyword. Semantically the only difference is that with **once** the action will be carried out at most once whereas the **when** keyword will invoke the action whenever the condition is true. The *action* part starts by an indication of whether the action is related to the DE or the CT side (in this case, the DE side).

The TorsionBar4-Baseline project contains a script called under the *scenarios* folder. The TorsionBar4-Baseline\_script launch is used to illustrate scripting. This launch has the script file already selected.<sup>2</sup> The simulation can be initiated by clicking on the Debug button as before. The script file is called `user_behaviour.script2` and opening it will show the following code:

```
when time >= 0.3 and time <= 0.4 do
(
  de boolean input_trigger := true;
);
```

The effect of this script is that when the time is between 0.3 and 0.4 s the value of a variable called `input_trigger` is set to **true** (the **de** part indicate that it comes from the DE side). Recall from the contract description earlier in Sect. 5.3.1 that the variable `input_trigger` is mapped to a variable with the location `TorsionBar.user.trigger`. So in effect, when the time is between 0.3 and 0.4 s, the `trigger` variable in the `user` object declared in the `TorsionBar` class is set to **true**.

---

<sup>2</sup>Note that an additional SDP, `USE_SCRIPT`, is used in this model. This prepares the constituent models to run a script and must be set to 1. This is, however, a modelling choice and not a requirement.

During simulation, the `Step` operation within the `User` class is executed periodically. The periodic operation checks for the value of `trigger` changing to true, after which it invokes the `Visit` operation of the controller (mimicking user behaviour). The controller then visits them as shown in Chap. 4. The core definitions for the `User` class are:

```
class User
...
operations
-- periodic operation
public Step: () ==> ()
Step() ==
  if trigger
  then (-- set angles to visit and reset trigger
        controller.Visit(ANGLES);
        trigger := false)
end User
```

As seen in Fig. 5.12, the first setpoint is changed right after 0.3 s, indicating that the script executed as expected. This output is generated by using the standard IO library class. Here the relevant part of the definition is:

```
Step() ==
...
  IO`printf("[%s] Setpoint changed to %s. Next change at %s.\n",
            [now, next_pos, next_time]);
...

```

## 5.6 Changing the Torsion Bar Model

In this section, we describe how to make some changes to the torsion bar model. We start by altering the CT model to represent it containing a larger, more powerful motor and then increase the mass of the load disk. These changes result in the load wheel exceeding a speed limit initiating an emergency stop and so we then describe how to input new parameters into the PID controller in the DE model to correct this undesirable behaviour.

```

[Debug Console] TorsionBarExtended [VDM RT Model]
[0.22000001] sp = n/a; hold_pwm = 0; sample_encm = 0; sample_encl = 0
[0.24000001] sp = n/a; hold_pwm = 0; sample_encm = 0; sample_encl = 0
[0.26000001] sp = n/a; hold_pwm = 0; sample_encm = 0; sample_encl = 0
[0.28000001] sp = n/a; hold_pwm = 0; sample_encm = 0; sample_encl = 0
[0.30000001] Setpoint changed to 0.08333333333333333. Next change at 1.80000001.
[0.30000001] sp = 3.008695960780686E-17; hold_pwm = 4.062926955231456E-17; sample_encm = 0; sa

```

Fig. 5.12 Triggering from a script during simulation

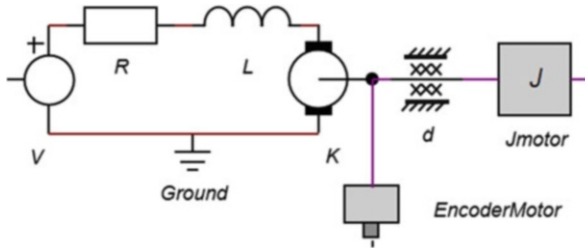


Fig. 5.13 Iconic representation of the motor

### 5.6.1 Adjusting the CT Model

We start by here by adjusting the parameters in the CT model to represent a larger electric motor. Figure 5.13 shows the iconic elements making up the motor and encoder model. To model a larger motor, it will be necessary to reduce the motor's resistance (R element) and increase the inductance (L element), motor constant (K element) and motor inertia (Jmotor).

The same method is used to effect a change in each of the parameters we need to change. Using the resistance as an example, first click on the icon of the R element to select it then right click to bring up a context menu. Select parameters from this menu to reveal the parameters/initial values editor, Fig. 5.14. To reduce the resistance of this element, click on the value for the row R and replace it with the new value, in this case a 1. We can now proceed to alter the values of the other elements, either by closing the parameter/initial values editor window and then selecting the next element, or you may select the next element to edit in the model hierarchy panel on the left-hand side. Doing the latter will reveal the parameters for this newly selected element, allowing them to be changed. You should edit the parameters of all four elements listed in Table 5.1, replacing the original values with the new values in the table.

Launching the co-simulation with these new parameters reveals that the new larger motor give a slightly smoother plot for the "load disk position" than with the original parameters, but otherwise the controller has ensured that the behaviour of the load disk is the same as before (Fig. 5.15). Inspired by this we might consider



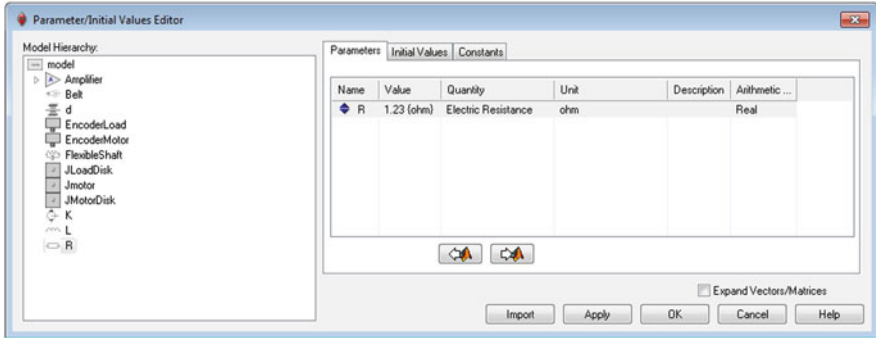


Fig. 5.14 Parameters/initial values editor view of the R element

Table 5.1 Parameter values for a larger electric motor

Element and parameter	Original value	New value
Resistance ( $R$ )	1.23 ohm	1 ohm
Inductance ( $L$ )	1.34 mH	2 mH
Motor constant ( $K$ )	38.9 m	60 m
Motor inertia ( $J_{motor}$ )	$6.76e^{-6} \text{ kg m}^2$	$1.276e^{-5} \text{ kg m}^2$

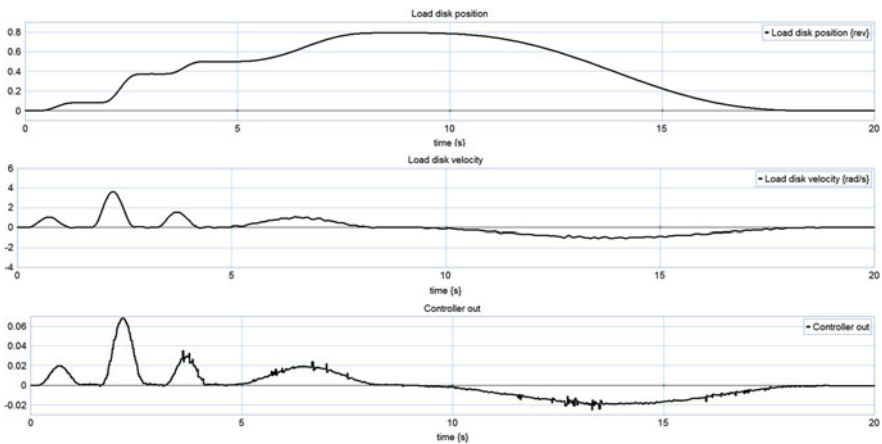


Fig. 5.15 Co-simulation results with a larger motor

whether the system could drive a more massive load disk now we have the larger motor attached. The current mass and dimensions of the load disk result in it having an inertia of  $0.00137 \text{ kg m}^2$ , this may be found by right clicking the `JLoadDisk` element and selecting the “Parameters” option. A larger mass leads to a great moment of inertia, so replacing the current value with  $0.01312 \text{ kg m}^2$  represents a load disk with a moment of inertia nearly one order of magnitude greater than the previous one.

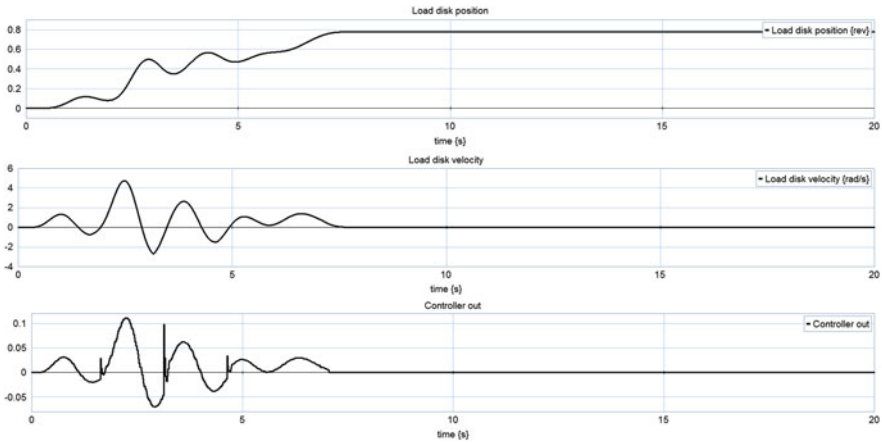


Fig. 5.16 Co-simulation results with a larger motor and larger load disk

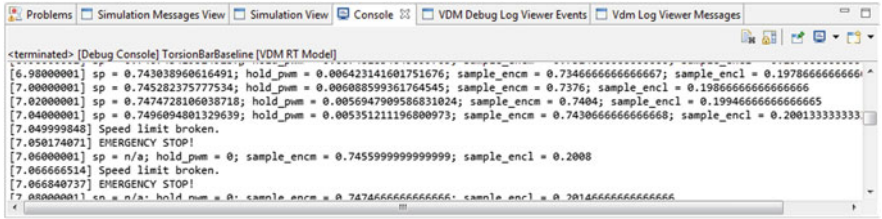


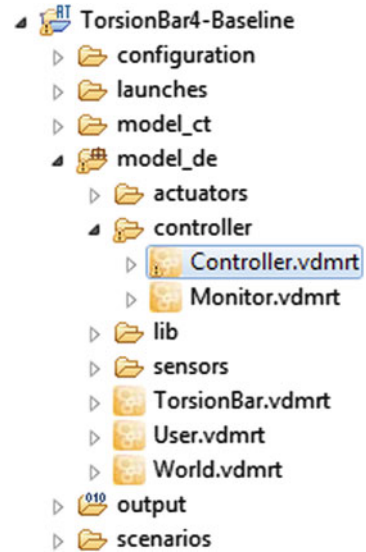
Fig. 5.17 Console output from simulation showing the emergency stop raised by the controller

Running the co-simulation with the more massive load disk shows that there is now a problem in the modelled system as the profile of load disk position stops changing after 7 s of simulation rather than returning to zero as expected, see Fig. 5.16. If we review the controller output in the console view, we see that at 7.05 s, the controller reported an emergency stop as one of the safety conditions has been breached (see Fig. 5.17). So if we are to continue using the larger load disk, then one option is to make a change to the controller to prevent the safety condition being broken.

### 5.6.2 Adjusting the DE Model

One way to correct the safety constraint breach introduced in Sect. 5.6.1 is to modify the parameters used in the DE controller. Within the DE controller, there exists a discrete time PID controller such as described in Sect. 3.6. This controller implementation utilises three parameters,  $K$ ,  $\tau_{auI}$  and  $\tau_{auD}$  to define its response

**Fig. 5.18** Location of the controller class in the project explorer



**Table 5.2** Control parameter values accounting for larger motor and load wheel

Control parameter	Original value	New value
$K$	1.0	0.2
$\tau I$	2E3	20
$\tau D$	0.05	0.189

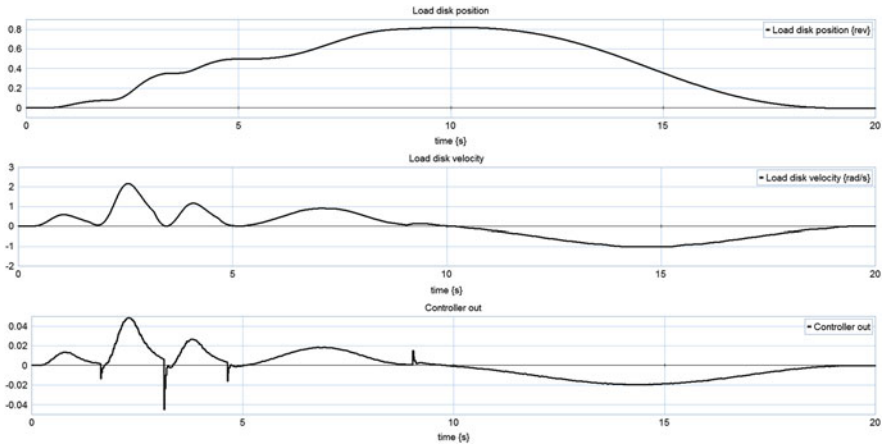
and by changing these we may alter the behaviour of the system such that the driven wheel does not breach the speed safety constraint.

To edit the parameters, we must first locate the Controller class in the project. The DE controller exists as a number of classes under the “model\_de” folder within the project in the project explorer (Fig. 5.18). Opening the “controllers” folder reveals the controller.vdmrt file we need to alter and double-clicking on the class name will then open it in the editor pane.<sup>3</sup> The  $K$ ,  $\tau I$  and  $\tau D$  parameters may be found by scrolling to near the bottom of this class in the **values** section of the class. The original and new values for all three parameters can be found in Table 5.2.

Once the values have been changed, the controller class must be saved for the changes to take effect. The tool indicates that unsaved changes exist by adding an asterisk to the start of the class name at the top of the editor pane. Saving can be performed by either selecting “save” under the file menu or by clicking on the single disc save icon on the tool bar.

With the new control values in place, the system can be co-simulated once again. This time it follows the setpoints without breaching the safety conditions (see Fig. 5.19).

<sup>3</sup>The small yellow triangle indicates that warnings are present in this file.



**Fig. 5.19** Graphical overview of changes of selected variables over simulation with the new control parameters in place

After co-simulation it is possible to inspect how well the simulation has exercised the DE part of the model. This can be valuable information in case one would like to ensure that a collection of scenarios are sufficient to execute each line of code in the DE model at least once. In order to get hold of this information, you first need to set a check-mark with the `Generate Coverage` in the `DE simulator` pane of the `debug configuration`. After the co-simulation is complete, there will be a new `coverage` directory in the `output` folder. If you also have the `LATEX` text processing system installed on your computer, you can furthermore generate a `pdf` file with the DE model including the test coverage information also with a table with coverage percentages per function/operation.

## 5.7 Conclusion

In this chapter, we have sought to show how the theory of co-modelling comes to life in the Crescendo tool. The tool provides a framework for constructing and editing VDM and 20-sim models setting up the contracts between them and running co-simulations. The ability to run scripts performing simulation scenarios, and the ability to set and experiment with design parameters enables the exploration of alternative designs. This process can be further systematised, as discussed in Chap. 10. In order to fully understand the material presented in this chapter, we strongly recommend obtaining and running the tools to appreciate their capabilities.

# Chapter 6

## Co-model Structuring and Design Patterns

Kenneth Pierce, Peter Gorm Larsen, and John Fitzgerald

### 6.1 Introduction

In the design of potentially complex embedded systems under software control, it is essential to have the ability to structure co-models in order to manage complexity and support re-use. This chapter looks in detail at the structuring of co-models and DE models in particular. The role of inheritance in object-orientation is introduced, along with the concept of a design pattern that records best practice in building models. Some design patterns for DE controllers are illustrated by extending the torsion bar example from that of Chap. 5. These include design patterns for sensors and actuators and for reuse of thread definitions. In addition, some insight is offered on how to build constituent models that can participate in DE-only and CT-only simulations, as well as co-simulations, which is relevant for Chap. 8 on creating co-models.

As described in the introduction to Chap. 4, there are three key reasons to use a discrete-event formalism such as VDM to model control software: using abstraction to suppress unnecessary detail, the high degree of rigour permitted by systematic analysis of such models and finally the ability to structure models into separate units of behaviour that can be more easily understood and analysed. There are many ways in which the DE formalism allows models to be structured. For example, functionality can be divided into separate, reusable functions or

---

K. Pierce (✉) • J. Fitzgerald  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk); [john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk)  
P.G. Larsen  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

operations.<sup>1</sup> Custom data types can be defined to allow data to be structured and manipulated. Comparisons can be drawn between the `struct` type of the C programming language and the record types of VDM, for example; however, the DE formalism has a more comprehensive set of ways to define data types. An extension of these two forms is that of object-orientation, where objects combine data and functionality into single units. Objects may then be passed round so that functionality and data can be accessed or manipulated using the interface offered by an object (as defined in its class).

Building a well-structured model brings several benefits. These include breaking down a solution into behavioural units that make the solution more tractable in construction, more understandable to others and more maintainable in the future. Splitting functionality also allows unit tests to be defined and permits division of labour between multiple parties. There are however a large number of equally valid ways to build a model that we might say is “well-structured” according to these criteria. In the context of our approach, we also want to structure DE models in a way that is conducive to collaborative modelling and analysis through co-simulation. Therefore, the primary purpose of this chapter is to present some particular ways of structuring DE models that we have found most useful in our approach.

The various structuring suggestions will be illustrated using an extension of the `TorsionBar4-Baseline` co-model called `TorsionBar5-Extended`. The extended model has the same observable functionality as the original, but has a more elaborate internal structure. Each of the structuring methods introduced will be motivated by some need not currently met in the baseline model. What we are mainly presenting here are what are known as *design patterns*. A design pattern is a way of presenting tried-and-tested methods for structuring and divisions of functionality derived from previous solutions to similar problems. The concept of design patterns in object-oriented languages will be introduced with a standard example, which will then be applied in the extended model. Design patterns also feature heavily in Chap. 9 on faults and fault tolerance.

This chapter continues with an introduction to the concept of inheritance in object orientation in Sect. 6.2, because it plays a major role in the remainder of the chapter and elsewhere in our approach. This is followed by a discussion of how and why to introduce generic interfaces for sensor and actuator objects in Sect. 6.3. The more general concept of design patterns is introduced in Sect. 6.4, with the standard *decorator pattern* applied in the `TorsionBar5-Extended` example. Next, Sect. 6.5 explains how threading functionality can be placed in an abstract class to permit reuse. This is followed by Sect. 6.6 discussing how both DE and CT models can be structured to allow both DE- and CT-only simulation and co-simulation using a single co-model. This section also includes guidance on the placement of the boundary between the DE and CT models in a co-model. This flexible structuring is particularly relevant for Chap. 8, which describes ways

---

<sup>1</sup>In other languages these can be alternatively called subroutines, procedures or methods.

to build up to initial co-models. Finally, Sect. 6.7 ends this chapter with a short summary.

## 6.2 Object-Orientation and Inheritance

The concept of *inheritance* is essential to object-orientation and is important to understand for the design patterns described later in this chapter. As initially described in Sect. 4.7, an object-oriented model is made up of one or more objects. Each object typically contains some data in the form of instance variables and functionality in the form of operations that may manipulate that data. The types of data and the functionality of an object are defined by its class. Multiple object instances can be created from the same class definition, where each object will have the same functionality.

Inheritance allows a class to be defined as a *subclass* of another class, creating an extended class. The subclass inherits all visible properties from the *superclass*, including instance variables and operations. Naturally, a subclass may define its own instance variables and operations as well. Through inheritance, a subclass inherits from the superclass:

- Its value and type definitions.
- Its instance variables, including all invariants that restrict the state.
- Its operation and function definitions.
- Its thread definition (see Sect. 4.8.1<sup>2</sup>).
- Its synchronisation definitions (see Sect. 4.8.2).

The subclass may also *override* operations of the superclass, allowing it to change their behaviour. Of course from a design perspective, the subclass has a responsibility to alter the operation in a sensible way. Note that pre- and post-conditions are not inherited by the subclass when overriding an operation, and as such there is no restriction on how the subclass overrides the operations. Copying the pre- and post-conditions to the subclass operation is good practice in situations where the form of the result is the same, but the way it is achieved is changed.

In certain cases, a class may not give a definition for an operation, by declaring the body of the operation as **is subclass responsibility**, in which case a subclass *must* override this operation before it can be instantiated and used. A class that contains any usages of the **is subclass responsibility** keyword are considered as an *abstract class*. Abstract classes cannot be instantiated as objects directly, but may serve as superclasses.

---

<sup>2</sup>If a class inherits from several classes, only one of these may declare its own thread (possibly through inheritance). Furthermore, explicitly declaring a thread in a subclass will override any inherited thread.

The concept of inheritance is demonstrated by the PID controller class used in the example in Chap. 4. This class inherits from a class called `DTControl` (Discrete-Time Control). The role of a PID object is to calculate an output for an actuator based on the error between the measured value and setpoint. For this purpose, it has an `Output` operation. It must also know the sample time (time since the last calculation) in order to calculate the integral and derivative elements, so for this purpose it has a `SetSampleTime` operation.

In fact, there is a family of four low-level control objects in the DT control library that inherit from `DTControl`; these are P, PI, and PD and PID. The full library family is shown in the class diagram in Fig. 6.1, where the arrow with an open arrowhead indicates inheritance. Thus, this diagram shows that `DTControl` is a general interface for the four other classes.<sup>3</sup> The setting and storing of the sampletime as an instance variable will be the same in all cases, so this is handled inside the `DTControl` class; however, the calculation of the output will differ across classes, so this is delegated to the subclasses using the **is subclass responsibility** keyword. This can be seen in the definition of the `DTControl` class:

```
class DTControl
instance variables

protected sampletime: [real] := nil;

operations

-- set the sample time used to calculate response
public SetSampleTime: real ==> ()
SetSampleTime(s) ==
  sampletime := s
pre s >= 0;

-- calculates output, based on the error
public Output: real ==> real
Output(err) ==
  is subclass responsibility;

end DTControl
```

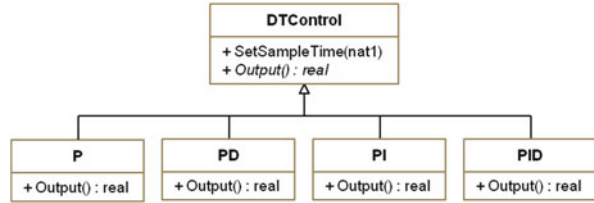
Note here that the `sampletime` instance variable is declared as **protected**, this means that it can be accessed by the class and all of its subclasses. This is in contrast to **private** (accessible only within a class) and **public** (accessible by any class in the model).

---

<sup>3</sup>Of course the P controller does not need to know the sampletime.



**Fig. 6.1** Class diagram of the discrete-time (DT) control library



Building the library not only allows for consistency across the four controller classes, it also allows the type of controller object used to be changed seamlessly. As long as the DE controller defines its instance variables using the `DTControl` type, any of the four subclasses can be instantiated using the `new` keyword and can be used by the DE controller without having to alter the model in any other way. The selected controller object can also be instantiated elsewhere and passed to the DE controller. This seamless switching idea is very useful for testing alternatives during Design Space Exploration (DSE) activities, described in greater detail in Chap. 10. It also enables some more complex behaviour, as we shall see later in this chapter in Sect. 6.4.2.

### 6.3 Interfaces for Sensors and Actuators

In the previous section, we saw how the interface to the proportional control objects was extracted and placed in the `DTControl` superclass. A similar style can be applied to sensor and actuator objects. Recalling the object-oriented torsion bar controller from Chap. 4, the encoder and motor shared variables are accessed through encoder and motor objects. The `Encoder` class defines an instance variable that is linked to the co-simulation contract and provides a `Read` operation that returns this value:

```

class Encoder

instance variables

-- this value will be set through the co-simulation
protected val: real := 0;

operations

-- read the current value of this sensor
public Read: () ==> int
Read() == return floor val;

end Encoder
  
```

The `Motor` class is very similar, though in addition it has a `Write` operation that allows a value to be set. This style for sensor and actuator objects is common to the DE models of our approach. The interface here is a `Read` operation for sensors and `Read` and `Write` operations for actuators. In a similar manner to `DTControl` above, this interface can be extracted and placed in an superclass. The general interface to the `Encoder` class is as follows:

```
class ISensorInt
-- this sensor yields a single integer value

operations

-- read the current value of this sensor
public Read: () ==> int
Read() == is subclass responsibility

end ISensorInt
```

This interface appears in the `TorsionBar5-Extended` co-model, along with an altered version of the `Encoder` class that extends this interface:

```
class Encoder is subclass of ISensorInt

instance variables

val: real := 0;

operations

-- read the current value of this sensor
public Read: () ==> int
Read() == return val;

end Encoder
```

The same style is followed for the `Motor` class in the `TorsionBar5-Extended` co-model as well, with the `IActuatorPWM` class forming the interface.

At this point, it is worth noting the suggested naming conventions for sensor and actuator interfaces and implementing classes. Interface names should begin with the letter `I` (for Interface) followed by the word `Sensor` or `Actuator` and ending with the type that is written or read. So in the example above, the encoder interface yields an integer value, and the motor interface takes a `PWM` value in the range  $(0,1)$ . Implementing subclasses should then be given a useful name indicating what the sensor or actuator represents, for example, `motor`, `encoder`, `IR sensor`. In models of more complicated systems, there will typically be more sensor and actuator types than the two in our small example. The types of values accessed through sensors

or modified through actuators may be the same (real numbers will be common, for example). Where interface types are the same, the interface can be reused.

Here we are recommending that sensors and actuators are built in a certain way, following objected-oriented practice. While this might not appear particularly important for such simple objects, it has two main benefits. First, different sensor and actuator classes can be defined for the same interface. If the various parts of the controller are defined using the interface for access, then the alternative sensor and actuator objects can be substituted seamlessly.

For example, if a model is built using the DE-first approach (see Sect. 8.5), an encoder object might need to calculate the value from a DE model of the plant. Once the switch to co-simulation is made, however, the calculation is done at the CT side, so the encoder object does not need to perform a calculation. This warrants the use of a different encoder class, yet has the same interface. Of course, a component that uses the interface has an expectation that an object with that interface will behave in a certain way (e.g., that a rotary encoder will yield information about the rotation of the component to which it is attached), so it is important to bear this in mind when producing classes that implement the interface.

The second and primary benefit, extending from the first, is that it enables sensor and actuator objects to be incorporated into a *design pattern*. In the software world, a design pattern is a description of an approach to building some part of the software, based on successful solutions built previously. Later in this book, we describe some design patterns building on experiences with our approach.

To better acquaint readers unfamiliar with software design patterns, the next section explains the principles of design patterns in more detail. It describes a common design pattern from the world of object-oriented software and shows how it can be applied in the `TorsionBar5-Extended` co-model. Further design patterns appear in Chap. 9 on faults and fault tolerance, and all design patterns described in this book are given in full in Appendix C.

## 6.4 Design Patterns

A *design pattern* is a template that outlines a possible solution for a category of problems. 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. [39], which relates to object-oriented software, but which can be applied more generally. The approach originally drew inspiration from the field of architecture [1].

The general form of each design pattern includes a *name*, a *problem* description, a *solution* description and a description of the *consequences* of its application [39]. Note that since our approach is relatively new, experience relating the consequences arising from some design patterns described in this book are limited; however, we provide this information where possible.

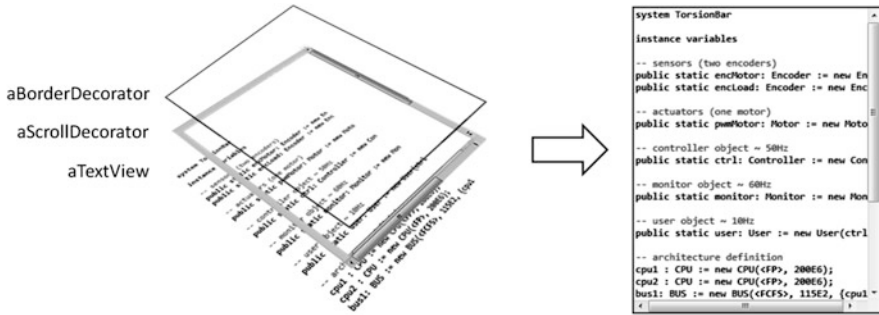
Where design patterns in this book are described in full (such as in Appendix C), we adapt from Gamma et al. [39] and provide the following information for each design pattern description:

- Name:** An identifier that conveys the essence of the design pattern succinctly.
- Intent:** What does the design pattern do? What problem does the pattern address? What is the rationale or intent?
- Motivation:** A scenario that illustrates how the design pattern solves the problem and can help interpret the rest of the description.
- Structure:** A graphical representation of the elements of the solution, for example, a UML class diagram.
- Application to DE domain:** Model fragments and guidelines on how the solution can be realised within VDM (where applicable).
- Application to CT domain:** Model fragments and guidelines on how the solution can be realised within 20-sim (where applicable).
- Use in examples:** If a design pattern is used in the Crescendo examples, it will be listed here (where applicable).
- Related patterns:** Descriptions of other design patterns which are closely related (where applicable).
- Also known as:** Other well-known names for the design pattern (where applicable).

### 6.4.1 *The Decorator Pattern*

To help those readers who may be less familiar with design patterns and their application, we now look at a standard pattern from the literature. We then show the application of this design pattern in the `TorsionBar5-Extended` example. The design pattern we describe is the *decorator pattern*, which presents a flexible alternative to subclassing to add additional functionality to objects dynamically. The following partial description is taken from Gamma et al. [39], and we direct the reader there for a full description:

- Intent:** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Motivation:** It is sometimes desirable to add functionality to individual objects and not to an entire class. For example, visual components in a graphical user interface (GUI) library may need scrollbars adding dynamically if the content becomes too large to fit the screen. In this case, the visual component object can be enclosed in a scrollbar object that adds this additional scrolling functionality. The enclosing object is called a decorator. The interface approach described previously in this chapter is used to allow seamless addition and removal of extra functionalities dynamically.



**Fig. 6.2** Visual representation of decoration of a textual view component with scrollbar functionality

**Applicability:** Use the decorator pattern:

- to add responsibilities to individual objects dynamically and transparently;
- for responsibilities that can be withdrawn; or
- when extension by subclassing is impractical.

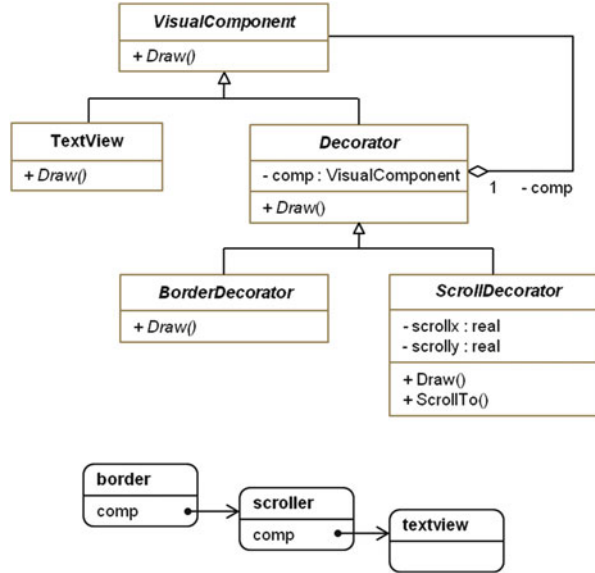
The full description of the decorator pattern in Gamma et al. [39] gives a rich example involving decoration of GUI components. Here we describe part of their example where a textual view component is decorated with a scrollbar and a border. Consider the visual representation of this idea in Fig. 6.2. Here, the responsibility of the `aTextView` component is to render the text. It knows about fonts and layout. The `aScrollDecorator` is responsible for rendering the scrollbars and moving the `aTextView` around in response to the user moving the scrollbar. The `aBorderDecorator` is responsible for drawing the dark border around the component.

In order to enable this functionality, there are two key parts to this design pattern. First, a common superclass, `VisualComponent`, is defined that requires all subclasses to implement a `Draw` operation to render the contents of the component. Second, an abstract `Decorator` holds a reference to a decorated object of the type `VisualComponent`. Finally, the concrete decorators, `BorderDecorator` and `ScrollDecorator`, implement the `Draw` operation.

Figure 6.3 contains a class diagram showing the relationships between the classes statically. It is often useful to see how the objects instantiated from these classes are related at runtime as well. This can be shown in an *object diagram*, where the blocks with rounded edges are objects and the arrows show references between objects. An object diagram is given in the lower half of Fig. 6.3. This shows a border object decorating a scrollbar object, which in turn decorates the text view object.

Figure 6.3 contains a class diagram showing the relationships between the classes statically. It is often useful to see how the objects instantiated from these classes are related at runtime as well. This can be shown in an *object diagram*, where the blocks with rounded edges are objects and the arrows show references between objects.

**Fig. 6.3** Class and object diagram for a GUI-oriented version of the decorator pattern



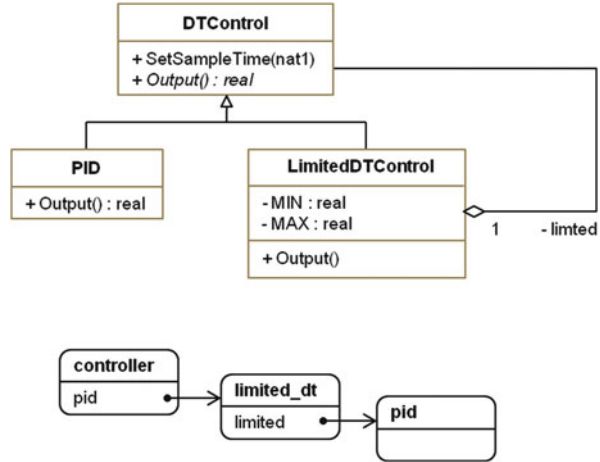
An object diagram is given in the lower half of Fig. 6.3. This shows a border object decorating a scrollbar object, which in turn decorates the text view object.

When the `Draw` operation of the `aScrollDecorator` object is called, it first calls the `Draw` operation, then can, determine using its size and the current scroll position, decide what portion of the text to display. Note that using the `VisualComponent` interface in this way means that decorators can be stacked; since the `aScrollDecorator` object is also an instance of the `VisualComponent` class, it can be in turn decorated, for instance, to add a border.

## 6.4.2 Application of the Decorator Pattern

We used the standard GUI example of the decorator pattern in the previous section as this paradigm should be familiar to almost any user of a modern computer system. It is also useful to see how design patterns such as this can be applied within our approach. Recall in the `TorsionBar4-Baseline` model that a `PID` object is used to calculate the control value for the `Motor` actuator based on the current controller reading. The motor however takes a `PWM` value in the range  $(-1,1)$ , as defined by the `PWM` type (which is defined in the `IActuatorPWM` interface in the `TorsionBar5-Extended` co-model). For large deviations between the measured position and desired position, however, the `PID` object will typically yield a value way outside of the  $(-1,1)$  range. Therefore in `TorsionBar4-Baseline`, a limit function is used that restricts the output value to the valid range.

**Fig. 6.4** Class and object diagram for the LimitedDTControl application of the decorator pattern



The decorator pattern can be applied here with the DTControl object taking the role of the VisualComponent. This has been done in the TorsionBar5-Extended co-model. A decorator class, LimitedDTControl, has been created that encloses a DTControl object and imposes the limit on its output. When Output is called, the limiting class first calls Output on its enclosed object, then applies the limit to this value before it is returned. Calls to SetSampleTime are passed to the enclosed object to ensure it is calculating with the correct time. A class and object diagram showing this application is shown in Fig. 6.4.

The definition LimitedDTControl class is given below. Note that a constructor that allows the enclosed object to be passed in and the limit function that restricts the output are omitted for the sake of brevity:

```

class LimitedDTControl is subclass of DTControl

instance variables

limitedController: DTControl;

operations

-- limit the output of the enclosed object
public Output: real ==> real
Output(err) == limit(limitedController.Output(err), -1, 1);

-- delegate sampletime to enclosed object
public SetSampleTime: real ==> ()
SetSampleTime(s) == limitedController.SetSampleTime(s);

...

end LimitedDTControl
    
```

Hopefully, this example has shown the power of design patterns. A pattern that was initially described in terms of GUI components has been applied to an embedded controller model. This again makes clear that while the exact forms of two applications of the same design pattern may look different, the common approach described in the pattern can be seen in both. A number of design patterns described in Chap. 9 and Appendix C use variations of the decorator theme, for example, to create filters for sensors. The interface classes for sensors and actuators then play a role similar to the `VisualComponent` and `DTControl` classes in the examples above.

## 6.5 Using Inheritance for Threads

Concurrency is a key element of the DE models used in our approach. Concurrency is achieved by allowing multiple threads of control to exist and run in the model at the same time (see Sect. 4.8). The DE models in our approach will often have multiple threads where, for example, a supervisory controller may run at the same time as a safety monitor.

Threads are defined at a class level using a `thread` section; however, their definitions are often very similar across classes. It is therefore useful to create a generic class to handle definition of periodic threads, from which classes that need threads can inherit. The `TorsionBar5-Extended` model has one such class, called `AbstractThread`.

### 6.5.1 An Abstract Thread Class

The `AbstractThread` class has instance variables corresponding to the four parameters of a periodic thread (period, jitter, delay and offset), and these are set by the `SetupThread` operation:

```

-- setup loop parameters
protected SetupThread: nat1 * nat * nat * nat ==> ()
SetupThread(f, j, d, o) == (
  -- initialise the loop parameters
  sampletime := 1/f;
  period := floor (sampletime * RESOLUTION);
  jitter := j;
  delay := d;
  offset := o
);

```



Note that this operation takes the frequency (in Hertz) as a parameter because this is more intuitive than using a period in nanoseconds. The operation performs conversion to thread period, taking into account the nanosecond resolution of the DE simulator. The periodic thread is defined in terms of these instance variables in the following way:

```
thread periodic (period, jitter, delay, offset) (Step)
```

One aspect that is often required in the DE models of our approach is timekeeping. This includes keeping track of the current time and the time that has elapsed since the periodic operation last occurred. In the above definition, the periodic thread definition calls an operation called `Step`. This operation handles timekeeping, so that the following instance variables are available to subclasses: `now`, the current simulation time in seconds and `delta`, the time that has elapsed since the `Step` operation last ran.

Subclasses of the `AbstractThread` class need some way to define the actions that the class should perform during each period. If the `Step` operation is overridden, however, then the timekeeping calculations are lost (or must be recalculated by the subclass, which defeats the point of calculating them in the `Step` operation). Therefore, the abstract thread offers another operation, `StepBody`, that subclasses can override. The definition of the `Step` operation, showing the timekeeping and calls to other operations, is as follows:

```
private Step: () ==> ()
Step() == (
  -- update the time keeping (in zero time)
  duration (0) (
    last := now;
    now := time / RESOLUTION;
    delta := now - last;
  );

  -- call the loop pre-amble
  BeforeStep();
  -- call the loop body
  StepBody();
  -- call the loop post-amble
  AfterStep();
)
-- time book keeping must be consistent
pre last <= now
```

Note that in addition to `StepBody`, two other operations are provided, `BeforeStep` and `AfterStep`, that are called before and after `StepBody`, respectively. These three operations can be overridden independently

by the subclass. Splitting the step operations into three is a modelling choice; in applying this approach, the designer is free to choose whether it is useful to have three, fewer or more operations.

In the case of this example, as shown below, the `BeforeStep` is used to sample/hold (see Sect. 3.6.4) the shared variables and `AfterStep` is used for diagnostics and logging. The `StepBody` then implements the actual control strategy and can be replaced without affecting the sampling or the diagnostics.

The `AbstractThread` class provides a default implementation for these three operations that does nothing. For example, the `StepBody` operation is defined as follows (where `skip` is a statement that does nothing):

```

-- action to execute each control loop
protected StepBody: () ==> ()
StepBody () == skip;
```

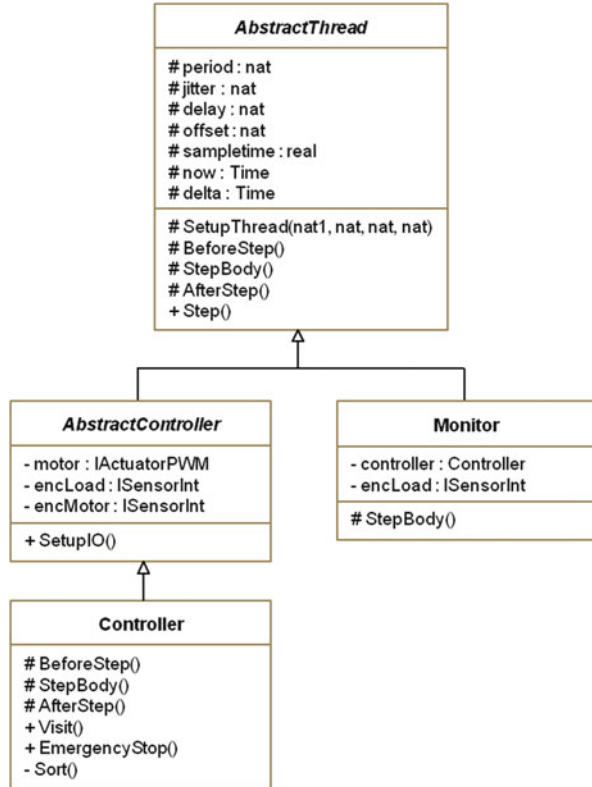
This is a slightly different design pattern of inheritance to that seen in the DT-Control example described above (see Sect. 6.2). In that example, the body of the Output operation is declared as **is subclass responsibility**, meaning that the subclass *must* override it and provide an implementation. In the case of `StepBody`, a default implementation is provided so that the subclass does not have to override it.

## 6.5.2 Using the Abstract Thread Class

The `TorsionBar5-Extended` example has two active classes (those with threads), a `Controller` class and a `Monitor` class. Both reuse the predefined thread behaviour of the `AbstractThread` class. In addition, an intermediate `AbstractController` class is introduced between the abstract thread and controller that is responsible for setting up the IO of the controller (in terms of sensor and actuator objects). A class diagram showing this hierarchy is given in Fig. 6.5.

The `AbstractController` class is declared as a subclass of the `AbstractThread` class. It defines three instance variables that represent the sensors and actuators in the system (two encoders, `encMotor` and `encLoad`, and one motor, `pwmMotor`). It provides an operation called `SetupIO` that sets these references. This is done by calling operations on an object called `TorsionBar`io`, which is known as a “factory” and is described in Sect. 6.6.3. The `SetupIO` operation is defined as follows:

**Fig. 6.5** Class diagram for the use of `AbstractThread`



```

-- setup sensors and actuators
protected SetupIO: IOFactory ==> ()
SetupIO(io) == (
  -- set instance variables
  encMotor := io.GetEncMotor();
  encLoad := io.GetEncLoad();
  pwmMotor := io.GetMotor()
)

```

In addition, the `AbstractController` class overrides the `BeforeStep` and `AfterStep` operations. In the former, it writes to the actuator and reads the sensors to be stored in local instance variables, while in the latter, it outputs some diagnostic information using the IO library provided by VDM. This means that all the concrete controllers will inherit these behaviours and only have to provide implementations for `StepBody`.

The definitions of the `BeforeStep` and `AfterStep` operations are given below. Of particular interest is the precondition on the `BeforeStep` operation. This ensures that the `SetupIO` operation has been called by the subclass and that objects have been assigned to the instance variables. Note that the call to

IO`printf is shortened for the sake of brevity, but it shows the use of the %s placeholder in the first parameter (a string) being replaced by values in the second parameter (a sequence of values).

```

-- action to execute before each loop body
protected BeforeStep: () ==> ()
BeforeStep() == (
  -- write actuator value
  pwmMotor.Write(hold_pwm);
  -- read sensor values
  sample_encm := enc2rot(encMotor.Read());
  sample_encl := enc2rot(encLoad.Read())
)
pre pwmMotor <> nil and
     encMotor <> nil and
     encLoad <> nil;

-- action to execute after each loop body
protected AfterStep: () ==> ()
AfterStep() == IO`printf("[%s] hold_pwm = %s; ...",
                        [now, hold_pwm, ...]);

```

The concrete controller `Controller` is defined as a subclass of the `AbstractController` class. It must provide an implementation for `StepBody`. In the case of the `TorsionBar5-Extended` example, this implementation follows that first shown in Sect. 4.6, deciding when to change the setpoint and calculating the output for the motor using a PID controller. The controller must call `SetupThread` and `SetupIO` from its superclasses, which is done in the constructor shown below. This also shows the creation of the PID control object, following the “limiting” design pattern described above in this chapter (see Sect. 6.4.2).

```

public Controller: nat1 ==> Controller
Controller(f) == (
  -- setup superclass
  SetupThread(f, 0, 0, 0);
  SetupIO();

  -- initialise instance variables
  pid := new LimitedDTControl(new PID(k, tauI, tauD));
  pid.SetSampleTime(sampletime);
);

```

Finally, the `Monitor` class inherits directly from `AbstractThread`. The operation `StepBody` is defined as in the `CheckMonitor` operation from Sect. 4.6 (not repeated here). In order to do this, it requires a reference to the controller that it

```

-- constructor for Monitor
public Monitor: nat1 * Controller ==> Monitor
Monitor(f, ctrl) == (
  -- setup superclass
  SetupThread(f, 0, 0, 0);

  -- set instance variables
  controller := ctrl;
  encLoad := TorsionBar`io.GetEncLoad()
);

```

is monitoring, as well as access to the encoder on the load. As in the example above, this is done in the constructor:

## 6.6 Structuring Constituent Models for Flexible Simulation

In this section, we look at ways to structure DE and CT models to allow them to participate in both co-simulations and single-domain simulations. One benefit of our approach is that the constituent DE and CT models of the co-models can still be analysed in their existing tools, as well as through co-simulation. In order to do this, some care has to be taken in how they are built and structured, which is what is explored in this section. The `TorsionBar5-Extended` co-model is built in such a way that it permits analysis through CT-only simulation, DE-only simulation and co-simulation.

This property is useful when following the domain-first approaches to building co-models: DE-first and CT-first (briefly introduced in Sect. 2.7.3 and explained in detail in Chap. 8). In particular, it means that single-domain regression tests can be performed after the co-model stage has been reached. For example, if a CT-first approach is followed by creating a CT-only model with a simple controller and large changes are made to the CT-model, old tests can be performed again in a CT-only simulation to confirm that the changes are sound.

We can view the switch between domain-only simulation and co-simulation as moving the “boundary” (or “interface”) between the constituent models. DE-only and CT-only simulations are the two extremes, where the other model plays no part in the simulation. Then for co-simulation, the boundary falls somewhere in between the constituent models, with the contract defining the bridge over this boundary. The choice of this boundary depends on the purpose of the model, and it may well change during the course of a development as designs evolve and become more detailed.

The structuring techniques described later in this section can also be useful if the co-model needs to support switching between multiple boundaries. Before describing the structuring of CT and DE models for flexible simulation (see

Sects. 6.6.2 and 6.6.3), we first look in more detail at the factors affecting the placement of the co-model boundary.

### 6.6.1 *Co-model Boundaries*

In order to place the co-model boundary in the “right” place for a given co-model’s purpose, it is necessary to have a solid idea of the components that form the system. In our approach, physical components are typically modelled in CT and software elements modelled in DE (though this is by no means mandated). Certain components however fall around the co-model boundary (and could thus be modelled in either CT or DE). Examples include loop controllers, sensors and actuators. Choices here can therefore affect the boundary and the content of the co-simulation contract.

In the early development stages, many design decisions will not yet have been made, so multiple alternative components might be considered. In fact, it is entirely possible that the same solution could be realised in software, hardware or a mixture of the two, in which case it is interesting to explore the cost and benefits of each solution. We revisit the possibility of trading off different solutions in the forward look to Cyber-Physical Systems (CPSs) in Chap. 14.

There are a few factors that should be considered when determining the co-model boundary and where each component is modelled:

**Simulation Performance:** Would a choice of boundary have an effect on the time taken to perform a simulation? An example here would be the location of a PID controller. If the PID were on the CT side, then the co-model interface carries the setpoint for the controller which may be updated at a lower frequency than the sensor and actuator signals, thus information traversing the co-simulation interface less frequently.

**Abstraction:** It could be the case that the implementation details of some component or series of components is not important to the purpose of the model. An example here would be the sensing of a shaft position by an encoder. At the detailed level, the sensor is modelled by scaling, sampling and quantising the value, while at the simpler level, the actual value for the position held by the simulator is sent over the boundary. A second example of this is a movable guide that diverts paper down one of two paths with a small probability that the paper arrives and collides with the mechanism while it is switching position. This can either be modelled in CT if the dynamic response is important or in DE if a simpler model where we only consider collision as a probability is sufficient.

**Maturity:** If part of a model is not well understood or represents an unstable part of the design, then a more abstract model may be an attractive option. As the design becomes more mature and design choices are firmed, then the extra fidelity potentially afforded by a more detailed model may be justified.

**Modelling Gaps:** It could be the case that either or both of the co-model parts are not yet complete, meaning that sensor implementations on either side are not yet available. In such a case, the co-model boundary could be wider than intended for the final model, where the width means that the data is read from or written to points that will not form the final interface.

As mentioned above, this boundary may change during the co-modelling process. For example, co-simulation performance and control loop tuning may well be important early on, hence modelling these on the CT side is preferable. Later on in the development, when it is desirable to predict the performance of the DE controller (including its ability to meet deadlines), then modelling the control loop in DE is preferred. Abstraction levels, maturity and the closing of modelling gaps may also affect the boundary choice as co-modelling progresses.

### 6.6.2 Structuring CT Models for Flexible Simulation

A CT model will generally contain elements for modelling the plant to be controlled and sensors and actuators. In addition, a CT-only model will have a controller block that is used to test the response of the plant and for creation and tuning of loop controllers.

Once the move to a co-model is made, this controller block is replaced by a controller block that connects to the DE model through the co-simulation contract. Blocks in 20-sim can have multiple alternative “implementations” that can be switched between. We recommend using this feature to retain the CT-only test controller implementation while introducing a co-simulation implementation. In this way, it is possible to switch between CT-only simulation and co-simulation. Additionally, if different co-model boundaries are explored, further implementations can be created reflecting the choice of boundary.

To create a new implementation for a block or to swap between existing implementations, you right-click on the block and select the `Edit implementation` menu. Figure 6.6 shows a screenshot from the `TorsionBar5-Extended` example, showing that the controller block has two implementations, one called “CTOnly” and the other called “CoSim” (which is currently selected as indicated by the check mark). This menu also has options to add, remove and rename implementations.

Note that each implementation can have its own separate icon. We recommend altering the visual style of the different implementations to make it easy to see which is selected at a glance. In the case of the example in the figure above, the colour will change; however, more complex icon changes are possible using the 20-sim icon editor (again accessible through the right-click menu through the `Edit icon` option).

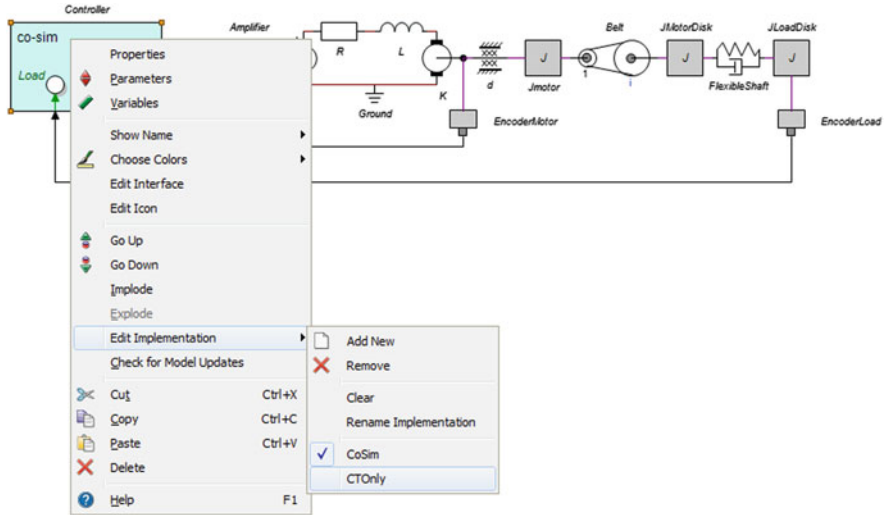


Fig. 6.6 How to change the implementation of the controller

### 6.6.3 Structuring DE Models for Flexible Simulation

A DE model generally contains various representations of software elements of the system. In addition, a DE-only model will usually have an approximation of the plant, sufficient to test the supervisory behaviours of the controller model. Here we focus on the structuring of such models, while specific guidance on building DE approximation is given in Sect. 8.5.

#### 6.6.3.1 Environment Models

A simplified environment model can be modelled as a class with its own thread, which acts as a basic simulator. It provides stimuli to, and observes the responses of, the controller model. In the *TorsionBar5-Extended* co-model, the *Environment* class approximates the position of the motor, and in turn the load disc, by making a number of simplifying assumptions, for example, that the PWM value directly controls the speed and that the shaft connecting the motor to the load disc is completely rigid.

This model is sufficient to test the supervisory control (the *Visit* operation and *Monitor* class), though it is of course not sufficient to tune the low-level PID controller. An extract of the *StepBody* operation is given below. Note that the *CSV* library is used to record the state of the environment in each time step. The file can then be visualised in an external tool (including 20-sim, using a *DataFromFile* block).



```

-- simulation loop
protected StepBody: () ==> ()
StepBody() == duration (0) (
  ...
  -- stop simulation, or perform step
  if time >= SIMULATION_TIME * RESOLUTION
  then World`done := true
  else (
    -- compute new position of motor based on pwm
    let speed = pwm * MOTOR_MAX_SPEED,
        distance = speed * sampletime,
        pos = position + distance,
        revs = pos / (2 * MATH`pi)
    in (
      -- update state
      position := pos;
      -- update variables read by sensors
      encm := floor (revs * ENC_RESOLUTION);
      encl := floor (encm / BELT_RATIO);
      -- diagnostics
      IO`printf ("%s] Environment step\n", [now]);
      -- CSV output
      if csv_open then
        let lspeed = speed / BELT_RATIO,
            lpos = revs / BELT_RATIO,
            - = CSV`fwriteval("tbar_extended_co-sim.csv",
                [now, lpos, lspeed, pwm], <append>)
        in skip; -- ignore return value of fwriteval
      )
    )
  );

```

As described in Sect. 6.3, the controller in the `TorsionBar5-Extended` co-model accesses the sensors and actuators in the system through interface classes called `ISensorInt` and `IActuatorPWM`, respectively. In order to interact with the simplified environment model in the `Environment` model, two concrete subclasses are used, `EncoderDE` and `MotorDE`. These classes implement the appropriate interfaces and have a reference to the environment object as in instance variable. For example, the `EncoderDE` class begins as follows:

```

class EncoderDE is subclass of ISensorInt

instance variables

-- environment model to access
env: Environment;

```

The DE encoder and motor classes use the `env` object to implement their operations. The `Environment` class must also provide access to the data.

The following extract shows the Read operations of the EncoderDE class, which returns the current encoder reading from the environment. Note that the class may be instantiated as “motor” or “load” encoder, indicated by the instance variable, type:

```

-- read the current value of this sensor
public Read: () ==> int
Read() ==
  cases type:
    <ENCM> -> return env.GetEncM(),
    <ENCL> -> return env.GetEncL(),
    others -> error
end

```

### 6.6.3.2 IO Factories

Within the DE model of the TorsionBar5-Extended example, there are two sets of sensor and actuator implementations: the Encoder and Motor classes for co-simulation and the EncoderDE and MotorDE classes for DE-only simulation. The controller model is built in terms of the interfaces, so it does not matter which of the two sets of objects is passed used by the controller, but at some point the decision has to be made.

The solution used in the TorsionBar5-Extended co-model is another design pattern from Gamma et al., called the *factory pattern*. This design pattern suggests defining “an interface for creating an object, [and letting] subclasses decide which class to instantiate” [39]. In this case, the IOFactory (input–output factory) interface provides operations to retrieve sensor and actuator objects. In order for the controller to access the factory, the *singleton pattern* [39] is used. This ensures that there is a single instance of a class that is globally accessible. This is done using a static instance variable defined in the system class (as seen in Sect. 6.5.2, the controller accesses TorsionBar`io):

```

-- sensors (two encoders)
public static io: [AbstractIOFactory] := nil;

```

The selection of the factory is made in the constructor of the system class. A value in the World class, DE\_ONLY, determines whether the DE factory (deio) or CT factory (ctio) is selected.<sup>4</sup> This value is set to 0 through a shared design parameter if a co-simulation is being run. This value is also used to instantiate the

<sup>4</sup>The deio and ctio objects are again singletons of the DE and CT factories, respectively.

environment object if a DE-only simulation is being run. The following is an extract of the system class constructor:

```
public TorsionBar: () ==> TorsionBar
TorsionBar() == (
  -- instantiate environment
  if World'DE_ONLY = 1 then env := new Environment(50);
  -- instantiate factory based on simulation type
  -- (must be done after env)
  if World'DE_ONLY = 1 then
    io := deio
  else
    io := ctio;
  ...
)
```

## 6.7 Conclusion

In this chapter, we discussed techniques for building and structuring constituent models, focussing on the use of object-oriented inheritance whereby a class can be defined as an extension of an existing class. We showed inheritance being used to promote reuse of definitions and cut down on repetition, as well as to define generic interfaces for some behaviours that can then be realised in a variety of ways depending on context. We also introduced design patterns as a means of describing solutions to groups of related problems, often distilling previous experience and best practice into a succinct description of both the problem and potential solutions. Finally, we discussed strategies for structuring DE and CT models for “flexible” simulation. Since models in our approach can still be analysed in their respective tools, it is often useful to retain this ability to run single-domain (DE- and CT-only) simulation even during co-modelling. This is particularly useful if the co-model began as a DE- or CT-only model. The same strategies are also useful where the boundary between the DE and CT model might change during a development. The notion of the co-model boundary, and what affects the choice of boundary, was also covered. The topics covered in this chapter are of particular relevance to the pragmatics of co-model creation (Chap. 8) and pattern-based approaches to fault modelling and fault tolerance (Chap. 9 and Appendix C).

**Part II**  
**Methods and Applications: The Pragmatics**  
**of Co-modelling and Co-simulation**

# Chapter 7

## Case Studies in Co-modelling and Co-simulation

Marcel Verhoef, Bert Bos, Kenneth Pierce, Carl Gamble,  
and Job van Amerongen

### 7.1 Introduction

Having described foundations for co-modelling and co-simulation using Crescendo, 20-sim and VDM, we now turn to the practice of collaborative development. In Part II of this book, we describe methods for constructing co-models (Chap. 8), techniques for fault modelling and fault tolerance for which collaborative approaches are particularly valuable (Chap. 9) and approaches to the exploration of design spaces (Chap. 10). Industrial case studies applying the technology are summarised in Chap. 11.

In this chapter, we present the background and challenges for two medium-scale case studies that will be used as running illustrations of the methods introduced in Chaps. 8–10. The first case study involves a small line-following robot (Sect. 7.2), which is a relatively simple example that provides a good basis for illustrating the principles of co-model construction, fault modelling and design space exploration. The second case study is more complex and is based on the experimental design of a personal transportation device, the “ChessWay” (Sect. 7.3), which gives insight into

---

M. Verhoef (✉)  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

B. Bos  
Chess iX, Haarlem, The Netherlands  
e-mail: [bert.bos@chess-ix.com](mailto:bert.bos@chess-ix.com)

K. Pierce • C. Gamble  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk); [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk)

J. van Amerongen  
University of Twente, Enschede, The Netherlands  
e-mail: [J.vanAmerongen@utwente.nl](mailto:J.vanAmerongen@utwente.nl)

the practice of co-modelling and co-simulation. We summarise the purpose of these case studies in Sect. 7.4.

## 7.2 The R2-G2P Line-Following Robot

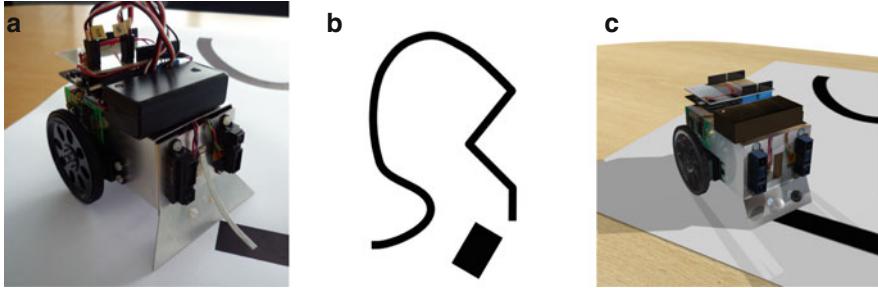
R2-G2P is a small two-wheeled indoor robot upon which several sensors are mounted (Fig. 7.1). The wheels are directly driven by a pair of continuous rotation servos, allowing both their direction and speed to be set, with the servos containing a feedback controller to maintain the set speed. There are four distinct types of sensors mounted on the robot. There are two infrared reflection sensors mounted near the front of the robot facing downwards. These sensors detect the reflectivity of the surface below, by emitting infrared light and measuring how much is reflected back via a light-dependent resistor. There are also two position encoders, one per wheel. Each encoder faces a 44-segment black and white disc attached to the wheel, allowing it to keep a count of how many segments have passed the encoder and in which direction, thus giving a representation of the angular distance travelled by each wheel. Finally, there are two infrared range finders on the front face of the robot with a micro switch between them to detect contact with objects.

The robot has been co-modelled performing two tasks, line following and line measuring. Both tasks present the engineer with interesting design decisions. For example, it is necessary to choose the number and positioning of infrared reflectivity sensors and how the wheel servos should respond to their signals. Such decisions become more challenging when considering the consequences of faults or difficult operating conditions. The construction of the R2-G2P co-model is discussed in Chap. 8. The modelling of faults and realistic operating conditions, and appropriate recovery strategies, are discussed in Chap. 9. Finally, Chap. 10 examines the systematic exploration of the robot's design space with a view to supporting the selection of optimal designs.

### 7.2.1 Line-Following

In the line-following mode, the controller primarily utilises readings from the infrared reflectivity sensors to guide the robot. Here it is assumed that the lines the robot has to follow, which are black, have a lower reflectivity than the surface upon which they are placed, which is white, in this way. The lines are all of a fixed width but are free to follow straight paths, smooth curves or any radius and also sharp corners.

The robot moves through a number of phases as it follows a line. At the start of each line is a specific pattern that will be known in advance. This known pattern allows the robot to calibrate its sensors as part of its fault tolerance strategy. Once a



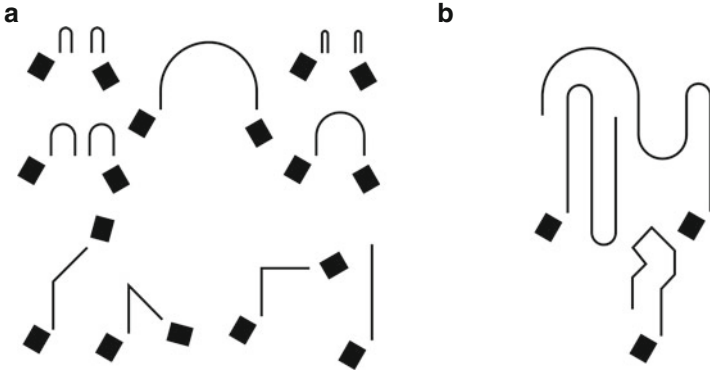
**Fig. 7.1** The line-following robot. (a) An R2-G2P robot. (b) A line-follow path. (c) 3D representation of the R2-G2P

genuine line is detected on the ground, the robot follows it until it detects that the end of the line has been reached, when it should go to an idle state. For modelling purposes, the co-model accepts a textual representation of a bitmap as the surface with a line to be followed.

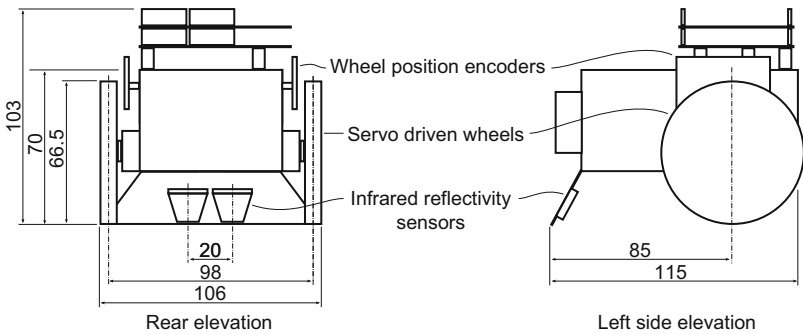
The ability of a design to follow a line is assessed using two criteria: the first is the time taken to follow the course of the line, from first detection of the line to passing the end of the line. The second criterion is the amount of energy used to drive the robot from start to finish. In this way, the best design can be a compromise between speed and efficiency. There is also a third implicit criterion, which is that the robot does not lose the line before reaching the end.

### 7.2.2 *Line-Measuring Extension*

The line-measuring mode of the robot is an extension of the basic line-following capability. In this mode, the controller also makes use of the encoders attached to each wheel to estimate the distance travelled by the robot while following the line, and thus it is able to estimate the length of the line. In this scenario, competing designs can be assessed using the same time and energy criteria as above but also using the accuracy of the line measurement as estimated by the controller. This additional need for accuracy of line measurement may favour a different number of sensors and control algorithms to the one favoured by the basic line-following task, in which only speed and energy are considered. Figure 7.2 shows examples of the lines to be measured.



**Fig. 7.2** Images showing the lines of known length for development (a) and for design comparison (b)



**Fig. 7.3** Line-following robot's major dimensions in mm

### 7.2.3 Assumptions and Robot Dimensions

There are a few key dimensions of the robot that should be respected during modelling. Figure 7.3 shows the significant geometric details of the robot. In addition to this, Table 7.1 contains important non-geometric property values.

The key assumptions regarding the lines to be followed are that they will always be 16 mm wide and the robot will start at a position where its left-hand side is over black, while the right-hand side is over white. There will then be a small gap of around 50 mm white space after the starting block before the line itself starts. The starting position of the robot will always be such that the robot will not be facing directly along the direction of the start of the line; instead the line will point either to the left or the right of the initial robot heading.



**Table 7.1** Robot property values

Property	Value	Unit
Mass	0.414	kg
Moment of inertia	$6.6e^{-4}$	$\text{kg m}^2$
Motor inductance	0.00171	H
Motor resistance	8.2	$\Omega$
Motor constant	0.796	–

### 7.3 The ChessWay Self-balancing Scooter

The ChessWay is a model-based design technology demonstrator based on the famous Segway Personal Transporter, see <http://www.segway.com>. The device exposes typical multi-disciplinary design challenges in real-time control, and for this reason it was put forward as a realistic industrial case study. Like the Segway, the ChessWay has two wheels, mounted on either side of a platform on which the rider can stand, while holding on to a handlebar (see Fig. 7.4). The weight of the system, including the passenger, is mostly positioned above the two wheels, and as such, it acts like an inverted pendulum which is inherently unstable.

Each wheel has an integrated direct drive electric motor which can generate sufficient torque to enable the system controller to keep the ChessWay upright, even while stationary. The controller measures the deviation angle and performs immediate action in order to keep the ChessWay stable, by moving towards the direction of the deviation, similar to the way that you might try to balance a pencil on the tip of your finger. It does this by controlling the current flowing through each wheel (motor) independently, hence controlling rotation and movement such that the base of the ChessWay is always kept directly underneath the centre of gravity of the entire system. The rider can move forward (or backward) by moving the handlebar slightly forward (or backward). The purpose of the controller is to provide a smooth and predictable driving experience. However, it is also intuitively clear that this is far from trivial as small disturbances may have a big impact on the performance of the device.

#### 7.3.1 Robustness: A Key Design Challenge

Safety plays a crucial and complicating role in the overall system design. For example, consider the situation where the power switch (indicated by p-SW in Fig. 7.4) is used to turn the system on. If the ChessWay is initially lying with the handlebar on the floor,  $90^\circ$  from upright, then the controller would immediately apply a large torque to the wheels in order to correct for this large measured position error. This could result in the handlebar suddenly swinging upright and possibly hitting the user. So, right from the start of the system design, we need to consider this initial scenario. One possible solution could be to start controlling the device

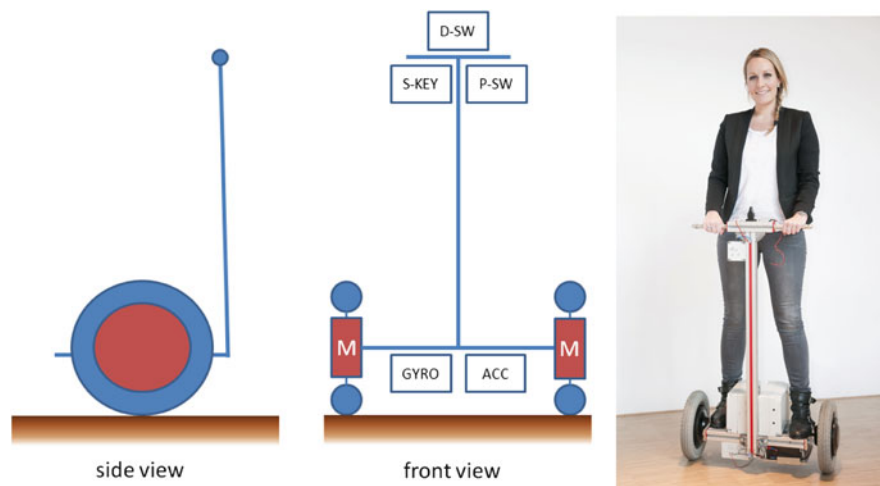


Fig. 7.4 The ChessWay personal transporter

actively if and only if the user has kept it manually upright for at least a few seconds, after the device is turned on. The user can then step onto the device safely once he or she feels force feedback on the handlebar, due to the torque applied to the wheels.

The user might also be operating the device outside its normal usage envelope, for example, creating sudden deviations by pushing the handlebar on purpose back and forth rapidly or by releasing the handlebar altogether. These situations can cause equally violent responses from the device.<sup>1</sup> One could consider filtering the measured angle to avoid malicious use and only allow active control within a certain range from the upright position in order to circumvent this. In all other situations, the approach could be to cut power to the wheels to avoid damage to both the device and the driver. Cutting power—leaving the wheels to rotate freely—is preferred to active braking because it lowers the cognitive load on the human. This may also create the opportunity to step down from the device normally in case of problems. If brakes are applied in an emergency situation while moving at speed, the user would simply fall due to inertia and the lack of time to respond to such a sudden change.

Note that there are also circumstances in which it is actually the *safest* thing to do to allow the ChessWay to fall over. For example, consider the case where the ChessWay hits some large object which causes the user to actually fall from the device. It could be dangerous to keep the motors engaged during this emergency situation. One possible solution is to monitor a safety switch (usually a so-called “rip cord” is used, indicated by **s-key** in Fig. 7.4) and cut the power to both wheels immediately as soon as the safety key is removed. We also need to consider how the

<sup>1</sup>Search [www.youtube.com](http://www.youtube.com) for “segway crashes” for examples of malicious users.

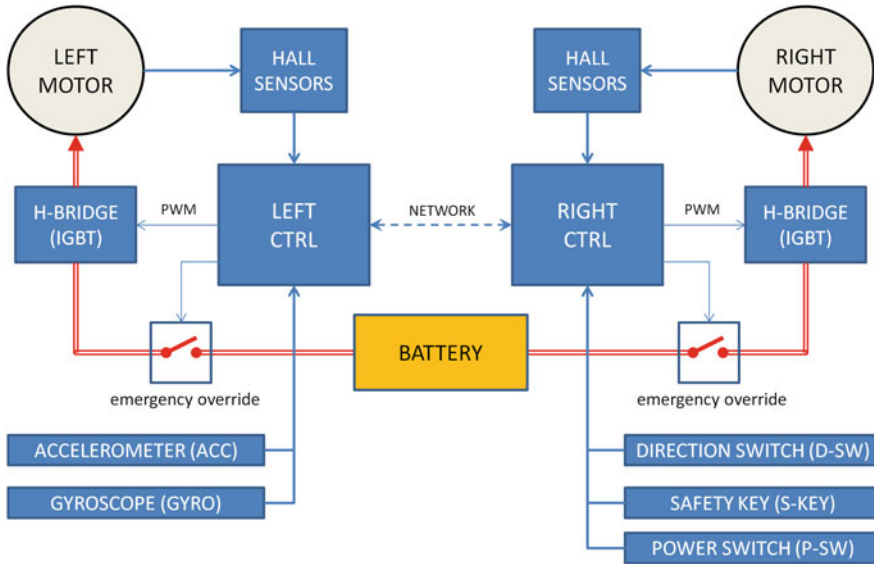


Fig. 7.5 Conceptual view of the ChessWay electronics architecture

system should respond when the rip cord is re-inserted again, to recover from such a situation.

Another key issue is the choice of the architecture of the control system itself. Modern embedded systems are becoming more and more distributed, for example, in order to balance cost price and performance. But what is the best way to interconnect all these components, sensors and actuators? These decisions have a major impact on overall system reliability, so it is worth exploring these architectures and make explicit trade-offs during the design process. An example conceptual control system architecture with the allocation of these different elements to the electronics is shown in Fig. 7.5. Note that in this architecture, safety is a shared concern between the two controllers. The controllers have to communicate to each other in order to assess the entire system state, as sensors are exclusively allocated to either of the processors. This implies that reliability of the communication link between the two controllers also becomes part of the system level reliability assessment.

The ChessWay is equipped with several sensors. Next to the power switch and safety key, there is also a direction switch, a gyroscope and an accelerometer (respectively indicated as *d-sw*, *gyro* and *acc* in Fig. 7.4). Furthermore, so-called Hall effect sensors are embedded within the wheel in order to measure rotation. Any of these sensors may degrade or fail entirely (e.g. missing, late, incorrect or jittery data) causing the system to behave unexpectedly. Of course, the same is true for the actuators, and also the control system itself may cause failures, for example, due to an internal communication or integrity fault. There also exist potential faults that appear gradually over the life-time of the device, such as battery degradation and

mechanical wear and tear, which may cause other (instant) system failures at some point in time.

The potential impact of the environment on the operation of the ChessWay should not be underestimated, as the device relies on the proper transfer of energy between the wheel and the surface in order to stay upright. For example, the surface might not be flat but curved, or there may be obstacles of different shapes and sizes on the track, some of these might even be too big to take on, friction might be different depending on the actual position the device due to a non-homogenous surface and ground contact may even be lost temporarily, for example, when the surface is discontinuous (a pothole or a small bump taken at speed). And these conditions may very well differ between both wheels, that is, only one wheel might be on a slippery surface. These scenarios create a very large design space that needs to be explored rigorously in order to assess overall system robustness. Note that this design space is in fact infinite, so all we can do is to be as thorough as possible, to build up confidence in the design without formal proof of correctness.

The main point to take away from this section is that design for resilience is not an afterthought, but an integral part of the design process that needs to be addressed right from the start.

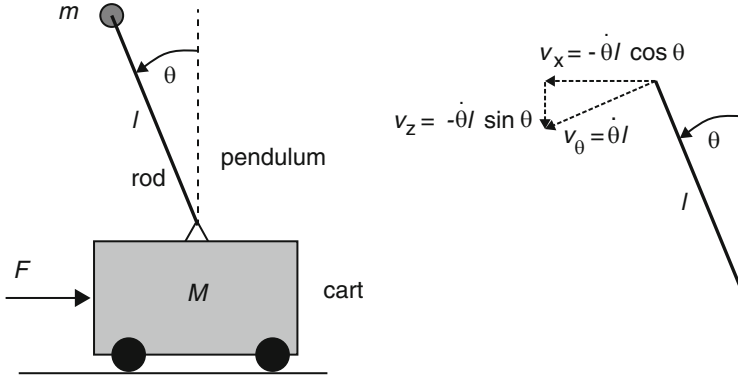
### 7.3.2 *The ChessWay Control Problem*

We consider the well-known inverted pendulum basic physics problem, as shown in Fig. 7.6, as a competent abstract mathematical representation of the ChessWay, if we ignore the ability to steer the device. The driver is modelled as a point mass which is connected to a weightless rod that is connected to a car through a frictionless joint. This so-called rigid body system is inherently unstable, which can be stabilised by moving the cart, by applying a force  $F$ . This force corresponds to the torque generated by the motor of the ChessWay, which is transferred through friction between the wheel and the surface. We select the following parameters for our model:

- $M = 20$  (kg), the mass of the cart
- $m = 70$  (kg), the mass of rider
- $l = 1$  (m), the distance of the top of the cart to the centre of mass
- $g = 9.8$  (m/s<sup>2</sup>), the gravity constant

A control model of this system should describe the translational motions of the cart as well as the rotations of the rod. The coupling between these two motions is highly non-linear, leading to a rather complex model. One of the approaches is to use the Lagrange function  $L$ , which is given by

$$L = K - V \tag{7.1}$$



**Fig. 7.6** *Left:* Inverted pendulum; *right:* translation of the angular velocity ( $\dot{\theta}$ ) to two linear velocities ( $v_x$  and  $v_z$ )

where  $K$  is the *kinetic* and  $V$  the *potential* energy. The potential energy is given by the gravity force  $mg$  times the height  $h$  of the mass  $m$  above the cart. With  $h = l \cos \theta$ , this yields

$$V = mgl \cos \theta \quad (7.2)$$

The kinetic energy consists of the kinetic energy of the mass  $M$  of the cart and mass  $m$  in the translation domain ( $\frac{1}{2}(M + m)\dot{x}^2$ ) plus the kinetic energy in the rotation domain  $\frac{1}{2}J\dot{\theta}^2 = \frac{1}{2}ml\dot{\theta}^2$ . The latter can be written as a function of the linear velocities of  $m$  in the horizontal ( $\dot{\theta}l \cos \theta$ ) and vertical ( $\dot{\theta}l \sin \theta$ ) directions, as shown in Fig. 7.6:

$$K = \frac{1}{2}(M)\dot{x}^2 + \frac{1}{2}m(\dot{x} - l\dot{\theta} \cos \theta)^2 + \frac{1}{2}m(-l\dot{\theta} \sin \theta)^2 \quad (7.3)$$

This can be written as

$$\begin{aligned} K &= \frac{1}{2}(M + m)\dot{x}^2 - ml\dot{x}\dot{\theta} \cos \theta + \frac{1}{2}ml^2\dot{\theta}^2(\cos^2 \theta + \sin^2 \theta) \\ &= \frac{1}{2}(M + m)\dot{x}^2 - ml\dot{x}\dot{\theta} \cos \theta + \frac{1}{2}ml^2\dot{\theta}^2 \end{aligned} \quad (7.4)$$

The equations of motion follow from the Lagrange equation:

$$\frac{d}{dt} \left[ \frac{\partial}{\partial \dot{q}} L \right] - \frac{\partial}{\partial q} L = \tau \quad (7.5)$$

where  $\tau$  represents the external forces and moments and  $q$  stands for  $x$  and  $\theta$ . Combining (7.5) with (7.2) and (7.4), this yields for the  $x$ -equation:

$$\frac{d}{dt} \left[ (M + m)\dot{x} - ml\dot{\theta} \cos \theta \right] - 0 = F \quad (7.6)$$

or

$$(M + m)\ddot{x} - ml\ddot{\theta} \cos \theta + ml\dot{\theta}^2 \sin \theta = F \quad (7.7)$$

and for the  $\theta$ -equation:

$$\frac{d}{dt} \left[ -ml\dot{x} \cos \theta + ml^2\dot{\theta} \right] - mgl \sin \theta = 0 \quad (7.8)$$

or

$$\begin{aligned} ml(-\ddot{x} \cos \theta + l\ddot{\theta} - g \sin \theta) &= 0 \\ -\ddot{x} \cos \theta + l\ddot{\theta} - g \sin \theta &= 0 \end{aligned} \quad (7.9)$$

The two equations of motion for  $x$  and  $\theta$  are thus

$$\begin{aligned} \ddot{x} &= \frac{1}{M + m} (F + ml\ddot{\theta} \cos \theta - ml\dot{\theta}^2 \sin \theta) \\ \ddot{\theta} &= \frac{1}{l} (\ddot{x} \cos \theta + g \sin \theta) \end{aligned} \quad (7.10)$$

This model can be linearised when we only consider small variations in the angle  $\theta$ . For small values of  $\theta$ , the following approximations hold:

$$\begin{aligned} \sin \theta &\approx \theta \\ \cos \theta &\approx 1 \\ \theta^2 &\approx 0 \end{aligned} \quad (7.11)$$

When we substitute (7.11) in (7.10), the model is simplified into

$$\begin{aligned} \ddot{x} &= \frac{1}{M + m} (F + ml\ddot{\theta}) \\ \ddot{\theta} &= \frac{1}{l} (\ddot{x} + g\theta) \end{aligned} \quad (7.12)$$

The resulting Eqs. (7.10) and (7.12) are deceptively simple, but they contain an *algebraic loop*, as  $\ddot{\theta}$  and  $\ddot{x}$  are mutually dependent. This is also reflected in the initial controller model shown in Fig. 7.7. An algebraic loop is a closed loop without an integrator or a delay element, or, in other words, the input of a block depends directly on its own output. This loop is indicated in the block diagram of the linearised system, see Fig. 7.7. This loop must be removed before we can simulate the system.

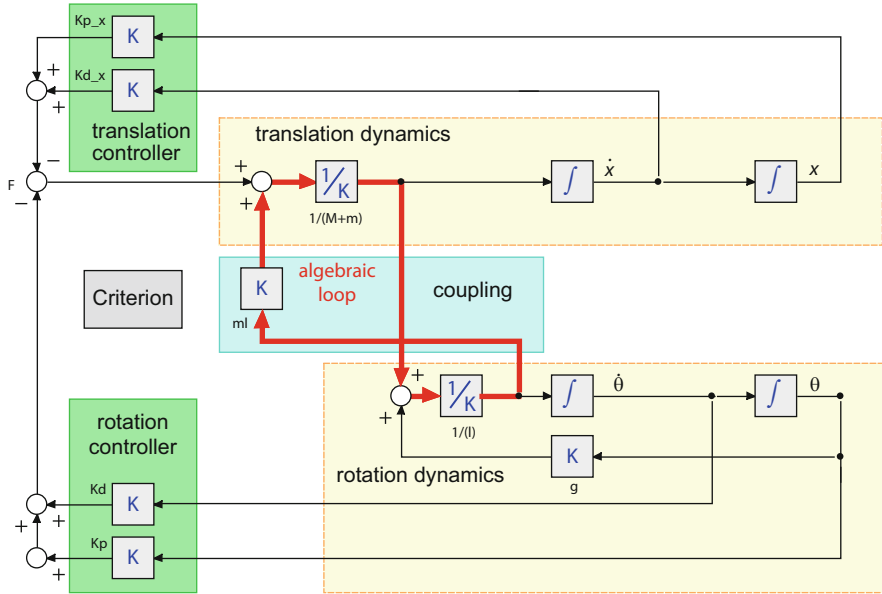


Fig. 7.7 Block diagram of the linearised system (7.12). The algebraic loop is indicated by the thick arrows

A similar problem occurs when sensors are introduced. One could imagine that a multi-axis accelerometer could be used to determine the angle of the ChessWay, as gravity is always pointing downwards. However, when the angle is non-zero (which tilts the frame of reference of the accelerometer) and the ChessWay is accelerating, the measurement of gravity will be biased by this forward or backward movement. Alternatively, a gyroscope can be used to measure the angular rate, which can be integrated to obtain the angle. In order to find the initial values of this integrator, the orientation of the device must be known a priori. For that, we would again need the accelerometer, so even ignoring issues such as sensor accuracy and drift, determining the orientation of the device is a non-trivial problem.

### 7.4 Conclusion

The case studies presented in this chapter demonstrate the intrinsic complexity of modern real-time control systems. This is of course a significant challenge and representative of many industrial products being developed today. The trade-off between the large variety in usage scenarios, functional requirements, environmental conditions and fault types makes it clear that analysis of any design can only sensibly be done semi-automatically in order to be effective. Deciding between the myriad

of options is envisioned as a typical design space exploration task in the Crescendo technology.

The need for this type of methodology is, for example, demonstrated by the public debate regarding the legality of allowing the Segway on public roads. For the device to become street legal, robustness had to be demonstrated to several third parties, such as government road safety inspectors, and to private insurance companies.

The line-tracking robot and ChessWay case studies will be used in the following chapters to show that our methodology allows the system developer to define alternative system architectures and controller strategies based on several possible user or usage scenarios, while reasoning about the suitability of the system under possibly changing environmental conditions and in the presence of potential faults.



# Chapter 8

## Methods for Creating Co-models of Embedded Systems

Kenneth Pierce, Sune Wolff, and Marcel Verhoef

### 8.1 Introduction

In Chap. 2, we introduced concepts to describe alternative routes to the production of initial co-models. In this chapter, we illustrate these approaches in some detail using the case studies outlined in Chap. 7: the *line-following robot* and the *ChessWay*. The *ChessWay* will illustrate DE-first co-model construction path and the line-following robot will demonstrate the CT-first path. Furthermore, we will show how higher-level descriptions such as SysML can be used to aid the process of initial model construction.

Section 8.2 reviews the three co-model production paths, considering the circumstances under which each is likely to prove most effective. In Sect. 8.3, we show how the SysML notation can be used both to clarify the purpose of a particular co-model and to decompose the problem at hand in different views, providing insights that help in selecting the co-model production path. The technique is illustrated using the line-following robot in Sect. 8.3. The chapter then looks in more detail at the common approaches to co-model production. Section 8.4 illustrates the CT-first (continuous-time first) approach using the line-following robot. Section 8.5 illustrates the DE-first (discrete-event first) approach using the *ChessWay* example. Section 8.6 briefly explains how the contract-first approach works. Finally, Sect. 8.7 provides a concluding summary.

---

K. Pierce (✉)  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

S. Wolff  
Aarhus University, Aarhus, Denmark  
e-mail: [swo@eng.au.dk](mailto:swo@eng.au.dk)

M. Verhoef  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

## 8.2 Paths to Co-models

In Chap. 2, we outlined three approaches to the production of a first co-model: DE-first, where initial models are produced in the discrete-event formalism; CT-first, where initial models are produced in the continuous-time formalism; and contract-first, where the contract forms the basis for development of both constituent models. In this section, we look in more detail at these approaches and the factors that influence the selection of a path in a given development context. We also consider some alternative approaches. When choosing a path to co-modelling, it is important to understand the purpose of the co-model (why it is being produced) and to have an idea of the various elements of the system being designed. In Sect. 8.3, we look at one way to help discover these elements using SysML.

### 8.2.1 *When to Use DE-first*

The DE-first approach begins with initial models being produced in the discrete-event formalism (VDM). This DE model will include 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, so it is a good choice if this aspect is critical, such as in systems with safety or security concerns. Initial system models in the DE formalism will have overly simplified plant dynamics, and therefore loop controllers cannot be tuned, for example. In addition, the complexity of the plant model increases rapidly as the DE-first approach continues, so there is a natural limit after which a co-model should be built. If legacy models exist in the DE formalism, or if the development team has mainly DE experience, then this approach seems the most natural. It should be noted however that legacy models might be seen to bias a development or result in “tunnel vision”, so careful consideration of all factors is recommended.

### 8.2.2 *When to Use CT-first*

The CT-first approach begins with models being produced in the continuous-time formalism (20-sim). The CT model should include a basic controller, such as a loop controller, in order to gain confidence in the model dynamics. This approach allows plant dynamics to be studied early on in the development and loop controllers to be tuned at an early stage. Another good reason to choose the CT-first approach is to perform a feasibility study to test whether the proposed design is in fact controllable. Following the other approaches in this case may waste effort before it is discovered that the design is physically infeasible.

Choosing this approach does mean that supervisory control cannot be studied until later in the development, so it should be used only if the plant dynamics are of a higher priority than supervisory control. If legacy CT models exist, or if the development team has most CT modelling experience, then this approach is a good way to reach an initial co-model. Again, caution is urged about legacy models biasing new developments. Experience also indicates that if the physical plant already exists and a new controller is to be built, the CT-first approach is again the natural choice.

### ***8.2.3 When to Use Contract-first***

The Contract-first approach begins with definition of a co-simulation contract, followed by concurrent development of the two constituent models. In this way, the constituent models must evolve together. Following the DE-first and CT-first approaches for developing the constituent models is recommended. We suggest that minimal acceptance tests for the constituent models are defined, and that these are met before new versions are “released” for inclusion in the co-model. This means that the constituent models are not dependent on each other for testing, especially in the early, volatile phases of development.

This is perhaps the most “pure” approach, and in theory a co-model can be reached early on, though following the concurrent DE-first and CT-first approaches on the constituent models adds extra effort (in producing DE environment model and CT test controller). This approach may also suit a technical manager who is overseeing a co-modelling team or teams: the contract can act as a means to manage the teams’ work. Alternatively, if there is no obvious driver for choosing either the DE-first or CT-first approaches, then a Contract-first is the best way to go. For example, if there are no legacy models (or you wish to avoid using legacy models), or if the development team has a good mix of DE and CT modellers (or perhaps no experience of the formalisms used), this is the natural approach to take. It also stresses the unknowns, early on.

### ***8.2.4 When to Define the Contract***

Note that selecting the Contract-first path requires the co-simulation contract, and therefore the co-model interface, to be chosen early on. While this could be changed later on as the co-model evolves, it is desirable to avoid this disruption and therefore identifying the possible interfaces in the system under design early on is particularly important when choosing this path. Conversely, the early definition of the contract can help guide the teams working on the constituent models, especially if the teams are geographically separated. With the DE-first and CT-first approaches, the initial models can be used as a way to work out what the interface should be, and “playing

**Table 8.1** Summary of approaches to co-model construction

	Pros	Cons	Use where...
DE-first	Complex controller behaviour can be studied early	Plant dynamics over-simplified; loop controllers cannot be tuned; rapid increase in environment model complexity	Complex DE control needs priority; legacy DE models exist; modeller experience is mainly in DE domain
CT-first	Feasibility study; plant dynamics can be studied early on; loop controllers can be tuned	Complex DE control cannot be easily studied	Feasibility of control unknown; plant dynamics need priority; legacy CT models and/or loop controllers exist; modellers' experience is mainly in CT domain
Contract-first	A co-model reached early on; constituent models not mutually dependent for testing	Contract required early on; extra effort is required in building testing constituent models	Integration is required of two legacy models; no legacy models exist; modellers from both domains are available (or have no bias)
Other	A novel approach can fit better with existing practice	Limited experience from our existing guidelines	The standard approaches do not fit your development context; legacy models or developer experience in other formalisms

around” is much easier within the single formalism. The co-simulation contract should be derived from the initial, single-domain model, so there should be no need to significantly alter this model in order to integrate it into a co-model.

### 8.2.5 *Alternate Exploratory Paths to Initial Co-models*

The standard paths summarised in Table 8.1 should be applicable in many contexts; however, the best path to production of initial co-models may be different in other contexts. For example, if legacy models exist, it may be instructive to focus on what is unknown and take the opposite path from that suggested in the table. So if a DE model exists, instead of working towards a co-model from there, a CT-first approach could be taken to focus on the new aspects of the model.

In addition, real developments may not be as clear-cut as those described in the table. For example, in the industrial applications described in Chap. 11, the

ChessWay personal transporter study began with a CT-only feasibility study. This suggested that it was at least possible to build the device. Next, a DE-only model was built to explore the modal aspects of the controller (see Appendix D). Finally, a DE-first model was built that was influenced by the earlier efforts, described later in this chapter.

In the document handling system, also described in Chap. 11, a DE-first model was built. This approach was selected because of the background of the lead engineer, as well as the complexity of having multiple connected sections of “paper path” within the system. Later on, in the co-model phase, only a small part of the plant model was replaced with a CT model. The interesting point here is the placement of the co-model boundary with respect to the “standard” split mentioned previously.

### 8.3 Using SysML Initially

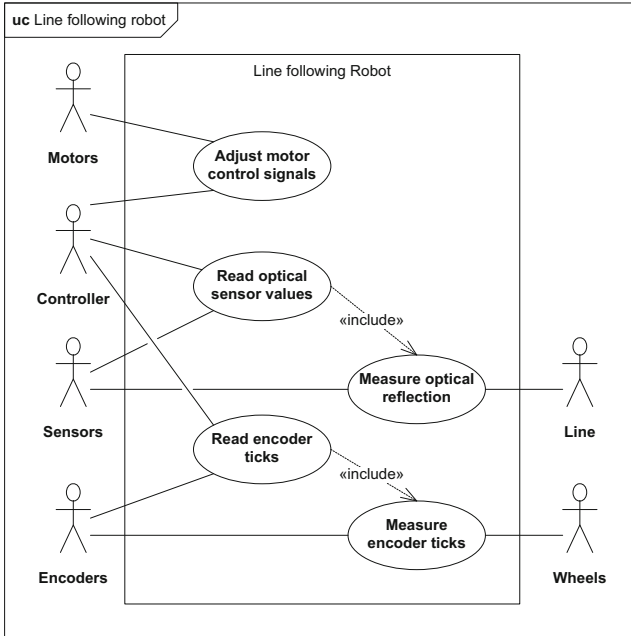
Before the actual modelling process is initiated, it is beneficial to have a clearly defined purpose for the model. This will help the model designer to apply an appropriate level of abstraction, while still ensuring the model is competent for the analysis at hand. In addition, having a fine-grained decomposition of the system is an advantage when having to decide which modelling approach to choose. Having specified the behaviour of the different blocks helps determine which parts of the system contain the main complexity and also determine the most appropriate modelling approach.

This section describes purpose modelling and system decomposition using the system modelling language (SysML) [87]. SysML is an extension to UML defined and maintained by the Object Management Group (OMG) in cooperation with the International Council on Systems Engineering (INCOSE). SysML is widely used in industry to manage and track requirements, link test cases to requirements, decompose systems into more manageable components and allocate requirements to the responsible system components.

The line-following robot is used as a sample application exemplifying the use of SysML for purpose modelling and system decomposition. The modelling of the line-following robot using the CT-first approach is described in Sect. 8.4.

#### 8.3.1 Purpose Modelling

In systems engineering, use cases are used to represent missions or stakeholder goals, and hence are perfect for defining the model *purpose*. Since use cases are described using natural language, it is a good common communication platform for engineers with different backgrounds and non-technical stakeholders like a customer or potential end users.

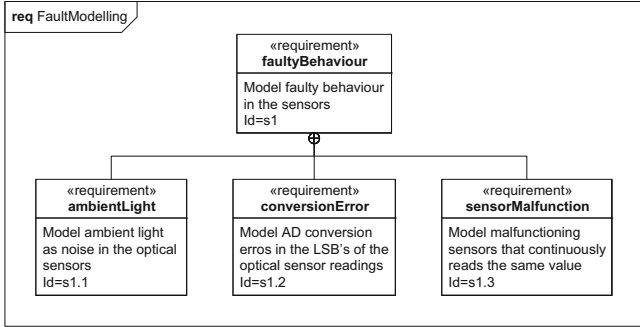


**Fig. 8.1** SysML use case diagram

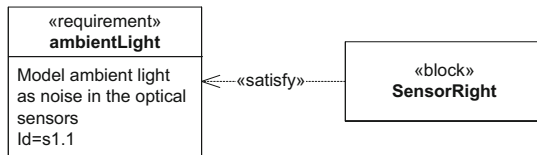
One of the key aspects of the model purpose is to identify all actors interacting with the system. An actor can either be a person, a role, an external system or a more abstract actor such as time. Unexpected actors can also be modelled: unauthorised users, power loss or other unexpected interactions with the system. A use case diagram for the line-following robot is shown in Fig. 8.1.

The main actors of the system are the hardware and software components of the robot: the wheels, encoders, motors, sensors and the controller. In addition, the line the robot needs to follow is included. The use cases define the main functionality of the controller: the optical sensors are read to determine where the line is; the encoders are read to determine how far the robot has travelled; and the motor control signal is set to ensure that the robot follows the line. The *«include»* relation is used to describe the dependencies behind the sensor and controller operational modes.

After identifying the use cases and actors of the system, more formal requirements that the model must satisfy can be defined. Some requirements can be *derived* directly from a use case, whereas other requirements will be *refinements* of use cases. In addition to these annotations, the *trace* association can be used to document the rationale behind a certain requirement. The use of these associations is a strong tool to ensure traceability of individual requirements and help document the rationale behind the requirements. Requirements diagrams can also be used to



**Fig. 8.2** SysML requirements diagram describing the required faulty scenarios that must be included in the co-model



**Fig. 8.3** The right optical sensor is given the responsibility of satisfying one of the requirements of faulty behaviour modelling

define different faulty scenarios that must be modelled. An example of this is shown in Fig. 8.2.

The main requirement states that faulty behaviour of the sensors must be modelled. Three sub-requirements define different types of faulty behaviour that must be included in the model. The responsibility of satisfying individual requirements can be defined in separate diagrams. An example of this is shown in Fig. 8.3.

### 8.3.2 System Decomposition

Once the model purpose and requirements have been determined, the system must be decomposed into its main parts. A *Block Definition Diagram* (BDD) is used for defining the main blocks of the system and how they are connected. Obvious candidates for main blocks are all the actors as well as the main nouns used in the use case descriptions. Some blocks are contained within parent-blocks which can be shown using the *part association*. A BDD for the line-following robot is shown in Fig. 8.4.

The BDD defines the composition of the main components of the robot. Separate blocks have been defined for the left and right wheels, motors and encoders. In addition, the physical body of the robot has been defined. The line the robot needs to follow is contained in a separate *Environment* block.

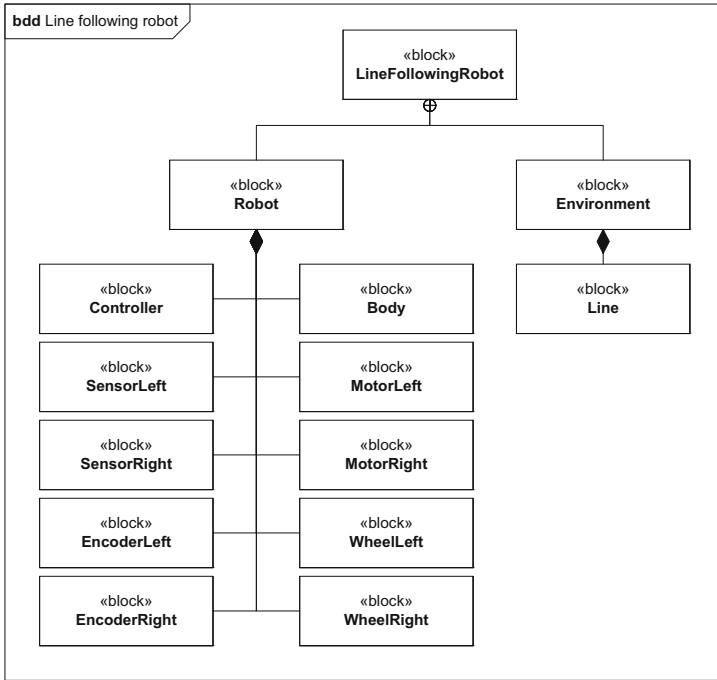


Fig. 8.4 SysML block definition diagram

Once the main blocks of the system have been defined, it is time to determine which parts of the system model should be modelled in the DE and CT formalisms. This is mainly a task for the domain experts who possess the detailed knowledge required for distinguishing this. Blocks describing rigid body entities naturally belong in the CT domain, whereas software controllers belong in the DE domain. There are exceptions to this though: if the controller simply needs to control an actuator in order to reach a preset output value using a Proportional-Integral-Derivative (PID) regulator, this could be done in a CT formalism. 20-sim is capable of tuning PID controllers and will in general obtain more precise results with less simulation speed overhead. Since the controller of the line-following robot is a software component, it is modelled using a DE formalism, whereas the rest of the system is modelled using a CT formalism.

To add levels of detail to the SysML model, an *Internal Block Diagram (IBD)* can be made for each of the main (parent) blocks of the system. In these diagrams, the child-blocks, their interfaces and interconnections are described using SysML ports. For defining a directed flow between two blocks, the *atomic flow ports* are used, which map directly to a signal port in the interface of the 20-sim submodel. The bi-directional *flow ports* of SysML are used to describe exchange of energy (flow ports in 20-sim). The IBD of the robot and environment is shown in Fig. 8.5.



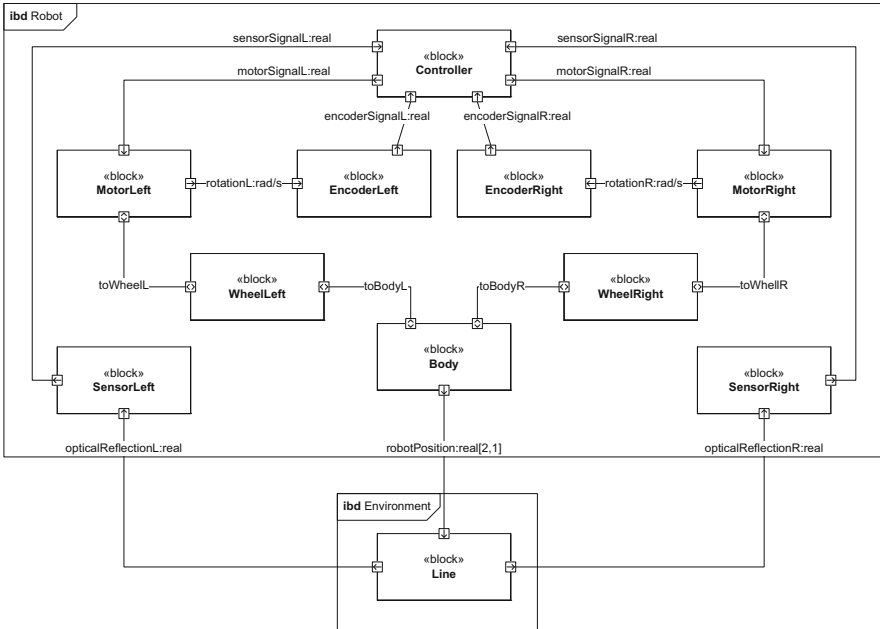


Fig. 8.5 SysML internal block diagram

All of the blocks defined in the BDD of Fig. 8.4 have been added to the IBD of Fig. 8.5 and their interfaces have been described. Signals to and from the controller are described using SysML atomic flow ports, while the rotational energy transferred between the motors, wheels and encoders are described using flow ports.

**8.3.2.1 CT Constructs**

To specify constraints on the parameters of the system, an additional BDD can be made, containing *constraint blocks* that define general physical constrains which can be applied to many different entities. An example of such a constraint is Newton’s second law of motion which is not tied to any specific property, but can be reused throughout the model. The use of constraint blocks to describe the forces acting on the robot is shown in Fig. 8.6.

A *parametric diagram* that is a child of a block shows how one or more of the general constraints are tied to properties of the owning block. An example of this is how Newton’s second law defined as a constraint block is tied to the mass value property of a rigid body block as well as the resulting forces acting on this block. If a non-causal CT modelling formalism like bond graphs is used, it is enough to use the constraint blocks since the causality description of a parametric diagram is not needed. This defines the differential equations of the system, and

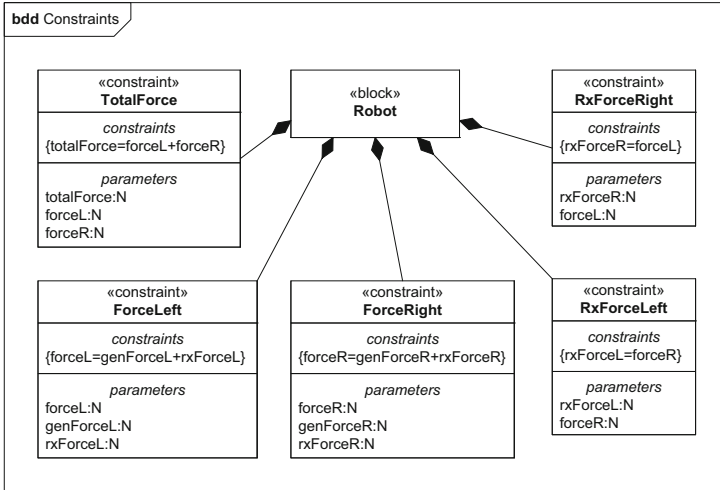


Fig. 8.6 SysML block definition diagram describing the physical constraints of the robot

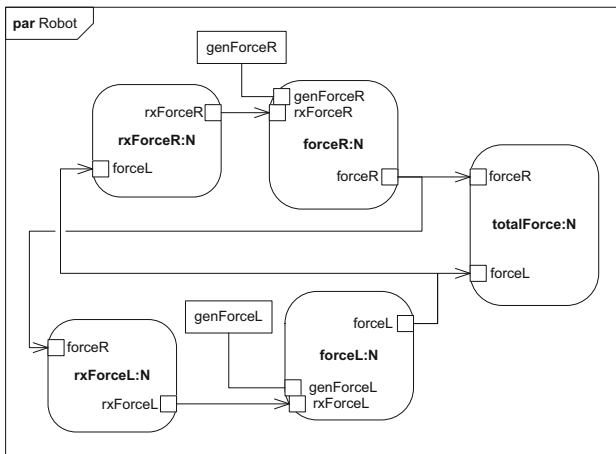
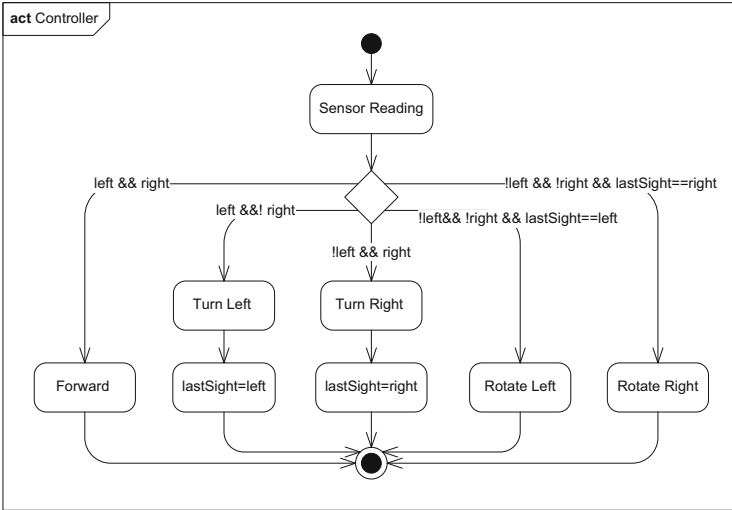


Fig. 8.7 SysML parametric diagram describing the combination of the physical constraints of the robot

20-sim calculates the causality at run-time. An example of a parametric diagram showing the causality of the forces acting on the robot is shown in Fig. 8.7.

### 8.3.2.2 DE Constructs

In addition to defining the main blocks of the controller in the BDD and IBD as described above, it is beneficial to make a more detailed specification of the software



**Fig. 8.8** SysML activity diagram

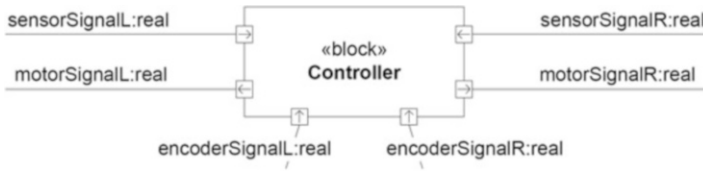
structure and behaviour. Using the UML *class diagram* to specify the structure of an object-oriented software structure is the most common approach. Since SysML is built on top of UML, a class diagram can be integrated into the SysML model.

The behaviour of the controller and other significant parallel processes can be specified using one of the behavioural diagrams of UML: *sequence diagrams*, *state machines* or *activity diagrams*. It is even possible to make a combination of these behavioural diagrams: using a state machine for defining the high-level state changes of the controller and separate sequence diagrams for each of the individual states. An activity diagram describing the main flow of the controller is shown in Fig. 8.8.

If both sensors can see the line, the robot continues forwards. If only one sensor can see the line, the robot slowly turns in that direction. The controller keeps track which sensors have last seen the line using the `lastSight` attribute. If none of the sensors can see the line, the robot rotates in the direction of the value of the `lastSight` attribute.

**8.3.2.3 Co-simulation Contract**

Defining the monitored and controlled shared variables of the co-simulation contract is supported by the details that have been added to all CT and DE blocks. The interface between two blocks modelled using different formalisms has already been specified: the name, type and direction of the individual ports have been defined in the interface and can be added directly to the contract. A close-up of the controller interface of the line-following robot can be seen in Fig. 8.9.



**Fig. 8.9** Close-up of the interface of the controller block. This interface defines the monitored and controlled variables of the co-model

Shared design parameters (describing constant valued properties) can be derived from constraint blocks in the parametric diagrams.

Events can be derived from sequence diagrams, which can specify both operation calls as well as events happening. These events must be added to the contract to enable event-driven communication.

## 8.4 The CT-first Approach

In the CT-first approach, initial models are produced in the continuous-time formalism. The focus is on developing a CT model of the plant dynamics first, with Discrete-Time (DT) loop controllers realising the core laws that control the plant. Once sufficient confidence is gained in the CT-only model, the steps towards an initial co-model are taken. There are two main choices here: either the loop controller can be moved to the DE-side, which improves analysis potential but reduces performance, or the DE model can be initially connected as a sequence controller, providing setpoints to the CT model.

### 8.4.1 Preparation

Our approach recommends that CT models have a certain shape, which may be somewhat more constrained than that experienced CT modellers are used to.

In general, a block is used to represent the plant and a block is used to represent the controller. The controller has input ports for monitored variables and output ports for shared and controlled variables. Sensors and actuators are also modelled as blocks, and these are connected in between the plant and controller. Naturally these blocks may contain further blocks (submodels) as appropriate.

A simplified overview of the CT-first approach is given in Fig. 8.10. Development begins with a model of the physical plant in the CT formalism. This model contains a plant block (P) and a controller block (C), which are linked by (one or more) sensor and actuator blocks (S and A). Initially, the controller block should be used

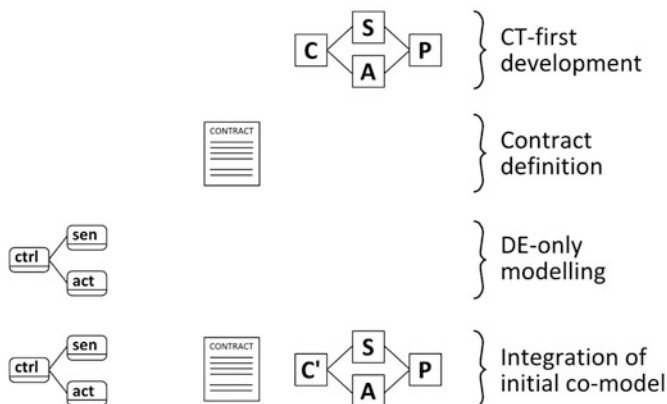


Fig. 8.10 The CT-first approach

to test the dynamics of the plant and to test the control laws. The sensor and actuator blocks model A/D (analogue-to-digital) and D/A (digital-to-analogue) conversion, respectively. The sensor blocks principally make values discrete in time (sampling) and value to ensure that the control designed can control the real system.

Once sufficient confidence is gained in the CT-only model, the steps towards an initial co-model are taken. The main change is that the controller block is replaced by one that connects to the co-simulation and acts as a place for the shared variables to reside. We recommended using a different implementation of the controller block so that the original controller can be swapped back in if necessary. The change of name from C to C' in Fig. 8.10 indicates this new implementation.

As described in Chap. 3, blocks in 20-sim can be implemented in a number of ways: graphically, with code, with bond graphs and with Idealised Physical Model (IPM) blocks. These types may also be mixed. Experts in 20-sim advocate using bond graphs or IPMs for the plant model in order to achieve the highest fidelity simulation; however, these may present a barrier to new users, as the notation is not as intuitive as the regular iconic diagrams they represent. Physical models can be built directly by coding differential equations and using predefined blocks from the 20-sim libraries, and this may present an easier option for those less familiar with bond graphs. Using signal blocks has lower fidelity and limited compositionality. If the model is divided into blocks, it should be possible to replace a simple plant model with a high-fidelity bond graph model with relative ease. Sensors and actuators are by their nature signal level, and a choice is not required for these components. The line-following robot example presented in the remainder of this section uses bond graphs organised into submodels, which represent its various components (body, wheels and servos). The sensors (wheel encoder and infrared sensors) are implemented using signal-level equations.

## 8.4.2 Plant Modelling

Since a model is a simplified representation of the system under study, abstraction is inevitable to capture only the relevant and interesting aspects of the system. The basic modelling goal is to derive a competent model that is as simple as possible, but sufficiently succinct to give the desired information. To judge whether a particular model is competent, it can be validated using simulations. There are two situations when modelling a plant. Either the plant exists and the model must reflect the real apparatus, or the plant does not exist and is being designed.

For an *existing* plant, we can model its structure using a phenomenological approach: look at the apparatus to discover what physical system elements the plant consists of. Values of the model parameters can be found using identification techniques, by exciting the real apparatus and measuring responses at relevant points.<sup>1</sup> 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.

Modelling a non-existing plant means that specifications for it need to be described as physical system models. Multiple models may be produced as design alternatives and compared to each other via simulation. In doing so the alternative models cover (a part of) the design space. As modelling and simulation is in general faster and cheaper than making prototypes, Design Space Exploration (DSE) can be facilitated through modelling and simulation. The DSE process can continue into controller design for the most promising design candidates. Then alternatives can span all parts of the model, such that one can trade off between controller solutions and plant solutions. Further guidance on DSE can be found in Chap. 10.

We recommend following the suggestions in the list below when building a CT model.

1. Use meaningful names in the models and submodels.
2. Fill in quantities and units of the physical system variables, to let 20-sim use this metadata for checking the model.
3. Annotate graphs with coloured backgrounds as a means to bring in structure.
4. Let excitation signals start with equilibrium values (usually 0), to check that the initial state of the model is in equilibrium.
5. Check that the 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.
6. Check the ranges of the values of the variables.
7. Check results of “standard” experiments, like step response and transient response.
8. Test non-linear model parts separately, as submodels in a test rig.

---

<sup>1</sup>This process is referred to as *system identification* in control engineering.

### 8.4.3 CT-first Modelling of the Line-Following Robot

The line-following robot co-model, first introduced in Chap. 7, was created using the CT-first approach. As mentioned in the previous section, for an *existing* plant, we should study it to see what elements it is composed of (a phenomenological approach), then build and combine models of these elements. A stepwise approach is often helpful, adding elements one at a time. As each stage is added, the model should be tested to ensure that the model is behaving as expected. Initially these tests might be made using simple signals or inputs, followed by a more complete CT-only controller.

The robot example was studied in Sect. 8.3 above, using SysML. That description gives the following elements that should be modelled:

- the robot's body;
- left and right wheels;
- left and right motors (which are in the form of continuous-rotation servos);
- left and right wheel encoders; and
- left and right infrared sensors and the line they should detect.

#### 8.4.3.1 Robot Body

The body is the main element of the robot to which the other elements are attached, and therefore the body was modelled first. The body submodel is shown in Fig. 8.11, along with the contents of the submodel. The body is modelled using a bond graph, with the MTF node at the centre representing the mass of the body. The `wheel_left` and `wheel_right` inputs may cause a rotation or translation of the body, which are computed in the `theta` and `position` blocks, respectively.

At this early stage, it is already possible to connect the position and rotation to a 3D visualisation and check the behaviour of the body. To do this, constant sources of effort (`Se`) are used to check that the body moves forwards, backwards and turns as expected.

#### 8.4.3.2 Wheels and Servos

The next item in the elements list are the wheels, so submodels were added in between the `Se` nodes (see Fig. 8.12). The sources of effort now take the roles of motors in testing. A wheel works by rotating about an axis and producing a translation perpendicular to this axis. This is modelled using the bond graph, also shown in Fig. 8.12, with the TF elements computing the translation from the given rotations.

Once testing of the wheel submodels is complete, the continuous-rotation servo submodels are added. These replace the sources of effort used for testing and rotate the wheels themselves. The input to the servos will be a signal from the

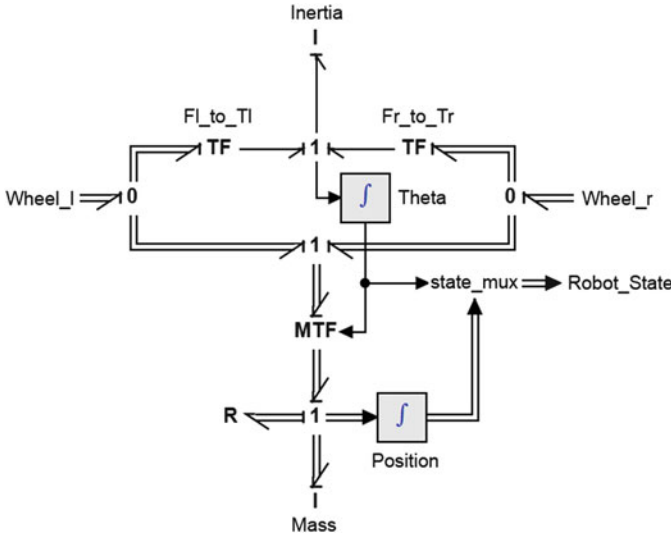


Fig. 8.11 Bond graph model of the robot's body

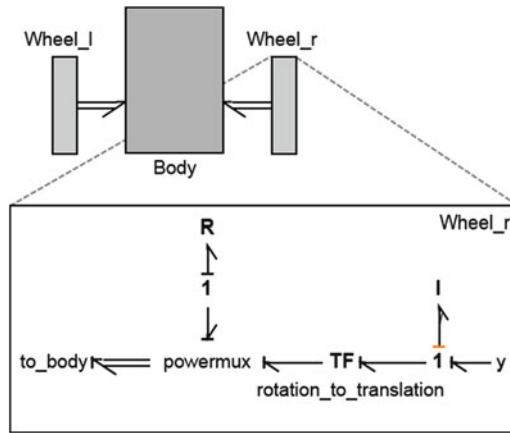


Fig. 8.12 Wheel submodel and bond graph

controller, and therefore it is in this step that the controller block was introduced. The servo receives a signal in the range  $(-1,1)$  representing full reverse and full ahead, respectively. This is shown in Fig. 8.13. A simple equation implementation of the block is used to test the robot with servos attached.

A servo takes a signal (typically using pulse-width modulation) which tells it what angle to move to and hold. In the case of continuous-rotation servos however, this signal tells the servo how fast to rotate. So this signal actually represents a setpoint, with the servo essentially representing a simple closed-loop controller. This



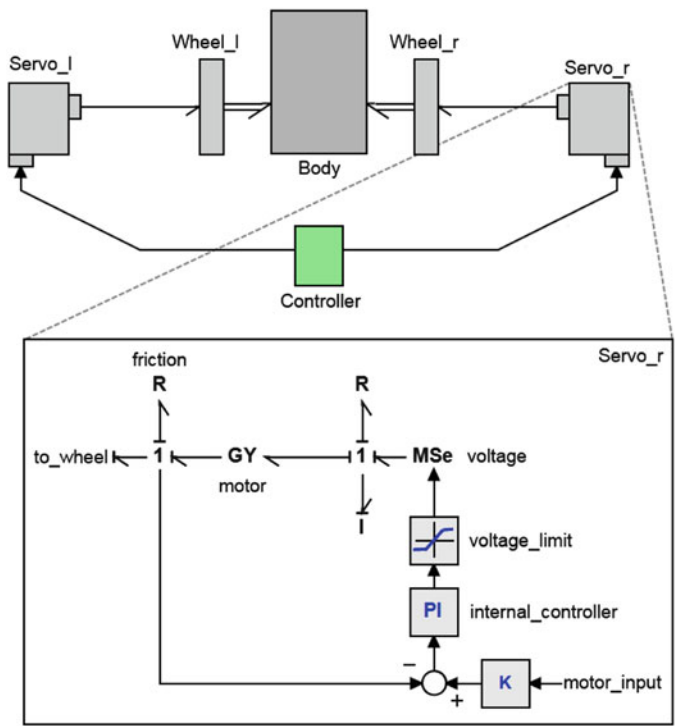


Fig. 8.13 Servo submodel and bond graph

can be seen in the bond graph model of the servo, also shown in Fig. 8.13, with a PI block controlling a motor element (GY) with a feedback loop.

### 8.4.3.3 Sensors

The encoders were the first sensors to be added to the model as they are simpler to model than the infrared sensors. The encoders on the robot count as the wheel turns (44 for one revolution). In the model, however, it is simpler to intercept the rotation from the servo (which is an input to the wheel), therefore encoder submodels were added in between the wheel and servos submodels, as seen in Fig. 8.14. Input ports were also added to the controller block. The encoder submodel integrates the rotation, giving an angle. This is then converted to a count and quantised, also seen in Fig. 8.14.

The final additions added to the CT robot model were the infrared sensors used for line following. Submodels were added which take as input the position and rotation of the robot body, as seen in Fig. 8.14. From this input, the submodels calculate their exact absolute position in the world. A black-and-white bitmap file (representing a piece of white paper with a black line printed on) was made and

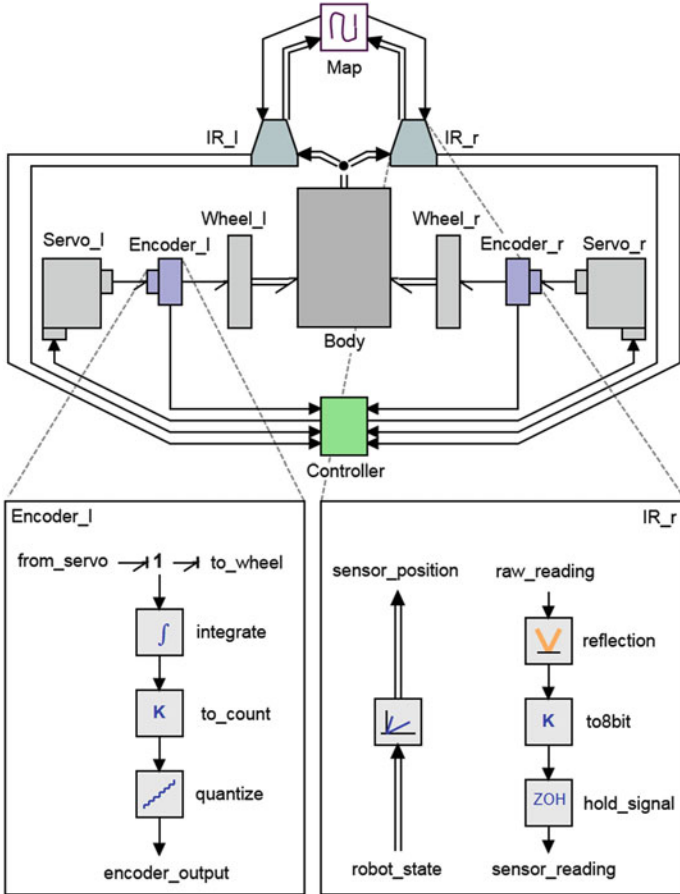


Fig. 8.14 Encoder and infrared sensor submodels

converted into a text file, with values for each pixel indicating whether that pixel is black or white. A `map` submodel was added, which accesses the map text file, taking sensor positions as input and yielding a value for black and white to the sensor submodels. The bitmap was also added to the 3D visualisation.

The final version of the CT-only controller block performed a basic line-following behaviour. This was sufficient to test the infrared sensor and map submodels. Once sufficient confidence was gained, the transition to an initial co-model was made, following the approach described in the next section.

### 8.4.4 Transition to Co-model

As mentioned in the introduction to this section, there are two main choices when making the transition to an initial co-model from a CT-only model. Eventually, all

supervisory, sequence and loop control should be realised in the DE model. Since the DE model should stand as a reference for software implementation and be used as a predictor of timing behaviour, it should contain all elements of the software.

There are reasons however why it might not be best to transfer loop controllers to the initial DE model. The main factor is simulation speed. Loop control typically requires much more frequent co-simulation synchronisations than sequence control as it occurs at a higher frequency. Another issue is complexity. While library classes are provided to help realise P, PI, PD and PID loop control, generation of a controller that only has to calculate the setpoint for these loop controllers is an easier task.

If the sequence controller method is followed, the loop controllers can be moved later. If alternative block implementations are used on the CT side, and subclasses used on the DE side, then it is possible to switch back and forth using the speed of the sequence control boundary until it is necessary to test the timing properties of the DE model. Note also that some systems may not require loop controllers in the software, for example, if actuators are used that contain feedback control already, such as servos.

In either case, a contract must be defined. Using the loop control boundary, this should match the loop controllers defined previously. For the sequence control boundary, this contract should allow the DE controller to pass setpoints to the loop controller(s). Monitored variables may or may not change, depending on the sensors used in the system (those used by the sequence controller may or may not be the same as the loop controllers). Once this is done, the CT model must be made ready to connect to co-simulation.

This block should define `global import` and `global export` variables that match the names and “directions” (import for controlled, export for monitored) of those in the contract. If a new implementation is used, it is possible to switch back to the original implementation to test the model CT-only again as shown in Sect. 6.6.

## 8.5 The DE-first Approach

In the DE-first approach, initial models are produced in the discrete-event formalism and the focus is on developing the DE controller first. The DE model contains a controller and a discrete approximation model of the controller’s environment (e.g. plant), which are linked by one or more sensor and actuator objects. The environment object is used to mimic an approximation of 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. These concepts are discussed in Chap. 2.

The environment essentially acts as a basic simulator running in its own thread and provides stimuli to, and observes the responses of, the controller model. At this stage, scenarios can be tested using DE mechanisms, such as reading values

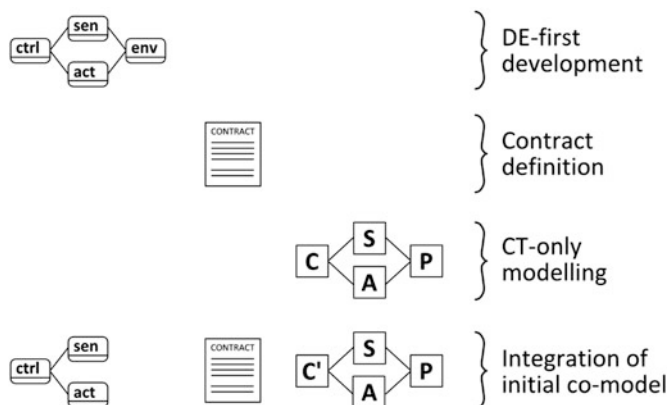


Fig. 8.15 DE-first approach

from a file. Once the co-model stage is reached, these scenarios can be realised with Crescendo mechanisms such as scripts (see Sect. 5.5).

### 8.5.1 Preparation

The shape of a DE-first model is closely linked to that of the general pattern for VDM controllers. In general, the controller uses sensor and actuator objects to access an environment model. This requires sensor and actuator classes that contain references to an environment class, which allows them to read and mutate the environment object, respectively. The controller and environment then run in separate threads in order to interact with each other.

An overview of the DE-first approach is given in Fig. 8.15. Development begins with a system model in the DE formalism. This model contains a controller object (*ctrl*) and environment object (*env*), which are linked by (one or more) sensor and actuator objects (*sen*s and *act*). The environment object is used to mimic the behaviour of the CT world in the DE domain.

Once sufficient confidence is gained in the DE-only model, the steps towards an initial co-model are taken. This requires definition of a contract, which should match that used by the sensor and actuator objects. In addition, alternative versions (implementations) may be provided for the sensor and actuator objects that do not interact with the environment object and act simply as locations for shared variables that are updated by the co-simulation engine.

The contract then informs the creation of the CT model, which should represent a higher-fidelity version of the initial environment model. These three elements can be integrated into the co-model and tested. During co-simulation, the environment object is not instantiated (hence it is omitted from the bottom of Fig. 8.15).

The order in which the elements of the DE-model are presented here is not necessarily the only order in which they can be built. You may wish to begin with the controller and tackle the environment last or begin with the sensor and actuator classes and build outwards. Either way, one good approach is to define minimal `Controller` and `Environment` classes, and abstract sensor and actuator classes, then build in empty operations first. These classes can then be elaborated in the most appropriate order.

## 8.5.2 Environment

We recommend building an `Environment` class that can act as (or be called by) a thread. An operation in the `Environment` class should be called by this thread periodically, with the time since the last call, `dt`, as a parameter. A common name for such an operation is `Step`. This `Step` operation should compute the new state of the environment model over the given time, responding to inputs as appropriate.

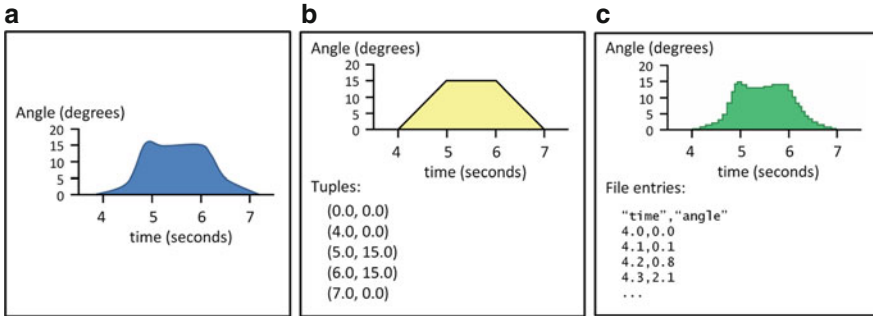
Within the `Environment` class, there are two main ways to build an environment model. The first is *data driven*, where some pre-calculated data is read into the model and provided to the controller model via the sensors. The second is to implement some integration method, such as Euler, acting as a basic CT simulator.

A combination of the two is perhaps more likely. For example, consider a simple ticker-tape reading machine. The tape is white with black marks on representing some information (e.g. Morse code). A small motor pulls the tape past an infrared sensor so that the controller can read and decode it. To model this, the `Environment` class needs to know the current position on the tape and be able to read data on the colour of the tape at that point. The position of the tape depends on the speed of the motor and the previous position of the tape, so a basic integrator will be required.

### 8.5.2.1 Data-Driven

The Crescendo tool includes two library classes to help with data input. The first is the `IO` class (introduced in Chap. 4). This is a general purpose class that includes operations to write data to the console. The useful function here is `freadval`, which reads a value from a file. This can be accessed statically, that is, using `IO.freadval`. Note that you must provide the expected type to be read in square brackets when calling the function, for example, `freadval[seq of char](filename)`.

An alternative is to use the `CSV` class for accessing *Comma-Separated Value* (CSV) files. The `CSV` class also provides a `freadval` function that takes an additional integer parameter which is the line of the file to read. Again you must provide the expected type in square brackets, with `seq of real` being an obvious



**Fig. 8.16** Visualisation of the change in angle of a dial as it is rotated  $15^\circ$  clockwise and back (a) and two DE approximations (b), (c). (a) Real-world curve. (b) Linear interpolation. (c) Pre-computed values

one for numerical sensor data. For boundary checking, the number of lines in the file can be found using the `flinecount` function.

To save on file accesses, the contents of a file can be read into a data structure during initialisation (e.g. within the constructor of the `Environment` class).

### 8.5.2.2 Basic Integration

The key here is to find approximations that allow controller logic to be tested in a reasonable time frame. The environment model will be composed of elements that model devices in the physical environment of the controller. It must also handle input from other external sources such as the user or the operating conditions. Approximation strategies for defining components of the plant include replacing non-linear relationships with linear approximations and replacing complex differential equations with a mixture of simpler differential equations and constants.

For example, in a water tank with an outflow, the rate of flow of water depends on the volume of water, therefore as the tank empties, the rate will decrease. A DE-first approximation might use a constant flow rate, where discrete quantities of water disappear from the tank at each time step. This is useful if we are primarily concerned with the correct logic of the controller, rather than with the exact time at which the tank empties.

Although linear approximation is also useful when defining external input, we may wish to define some erratic behaviours. For example, consider a user rotating a dial clockwise and back again, with the variation in rate and the overshoot that may be expected from a manual operation (Fig. 8.16a). One approach to approximating the curve is to identify the key points at which the curve changes and interpolate between these points linearly, representing the curve as a sequence of pairs each giving a time and corresponding angle (Fig. 8.16b). This gives a smooth, though approximate, change in angle. If a less linear approximation is required, the data-

driven method can be employed, with values that can be pre-computed and read from a file (Fig. 8.16c).

A basic integration method, such as Euler, can be sufficient for an environment testing core control logic. Consider a moving object with an acceleration, velocity and position, simulated over some time step,  $dt$ .

```
position = position + velocity * dt;  
velocity = velocity + acceleration * dt;
```

In a CT model, the integration method will be much more accurate and, in addition, the acceleration will be based on the mass of the object and in turn on the forces acting on that mass. In a DE approximation, the acceleration can be a constant, representing some force acting on the object. If necessary, more complex equations can be added, though this will take more time to model, so there is a trade-off here. In the above case, the forces acting on the object could be modelled and therefore a more accurate acceleration modelled; however, more effort is required to model and calculate these forces.

### 8.5.3 *Sensors and Actuators*

Sensor and actuator objects are used to allow the controller to interact with the environment. To do this, classes are required. In addition, different classes may be required for DE-first and co-model situations. In the former, for example, the objects must interact with the environment object, perhaps performing calculations to generate sensor values. In the latter, the objects may just act as placeholders, updated by the co-simulation engine. Once the implementation level is reached, additional code may be required to interface with hardware. To cope with this, abstract classes or interfaces are recommended. The controller can then be built referencing only these abstract classes, which means when the transition to a co-model, and later code, is made, the controller does not have to be changed and can seamlessly be placed into a co-model setting.

For each type of sensor and actuator in the system, an abstract class is created (see Chap. 6). It is suggested these have a name of the form `AbstractSensor` or `AbstractActuator`. Each class should have instance variables representing the value(s) of the sensor and actuator. A single **real** or **bool** variable is common, so where multiple sensors will have the same interface (type), only a single abstract class is required, for example, `AbstractSensorReal`.

Each abstract class should declare `Read` and `Write` operations as appropriate, whose body should be declared as **is subclass responsibility**. This declaration delegates the workings of the read and write operations to the concrete classes created later.

For the sensor and actuator classes to interact with the environment model, they must be able to access and modify its state. Therefore, operations should be added to the `Environment` class so that the sensor classes can read the data they need and the actuator classes can write the data they need. Typically, these can be of the standard object-oriented `Get` and `Set` style.

For each type of sensor and actuator in the system, concrete classes should be created that subclass the appropriate abstract class created previously (based on the type of the value). It is suggested that these have a name of the form `Sensor_DE` and `Actuator_DE`. As opposed to abstract classes, a concrete class is suggested for each type of sensor and actuator. These concrete classes should have an instance variable of type `Environment` and allow this to be set in the constructor.

Each concrete class must declare `Read` and `Write` operations with the same signature as the abstract superclass. These operations should then be implemented with a body that reads or updates the environment instance variable, as appropriate.

In the case that the system will be required to handle events, such as a button being pressed, then a slightly different structure should be employed. Here the controller must contain an asynchronous operation that is to 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 environment model should also contain an event generator, this object reads the environment 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. When the model changes from DE only to a co-model, 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.

## ***8.5.4 DE-first Modelling of the ChessWay Self-balancing Scooter***

### **8.5.4.1 The Environment Class**

The `Environment` class contains discrete approximations of the plant and external elements that may ultimately be replaced by CT models. External inputs include the state of the safety key, direction switch, on/off switch and user input (leaning forward/backward). The evolution over time of these environment variables is described by linear approximations. The environment class defines a set of reserved names, one for each external input, and an instance variable that maps each of these input names to a sequence of readings from points in the approximated curve. For example, the following defines such an approximation mapping for four inputs of interest.

The `tCtCurve` elements give a time, a value of the relevant input at that time and the gradient at that point. A possible behaviour for a scenario to be described as



```

values
  reserved: set of seq1 of char =
    {"RIGHT_SAFETY", "RIGHT_DIRECTION", "RIGHT_ONOFF", "USER"}
types
  public tCtCurve = real * real * real;
  public tCtBehavior = map seq of char to seq1 of tCtCurve
  ...
instance variables
  private mCtBehavior: tCtBehavior := {|->}

```

a constant within the environment model. For example, consider a simple scenario in which the ChessWay is enabled after 2 s and disabled again at 8 s, and the handlebar is moved forward and backward in the period between 4 and 7 s. The following mapping defines the scenario in terms of the evolution of the four external inputs mentioned above. This is only one possible approximation structure, and modellers are free to choose more or less elaborate and detailed approximations as they see fit, and depending on the purpose of the simulation.

```

{ "RIGHT_SAFETY" |-> [ mk_(0.0, 1.0, 0.0) ],
  "RIGHT_DIRECTION" |-> [ mk_(0.0, 0.0, 0.0) ],
  "RIGHT_ONOFF" |-> [ mk_(0.0, 0.0, 0.0),
                    mk_(2, 1.0, 0.0),
                    mk_(8, 0.0, 0.0) ],
  "USER" |-> [ mk_(0.0, 0.0, 0.0),
             mk_(4.0, 0.0, 0.2618),
             mk_(5.0, 0.2618, 0.0),
             mk_(6.0, 0.2618, -0.2618),
             mk_(7.0, 0.0, 0.0) ]
}

```

The operation `mainLoop` implements the core functionality in the `Environment` class, which executes as a periodic thread started by the `RunScenario` operation. On each iteration, it determines the system time and updates the environment model in a manner that reflects the causal relationship between the external inputs and elements of the plant (e.g. the user's state affects the accelerometer and gyroscope). First, it evaluates the external inputs to the sensors by reading the scenario, then it updates the wheel, then the Hall effect sensors and finally the user's state (including their deviation from upright).

The `mainLoop` operation also checks the current simulation "wall clock" against a preset maximum simulation target time. Once this is reached, the `terminate` operation is called, which will stop simulation and return control to the user.

```

operations
  private mainLoop: () ==> ()
  mainLoop() ==
  ( dcl ticks: nat := time,
    clock: real := ticks / World`SIM_RESOLUTION;
    evalSensors(clock);
    mLeftWheel.evaluate(); mRightWheel.evaluate();
    mLeftHall.evaluate(); mRightHall.evaluate();
    mUser.evaluate();
    if (ticks >= mMaxSimTime) then terminate()
  );
  ...
thread
  periodic (1E6, 0, 0, 0) (mainLoop) -- 1kHz frequency

```

### 8.5.4.2 The World Class

A special `World` class is the top-level entry point of the ChessWay DE model. In this class, instances of the controller and environment are created, and then the `RunScenario` operation instantiates the entire simulation model. It loads a user-defined scenario which specifies the initial settings of all ChessWay devices (such as the safety, direction and on/off switches) that are controlled by external forces (such as the rider) and the evolution of those settings over time, during a simulation run.

The environment model is linked to the `ChessWay` system class to facilitate simulation (the link has no counterpart in the final implementation of the system). Simulation commences through the `PowerUp` operations starting the periodic controller threads in the left and right controller objects. The environment simulator thread is started in order to execute the scenario. In this model, the coupling between the environment and controller objects is created by the `setEnvironment` operations of the left and right controllers.

```

operations
  public RunScenario: seq1 of char ==> ()
  RunScenario(fname) ==
  ( dcl env: Environment := new Environment(self, MAX_SIM_TIME);
    env.loadScenario(fname);
    ChessWay`lctrl.setEnvironment(env);
    ChessWay`rctrl.setEnvironment(env);
    ChessWay`lctrl.setRightController(ChessWay`rctrl);
    ChessWay`rctrl.setLeftController(ChessWay`lctrl);
    ChessWay`lctrl.PowerUp(); ChessWay`rctrl.PowerUp();
    start (env);
    waitForSimulationEnd()
  );

```

### 8.5.4.3 Sensors and Actuators

```

class MotorSensorDE is subclass of MotorSensor

instance variables

private mwheel: Wheel

operations

public GetValue: () ==> HallData
GetValue() ==
    let position = mwheel.GetPosition() in
        cases (position div 60):
            0 -> return mk_HallData(true, false, true),
            1 -> return mk_HallData(true, false, false),
            2 -> return mk_HallData(true, true, false),
            3 -> return mk_HallData(false, true, false),
            4 -> return mk_HallData(false, true, true),
            5 -> return mk_HallData(false, false, true),
            others -> error
        end;
end MotorSensor

```

The class `MotorSensorDE` shows how the state of the Hall sensors is determined, based on the current angular position of the wheel axis. Note that the so-called Grey codes are used to encode the motor position, whereby only one of the Hall sensors changes state when the wheel is rotated  $60^\circ$ . This enables the controller to assess the validity of the sensor data as well as determine the direction of rotation.

### 8.5.4.4 The ChessWay System

The ChessWay controller's distributed architecture is composed of two CPUs connected by a bus which enables communication between the controllers deployed on each processor, for example, to exchange information relating to their internal state. The VDM-RT model defines these in a system class that uses built-in CPU and BUS abstractions available in VDM-RT. The two FPGAs are defined as instance variables using the CPU constructor with parameters defining the scheduling policy, either Fixed Priority (FP) or First-Come-First-Served (FCFS), and processor capacity in terms of instructions per second. The BUS constructor's parameters are the type of bus (FCFS is in fact the only kind available in the language at present), its bandwidth in bytes per second and the identifiers of the set of CPU instances that the bus connects. The constructor in the system class deploys the instances of the `LeftController` and the `RightController`, one to each CPU, which are named `LeftCtrl` and `RightCtrl`, respectively.

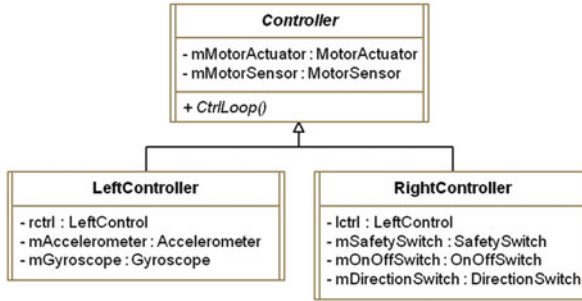


Fig. 8.17 ChessWay controller class diagram

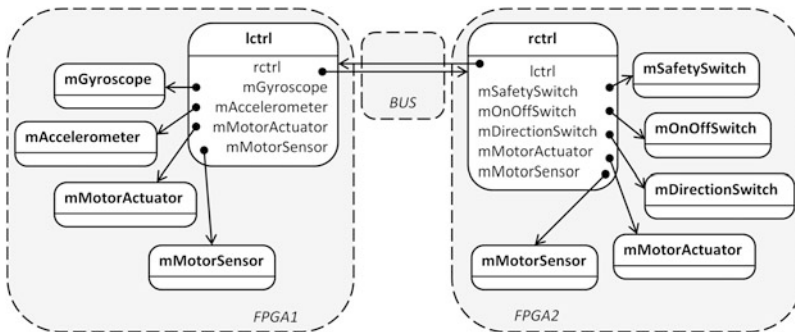


Fig. 8.18 ChessWay controller deployment model (object diagram)

```

system ChessWay

instance variables
  fpga1: CPU := new CPU(<FP>, 10E6);
  fpga2: CPU := new CPU(<FP>, 10E6);

  bus: BUS := new BUS(<FCFS>, 100E3, {fpga1, fpga2});

  static public lctrl: LeftController := new LeftController();
  static public rctrl: RightController := new RightController()

operations
  public ChessWay: () ==> ChessWay
  ChessWay() ==
  ( fpga1.deploy(lctrl, "LeftCtrl");
    fpga2.deploy(rctrl, "RightCtrl")
  );

end ChessWay
    
```

The ChessWay controller class hierarchy is illustrated in Fig.8.17, and the deployment of instances of these classes is illustrated in Fig.8.18. Alternative system architectures deploying the functionality to different processors can be explored by changing this part of the model.

### 8.5.4.5 The Controller Class

There are two controllers in the system model, so their common features are defined in a superclass from which the left and right controllers inherit. Each controller is linked to a `MotorActuator` and a `MotorSensor`.

```
class Controller
instance variables
  public mName : seq of char;
  protected mMotorActuator : MotorActuator;
  protected mMotorSensor : MotorSensor;
```

The right-hand controller in the `ChessWay` controls the right wheel and monitors the safety, direction and on/off switches. The `RightController` is created by subclassing the generic `Controller` class and overriding the operation prototypes for `CtrlLoop` and `PowerUp`. The `LeftController` class is similar to the `RightController` class with the exception that it manages an accelerometer and a gyroscope instead of the switches controlled by the `RightController`. For reasons of brevity, we omit details here.

### 8.5.5 Transition to Co-model

The sensor and actuator classes built for the DE-first model hold reference to the `Environment` class, and the instance variable comes from or influences the DE environment model. When running a co-simulation, the environment class will not be instantiated as this role will be filled by the CT model. Therefore, alternative sensor and actuator classes are required that simply act as placeholders for shared variables. 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. Class names along the lines of `Sensor_DE` and `Actuator_DE` classes are suggested.

In order to perform a co-simulation, the “DE” versions of the sensor and actuator classes must be instantiated. In order to perform a DE-only simulation, the original versions and `Environment` class must be instantiated. Following the principles of building models for flexible simulation described in Sect. 6.6.2, a conditional statement can be used to select which set of classes to instantiate. This allows the simplified environment model to be retained in the co-model, giving the option to perform both co-simulations and DE-only simulations into later stages of co-model design.

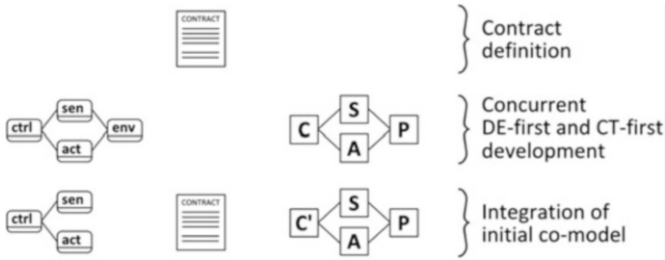


Fig. 8.19 The Contract-first approach

## 8.6 The Contract-first Approach

The Contract-first approach emphasises collaboration within a team, where one or more engineers work on the DE model while other engineers produce the CT model in parallel. The engineers collaborate to define the contract first, then constituent models are developed separately but concurrently, following the respective DE-first and CT-first approaches. The contract acts as a guide and target for constituent model development. An overview of this is shown in Fig. 8.19.

Unlike the DE-first and CT-first approaches, the Contract-first approach *requires* a contract to be defined upfront. While this does not have to be the final contract of the co-model, changes to the contract will affect both constituent models. We recommend defining a procedure for changing the contract, such that both constituent model teams can be prepared to propose, and to deal with, changes to the contract. An initial contract that at least captures the core ideal functionality of the co-model is recommended.

In order to produce a co-model from the constituent models, it is necessary to combine them with the contract. This is called *integration*. We recommend nominating an *integration tester* to perform this task. This could be handled by a third party, or someone from either the DE or CT team. Experience with co-modelling to date suggests that a DE modeller can perform this role relatively easily, since replacing a DE environment model with a CT model is a straightforward task.

It is important to note that the integration tester's job is not to test the constituent model's functionality, but that of the overall co-model. Effort is wasted if the constituent models are buggy and the integration tester is involved in debugging. Therefore, it is suggested that a minimal acceptance test is defined for each constituent model. The constituent models should not be passed to the integration tester unless they meet these criteria. As the development progresses, new acceptance tests can be defined for new functionality.

## 8.7 Conclusion

The choice of process for the development of a co-model is governed both by technical features of the problem and its decomposition and by pragmatic concerns such as the relative development risks of the DE and CT elements. In this chapter, we presented CT-first, DE-first and Contract-first routes to the production of co-models and the use of SysML to provide early clarity on the purpose of modelling and suggest structural decompositions. These approaches were illustrated using the line-following robot and ChessWay case studies (full models are available online on the book's web site). Practical experience to date suggests that co-modelling can—with care—be integrated successfully into a range of development processes and business contexts.

# Chapter 9

## Co-modelling of Faults and Fault Tolerance Mechanisms

Carl Gamble, Kenneth Pierce, John Fitzgerald, and Bert Bos

### 9.1 Introduction

Since it is not feasible to predict statically the behaviour of systems in the presence of faults, we advocate *fault simulation*, the simulation of the system in the presence of faults. This is achieved by modelling the faulty behaviour, *fault modelling*, then activating this behaviour, *fault activation* (or *fault injection*; note, however, that in some computer science fields, injection implies automated perturbation of a model or data set). Due to this approach, it is sometimes necessary to distinguish between the part of the model that represents the system we wish to build, the *System Under Test (SUT)* and aspects of the model that would not be realised in the real system. For example, we may model a fault to test the behaviour of the SUT, but we would not then go on to implement these in reality.

To cope with these behaviours, a system can implement some sort of *fault tolerance* or *resilience* mechanisms to continue providing a service in the presence of faults. This can be achieved through *error detection and recovery*. A system may also offer *degraded behaviours* if it cannot achieve a full service in the presence of faults.

Where possible, we adopt the terminology of Avizienis et al. [6] with respect to dependability concepts. 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 specification. Note that the term "fault" is often used informally to

---

C. Gamble (✉) • J. Fitzgerald • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk); [john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk);  
[kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

B. Bos  
Chess iX, Haarlem, The Netherlands  
e-mail: [bert.bos@chess-ix.com](mailto:bert.bos@chess-ix.com)



refer to any of these three concepts. Accordingly, this book considers three levels of behaviours during the process of modelling:

1. A model of *ideal* behaviour includes only the core behaviour of a component or system.
2. Models containing *realistic* behaviours, including *disturbances*, are more faithful representations of reality. They include such behaviours as manufacturing tolerances and signal noise. These behaviours may well be found on a component's data sheet, or come from previous experience, and are therefore within the specification of a component (and hence do not represent a failure in the terminology of Avizienis et al. [6]).
3. Models of *faulty* behaviours represent components or systems failing to meet their specifications.

This chapter provides guidance for the addition of realistic behaviours, faults and fault tolerance to a model. It starts by discussing how to identify a set of realistic and faulty behaviours for a model in Sect. 9.2 and offers a method to reduce that set if there is not sufficient resource to model them all in Sect. 9.3. General principles for modelling faults in Crescendo are then discussed in Sect. 9.4. The second part of the chapter starts with a note on coverage of fault tolerance mechanisms in Sect. 9.5 and then discusses the inclusion of fault tolerance mechanisms in a model using design patterns in Sect. 9.6. Afterwards, the application of these techniques is shown in the line-following robot in Sect. 9.7 and the ChessWay in Sect. 9.8. Finally, Sect. 9.9 provides a summary of the contents of the chapter.

## 9.2 Fault Identification

If a component is well known through use, then its real behaviours and faults may also be well documented. However, if a different component is being used or a component is being used outside its normal usage envelope, then we need a means to explore what deviations from ideal behaviour might occur. The suggested method is to employ a set of guidewords to inspire thinking about this set of deviant behaviours. There are two sets of guidewords presented here, HAZOP [86] (*Hazard and Operability study*, Table 9.1) and SHARD [80] (*Software Hazard Analysis and Resolution in Design*, Table 9.2); the choice of which to use is dictated by where the output of a component is used. The HAZOP set of guidewords is widely used in industry and is well suited to considering deviations in physical processes. The SHARD guidewords were inspired by HAZOP and are better suited to the electrical components providing or using services. So if a component represents a sensor that is read by the controller, then SHARD should be used; if the component modelled is within the plant, such as a hydraulic pump, then HAZOP would be more appropriate.

The first step when using guidewords is to identify the significant components that exist in the model. These components should be small enough that their behaviour can be understood, but not so small that their effects are trivial or need to

**Table 9.1** The HAZOP set of guidewords with examples

Guideword	Meaning	Example
No or Not	Complete negation of design intent	A motor that is frozen or gives no power
More	Quantitative increase	More voltage/current than expected is supplied to a component
Less	Quantitative decrease	A sensor that is less sensitive to some property than expected
As well as	Qualitative modification or increase	Extra actions, switch bouncing
Part of	Qualitative modification or decrease	Missing actions
Reverse	Logical opposite of design intent	Relay switches off instead of on for a given signal
Other than	Complete substitution	A motor catching fire when energised
Early	Relative to the clock time	–
Late	Relative to the clock time	–
Before	Relating to order or sequence	–
After	Relating to order or sequence	–

**Table 9.2** The SHARD set of guidewords with examples

Guideword	Meaning	Example
Subtle	Value is incorrect but is plausible	A rotary encoder missing a count each turn
Coarse	Value is detectably wrong	A bit flip in a register
Early	Value/message is early	A sensor responds quicker than expected
Late	Value/message is late	A sensor is slow responding to the environment
Commission	Value/message is sent when not expected	A switch bouncing
Omission	An expected value/message is not received	A sensor fails silently

be combined with multiple other components before it may affect the behaviour of the system. For example, a servo motor could be considered as a single component, rather than as a motor or a gear train, components that convert a signal to a voltage and the sensor that feeds back motor position.

With a set of components in place, the next step is to define the ways in which each component affects its environment. These are not only the inputs and outputs of the component but could also be its physical presence. Returning to the example of the servo motor, the inputs include the electrical voltage and speed/position signal; the output is the rotation/position of the output shaft.

The final step is to apply each guideword to each input/output of each component and to consider whether that input/output could deviate from the ideal behaviour in the manner described by the guideword. In the case of the servo component, the “more” guideword applied to its rotational output could mean more speed or more torque. An application of the “less” guideword could be applied to the voltage input resulting in lower speed or it could represent a lower electrical resistance in the motor due to a short circuit if such concepts are of interest to the model. It should

be noted that not all guidewords will apply to all components and inputs/outputs. An example of a guideword that does not apply would be the “after” word when thinking of the servo output shaft as there is no sequence of events for the shaft. At the same time, it is possible that a guideword may yield more than one failure behaviour. An example here could be a light sensor, where the “subtle” guideword could result in an error due to ambient light affecting the value, analogue-to-digital noise affecting the value or the effects of temperature affecting the sensor’s output.

### 9.3 Fault Selection

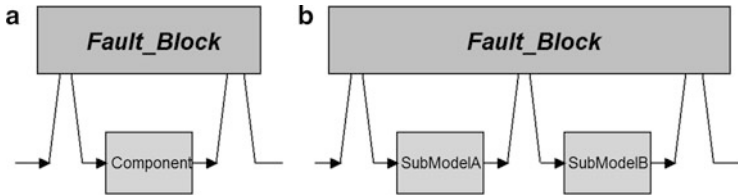
If there is not sufficient resource to model and analyse all faults determined by the previous step, then a subset of the faults that will be modelled must be derived. One method that can be used to produce an ordered list is to assess each fault using a modified version of the Failure Mode Effects Analysis (FMEA) technique. In FMEA, three properties of each failure mode are considered, the probability of occurrence, the probability of detection of occurrence and finally the severity of the effects of occurrence. From this, a value can be derived indicating the level of risk each failure mode poses in the system as it stands. For our purposes, we are only interested in the likelihood that a fault will occur and the estimated severity of the effects it could cause. We do not consider detection at this point, as even if a fault could be easily detected, if it could be a source of significant problems, then we should model it so that fault tolerance mechanisms in the models can be exercised.

The two properties of interest, occurrence and severity, are given a value from one to ten for each identified fault. For occurrence, a value of one indicates that we have no experience of this fault occurring in similar components or processes, while a value of ten indicates that it is almost inevitable that it will occur in a short period of time. Similarly for severity, a value of one indicates that the fault occurring would have no effect on the system, while a value of ten indicates very severe consequences including complete failure of operation, injury or death.

The score for each failure mode is calculated by multiplying the occurrence and severity figures together. When all scores are calculated, the failure modes can be placed into an ordered list according to the scores, where a higher score implies a higher priority. This list may then be used to guide modellers when choosing which faults to model.

### 9.4 Fault Modelling

Each faulty or realistic behaviour that was determined to be significant by the previous process should be included in the models. When including fault behaviour in the model, it is important that the introduced fault behaviour is distinct from the desired ideal behaviour. This is to ensure that any faults modelled do not get



**Fig. 9.1** Fault injector pattern. (a) Component wrapped by fault block. (b) Two submodels wrapped by a single fault block

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 Chap. 6). 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.

There is no concept of inheritance in 20-sim models, and so a different operation must be employed to add fault behaviour to a specific component. We propose the use of a *fault injector pattern* where a *fault block* effectively wraps the component allowing it to intercept and alter both the inputs and outputs of that component. Figure 9.1a 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, then 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, Fig. 9.1b.

The 20-sim feature permitting each block to have multiple implementations is used here (see Sect. 6.6.2). 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 modeller is then able to select the implementation of the fault block required for the simulation to be performed.

A collection of design patterns may be found in Appendix C.

## 9.5 Fault Tolerance Coverage

The list of faults selected for modelling can be used to assess the coverage of fault tolerance mechanisms. For each fault on that list, it should be possible to describe the mechanism and location in the design where it will be handled; this information can be appended to the list. This will yield a list of fault tolerance elements that should exist in the model. An example of such an element would be a voting mechanism and the use of redundant sensors measuring the same property, Fig. 9.2. This element could be said to be mitigating the risk of an individual sensor failing. If a fault will have no fault tolerance elements attached to it, then a justification of why the fault is not handled can be recorded instead.

## 9.6 Fault Tolerance Modelling

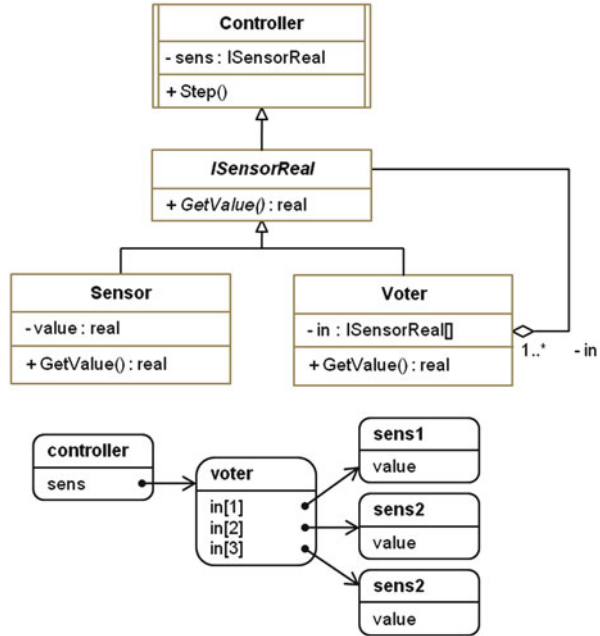
Unlike fault modelling, the inclusion of fault tolerance in a model should not be considered “pollution” of the model. At the same time, there is still an argument for keeping the definition of fault tolerance mechanisms and the description of the “normal” operation of the model distinct as this may reduce the complexity of maintaining both of them. Again we suggest the use of design patterns here as these help make the choice of mechanism explicit and can also reduce effort and increase confidence through the reuse of an existing design. The example of the voting mechanism from the previous section is used here also. In this pattern, Fig. 9.2, the use of a voter object that contains references to the actual sensors and also exhibits the same interface as the sensors means that the voter may be used in place of a single sensor seamlessly while also adding fault tolerance. A set of fault tolerance patterns may be found in Appendix C.4.

## 9.7 An Example Using the Line-Following Robot

Both the modelling of faults and fault tolerance explained above can be illustrated using the line-following robot. After the initial models of the line-following robot were constructed, the faults and fault tolerance that the robot might exhibit was explored. Following the guidelines, the first step was to utilise guidewords to help elicit a first set of faults to consider. The focus of this was on the two sensor types included in the model, the infrared reflectivity sensors for line-following and the wheel position encoders for distance measurement. A group of engineers sat down and considered the application of each SHARD guideword to each sensor in turn; the sanitised results of this work can be seen in Table 9.3.

If we take the example of the infrared reflectivity sensors that are crucial to the line-following robot’s ability to follow a line, then we can identify four distinct

**Fig. 9.2** Class and object diagrams of the voter pattern

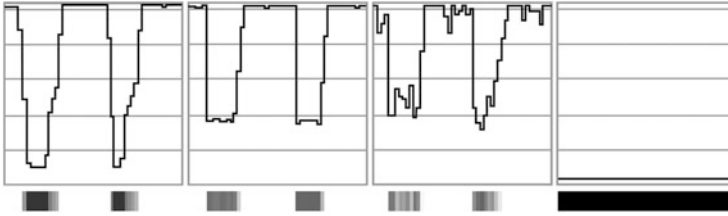


**Table 9.3** The results of applying SHARD guidewords to the R2-G2P robot

Guideword	Line-follow sensor	Wheel position encoder
Subtle	A/D conversion noise, incorrect reading due to ambient light	Missing or additional counts each turn
Coarse	Fail at a fixed value	A bit flip in a register
Early	Sensor responds quicker than expected	None found
Late	Sensor slow to respond to reflectivity change	None found
Commission	None found	Non found
Omission	Broken connection to sensor	Failure to count position pulses

behaviours that are significant. Figure 9.3 shows examples of the output of a sensor on a graph as the sensor passes over two black lines (shown below each graph). The first graph shows what we might consider to be the ideal behaviour of one of these sensors; here we can see the sensor’s output is uniformly high when the sensor is over a white region and it drops down to a very low value when the sensor is over a black region. If such an ideal sensor were available, then it would be a very simple task to follow a line.

The second graph represents one of the realistic behaviours identified for these sensors. Here the issue is that in bright conditions, the sensor will never return a very low value to the controller as there is always a significant amount of infrared light falling on its receiver. In our model, the addition of ambient light has no discernible



**Fig. 9.3** Graphs of sensor output behaviour, *left-to-right*: ideal behaviour, ambient light added, A/D conversion noise added, total sensor failure

effect when over a white region, but when over black, we see the value returned to the controller is higher than with the ideal sensor.

The third graph shows the effect of adding another realistic behaviour, that of electrical noise when performing A/D conversion. The combination of both ambient light and electrical noise reduces our confidence that we can determine if the sensor is over a black region or a white region from a single reading alone.

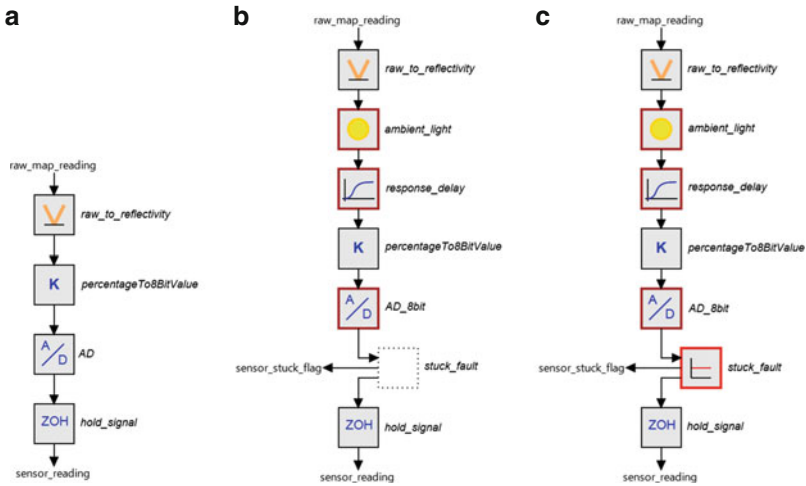
Finally, the fourth graph represents the sensor output in the case of a total failure of the sensor; in this case, the sensor constantly returns a reading indicating black under the sensor.

The sensor was implemented using a series of blocks within the CT model. The original sensor model, without any fault included, is shown in Fig. 9.4a. It contains a block to convert the colour of the floor at a point to a reflectivity value; the reflectivity is converted to an 8 bit value (0–255) before being sampled for consumption by the controller.

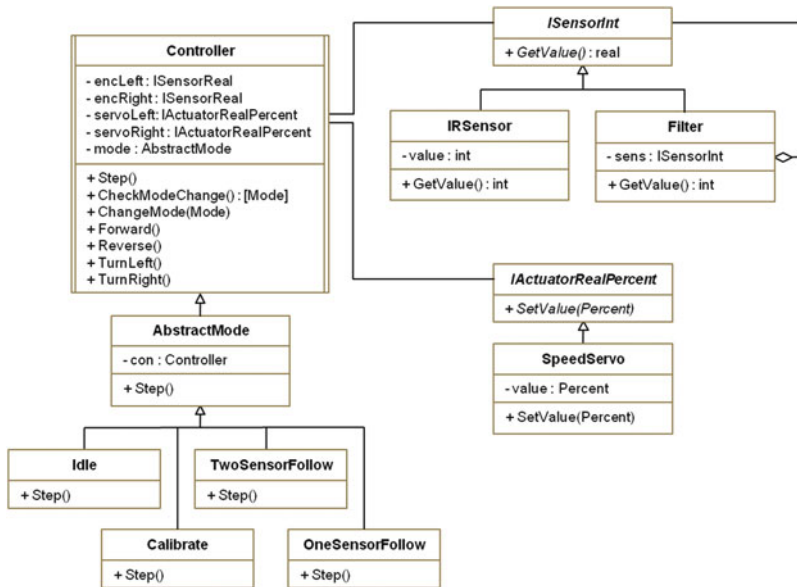
Figure 9.4b shows the block model of the sensor with blocks adding fault and realistic behaviour added. The `ambient_light` block adjusts the correct value output from the reflectivity block so the extra IR in the ambient light is accounted for. The small time for the sensor to respond to a change in the measured IR is modelled in the `response_delay` block. A/D noise is added in a modified `AD_8bit` block in which Gaussian noise is added into the signal before it is quantised to an 8 bit value. Finally, there is a `stuck_fault` block, with an icon indicating to us that the fault is not currently active. This empty box icon along with the sensor response and ambient light icons were all custom designed during the construction of the model and allow the differentiation between library blocks supplied by 20-sim and bespoke equation blocks built for the project.

Finally, Fig. 9.4c shows the sensor mode with the `stuck_fault` active. The fault was activated by selecting a different implementation of the block; in this implementation after a specified time during the simulation, the sensor value computed in the model is replaced by a fixed value to model the effect of the sensor output sticking. The icon is also changed by selecting the active fault implementation; this helps the modeller to visualise the state of the model.

Figure 9.5 shows a simplified class diagram of the fault controller used in the line-following robot, which contains mechanisms targeting each of the above faults.



**Fig. 9.4** Infrared reflectivity (line-following) models. (a) Original sensor model with no faults. (b) Model with ambient light, A/D noise, sensor response time and stuck value fault added, but not active. (c) Model with all faults and with the stuck value fault activated



**Fig. 9.5** Simplified class diagram of the fault-tolerant controller in the line-following robot



The basic line-following behaviour is contained within the `Controller` and the `TwoSensor` classes, which could follow a line if presented with the output of ideal sensors. The problem of ambient light is addressed by the `Calibrate` mode class. When the robot is first switched on, it expects one of its sensors to be over a black region and the other to be over a white region. The controller, after a brief time in the `Idle` mode, moves to `Calibrate` mode, where it proceeds to take a series of readings from each sensor. These readings are used to determine what values represent a white region and what represents black. The results of calibration are passed to the `TwoSensor` mode, which uses them for line-following operations.

The problem of A/D conversion noise is addressed by extending the `IRSensor` class to implement the `Filter Pattern` (see Appendix C.4.2). This filtered sensor returns a floating mean sensor reading to the controller instead of the value most recently read. This has the effect of reducing the variation of the signal due to the random noise and therefore increasing the confidence in decisions which rely on that value. The cost of such a floating mean is the response time of the sensor to a real change of value increases and so this may be detrimental to performance.

Finally, when the total failure of a sensor is detected, the controller shifts to the `OneSensor` mode. This mode utilises the one sensor that remains working to continue following the line, albeit with a slightly degraded performance in terms of the speed at which the line is followed.

## 9.8 An Example Using the ChessWay

There are two controllers in the `ChessWay` personal transporter, each with access to a different set of sensors. These controllers act in a distributed fashion, each controlling one wheel. The `ChessWay` is designed for humans to ride on, and therefore safety aspects are particularly important as explained in Sect. 7.3. For example, the controllers have a start-up mode that stops the handlebar moving violently when the device is lying on the ground. The controllers must also shut down the device if the safety key is pulled out (for example, if the user falls off).

The two controllers communicate over a data connection in order to access the sensor data (from the other controller) and synchronise their behaviour. Studies on prototypes showed that this data was often corrupted, failing a Cyclic Redundancy Check (CRC) and indicating that errors had occurred in transmission. In terms of fault identification (see in Sect. 9.2), the SHARD guidewords are applicable here. The corruption of messages is a course failure, since the controllers can detect an error using the CRC. If the message needs to be resent, then this can be viewed as the late arrival of the message. If further messages are sent as well as repeated messages, then data can arrive out of order.

It is important that the controllers can handle these communication errors. For example, the controllers rely in particular on accurate and up-to-date readings from the gyroscope, and acting on late or out-of-order information could lead to

dangerous oscillations or violent movements. This is one aspect that the engineers wanted to address at an early stage in design.

From a modelling perspective, the problem of corrupted communications is an interesting one, because the communication mechanism spans the border between software and physical systems. There is clearly a physical element to the communications, since electrical circuits are used to transmit the messages. In fact, the engineers suggested that the cause of the corrupted messages was most likely electrical interference from the powerful motors driving the wheels of the device. On the computing side, the messages are created and interpreted at a software, typically passing down through various transport layers, before eventually being converted to analogue signals to pass along the wire.

So this raises the question of where in the co-model to represent these errors (corrupted messages). Such questions often arise for components such as the sensors and actuators that also bridge the software and physical sides, as in the robot example in the previous section. The answer of course depends on the purpose of the co-model and the level of abstraction. In the case of the ChessWay, there was no model of ideal behaviour for communications on the CT side, so modelling the fault there would require a lot of extra modelling. On the DE side, however, the model already used the features of VDM to model two CPUs communicating via a shared bus; therefore, this seemed an appropriate place to model the fault. In addition, the various abstractions and structuring mechanisms of VDM lend themselves to modelling the fault.

Of course there are circumstances where a CT model of such faulty behaviour may be appropriate, if the nature of the electrical interference and its specific effects on messages was of interest to the engineer. In the case of the ChessWay example, however, the engineers felt a more stochastic approach was sufficient, where the effects on the controllers of dropping a certain percentage of messages are simulated (e.g. 5 %, 10 % and so on). This level of abstraction matched that of the initial co-model with normative behaviour.

In order to model the loss of messages, the *Ether* pattern was developed for and applied by the engineers in the ChessWay study (see Appendix C.3.1). The pattern permits the modelling of realistic communications between networked controllers. In VDM, the CPU and BUS abstractions can model communications that take time, but that are ideal in the sense that no messages are lost, duplicated or corrupted. The ether pattern introduces an explicit model of an *ether*, which represents a medium through which data must travel.

The pattern is general enough that it can be applied to represent direct connections between controllers, a connection through a wired system such as an ethernet or wireless communications. The implementation of the ether can be tailored to different guarantees offered by different media, ranging from having no quality of service guarantees to one where each message will arrive once and only once with all messages in order.

Figure 9.6 shows a class and object diagram for the ether pattern. The `Ether` class represents the communication medium: messages are passed into the ether and may or may not then arrive at the destination. The ether knows about components

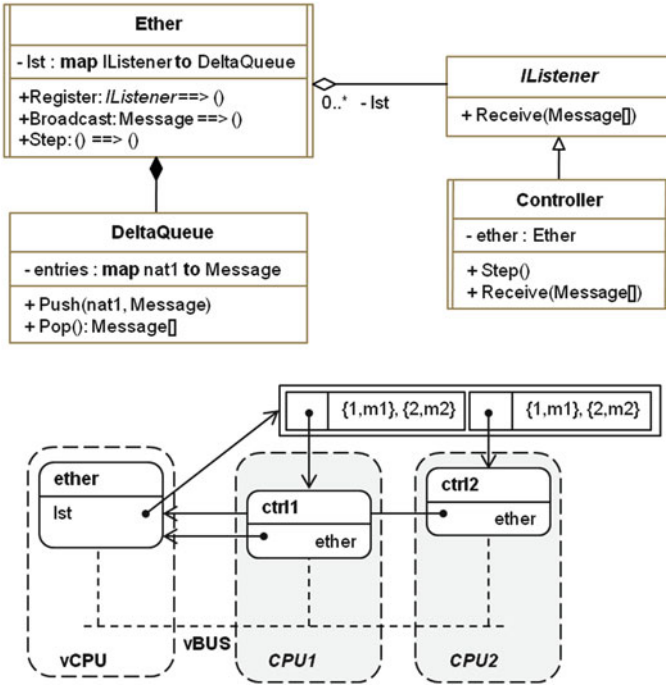


Fig. 9.6 Ether pattern class diagram (top) and object diagram (bottom)

that are connected to it (they must implement the `IListener` interface in order to interact with the ether), and those components can send messages by calling the `Broadcast` operation. When applying the pattern, point-to-point communications could be modelled by giving assigning identifiers to connected components; however, in the ChessWay case, this was not necessary as there were only two controllers.

To model the passing of messages, the ether holds the messages in a map of queues (one for each controller connected to the ether). The class is active, defining a thread that periodically updates each queue to indicate time passing. The `DeltaQueue` class models message travel time: messages are placed into the queue with a given delay to be simulated; this value is reduced as simulated time passes until it reaches zero, at which point the message is passed to the recipient.

The object diagram shows an ether object deployed on a virtual CPU, with two controllers on separate (real) CPUs, all connected by a virtual bus. The use of the virtual CPU and bus for the ether ensures that its use does not affect the (simulated) execution time of the controllers.

Communication errors modelled by creating subclasses of `Ether` override the `Broadcast` operation. In the case of the ChessWay example, a subclass is defined that took as a parameter a percentage of messages to drop. Then when `Broadcast`

is called, a pseudorandom number generator is used to decide if a message should be dropped (not placed on the queue).

Other subclasses could be created to model different errors. For example, corruption could be modelled by extending the message type. This would involve allowing the message to have a value indicating that it is corrupt. Again the `Broadcast` operation could be overridden to select which messages should be corrupted. To model injection of messages (in a “hostile” ether), the ether or an adversarial object could be created that places messages on the queue that were not sent by one of the components. In order to explore the design space with different quality of service guarantees, subclasses of `Ether` could provide different levels of service with associated costs in terms of transmission time, and co-simulations run with each of these implementations to observe the consequences.

## 9.9 Conclusion

We have described the use of SHARD and HAZOP guidewords to help identify a set of faulty/realistic behaviours that a component might exhibit and shown how a modified FMEA process allows prioritisation of these faults. The *fault injector* pattern allows fault modelling to be distinguished from the desired normal behaviours. Examples of faults and fault tolerance were outlined for both the line-following robot and ChessWay case studies. Finally, we discussed the use of known design patterns within the software controller to handle these faults.

Design for dependability is essential for the increasing range of embedded systems with rich digital control. From our experience developing and applying the co-modelling framework, the sharing of both fault models and resilience mechanisms as patterns is likely to prove highly beneficial in managing the additional design complexity introduced by the need for confidence in correct, safe and secure functioning of embedded systems.

# Chapter 10

## Design Space Exploration for Embedded Systems Using Co-simulation

Carl Gamble and Kenneth Pierce

### 10.1 Introduction

The purpose of conducting simulation experiments is to help select a design that best meets the requirements for a system, where best means better than the other candidate co-models rather than necessarily meeting them all. The problem that a designer may face at this point is that the number of potential designs available can far outweigh the number of simulations that can be performed. In these situations, some tactics must be employed if the designer is to have any confidence that the final design selected represents a good choice either in terms of compliance with the requirements or simply that sufficient alternatives have been considered.

There are two key concepts relating to this: (1) the *design space* and (2) the *response surface*. If we consider a design with two parameters that we may vary, then the cartesian product of all values each parameters may take is termed the design space. It represents all possible variations of a design. If we assume that we are able to estimate the performance of the design for each pair of parameter values and if we then plot those points, the shape we end up with is termed the response surface. This surface can guide the engineer to select a design that is optimal with respect to that performance measure. The following sections provide guidance on exploring this design space and assessing the performance of the design at those points.

Two case studies are presented in this chapter. The line-following robot is used early on to demonstrate the use of the Automated Co-model Analysis (ACA) features of the Crescendo tool. In this example, we explore the effects of moving the infra-red line following sensors on the speed and accuracy of line following along with the energy consumed by the robot. Then near the end of the chapter, the

---

C. Gamble (✉) • K. Pierce  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk); [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk)

automated tractor case study, similar to the one introduced in Chap. 1, is used to show how orthogonal matrices may be used to reduce the number of experiments needed to obtain a near optimal result. Here the experiment is used to find the optimal speed the tractor may drive given the ground conditions and location of its centre of gravity (CoG) due to any load carried.

The chapter begins by introducing the ACA feature of the Crescendo tool in Sect. 10.2, before presenting their use in the line-following robot in Sect. 10.3. The chapter then moves on to discuss issues that arise when the design space is larger or more complex. Section 10.4 discusses the identification of parameters for a simulation experiment, and Sect. 10.5 introduces approaches that may be taken to reduce the number of simulations performed while maintaining some confidence of discovering a good set of parameters. Section 10.7 presents the automated tractor example, and Sect. 10.8 discusses the issue of choosing a design based upon multiple attributes. In Sect. 10.9, we return to the line-following robot considering more sensors using the exploration ideas presented for the tractor example. Finally, Sect. 10.10 summarises the material from this chapter.

## 10.2 Using ACA

### 10.2.1 *How ACA Works*

ACA, as introduced briefly in Sect. 2.6, is a feature that enables running many different co-simulations with minimal user intervention. The ACA feature enables the user to select different configurations for each individual part of the co-model and then runs the co-simulation combining all possible configurations that were selected by the user.

It is possible to construct complete configurations by combining different partial configurations. Figure 10.1 together with the following description helps to illustrate the concept. The result of generating complete configurations from the partial configuration would be four 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 values to existing parameters, for example, simply adding an A2 value would result in four more different configurations.

### 10.2.2 *Configuring an ACA Launch*

An ACA debug configuration is used to describe the aspects of a model that we wish to vary during the series of simulations, and it is created by selecting ACA Launch and then pressing the New button, see Fig. 10.2.

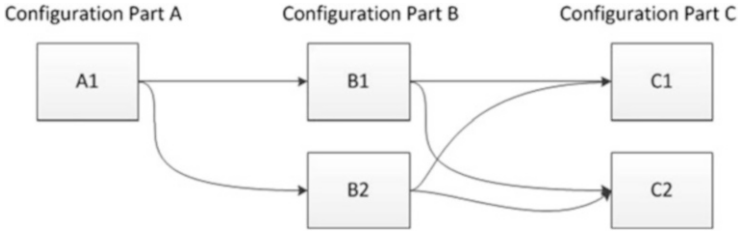


Fig. 10.1 How configuration parts are combined with ACA

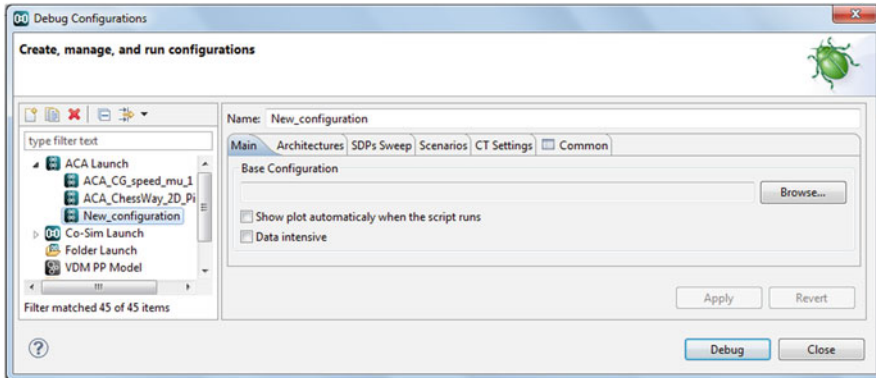


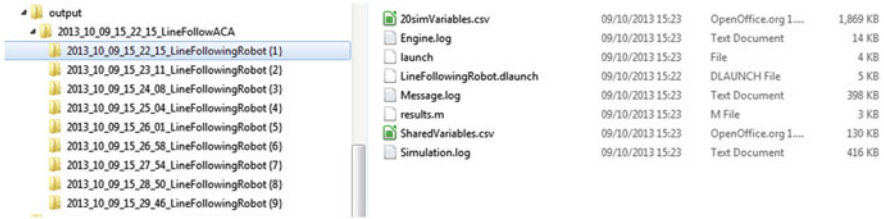
Fig. 10.2 Creating an ACA debug configuration

There are several options that may be configured, which will be outlined here to give an overview of what is possible with the ACA feature. For a more complete description of the features, one should consult the Crescendo user manual [60].

Each ACA debug configuration must have a reference to an existing “normal” debug configuration defined in its `Main` panel. From this, it learns the basic parameters of the simulation such as the base project, CT model, simulation time and so on.

The `Architectures` panel can be used to choose which configurations of the DE model to sweep over during the simulation. This is useful for exploring the effects of deploying the controller software onto different hardware configurations (CPUs with different speeds, different numbers of CPUs and so on) or employing different controller implementations. When multiple architectures exist in the base project, they will be listed on this panel and those architectures to be included in the parameter sweep may be selected.

The `SDPs Sweep` panel allows the user to define sets of values each SDP should adopt during the simulations. They can be defined in two ways. The first involves setting a `from` value, a `to` value and the step to increment by, the Crescendo tool then computes all values the parameter should adopt. The second method allows the user to explicitly define each value an SDP should adopt.



**Fig. 10.3** Structure of results from an ACA co-simulation run

The `Scenarios` panel lists all the script files included in the base project. If any exist, the user may select one or more of these to be included in the run of simulations.

The `CT Settings` tab is similar to the `Architectures` panel above, except that it allows options within the CT model to be selected. For example, if there are multiple implementations defined for submodels within the CT model, these may be selected for inclusion in the parameter swept over allowing their effects to be seen in the results. This is another way for activating faults in the CT model by selecting both faulty and non-fault submodel implementations.

A word of caution: each parameter defined above that is to be included in the sweep adds to the number of simulations that must be performed. For example, if there are two architectures, three SDPs each with three values, two scripts and two CT submodels each with two implementations, then this will require  $144(2 \times 3 \times 3 \times 2 \times 2 \times 2)$  simulations to be performed. Methods for reducing the number of experiments performed are discussed later in the chapter.

When the experiment is complete, Crescendo stores the simulation results in much the same way as for a single simulation (see Sect. 5.4) but with two key differences. The first difference is a two-level directory structure. A directory is created to contain the ACA run under the output directory and then the results for individual simulations are stored in their own subdirectories within it (Fig. 10.3). If the design space is large, then finding the results of a specific simulation could take some time, so the second change for ACA generated results is the addition of an index. The `index.html` file stored in the ACA results folder (not shown) contains all of the settings used for each simulation and so allows for searching for a specific simulation by SDP settings. Next to each experiment's settings is a hyperlink to the directory containing the results for that simulation, making it easy to locate the desired results.

### 10.3 An Example Using the Line-Following Robot

An example of using the ACA functionality is an experiment performed on the line-following robot to determine if the initial positions of the line following sensors were optimal for line-following performance. This raised the question of what



we meant by “performance” and how were we going to measure it. After some discussions, we identified four metrics that we believed would vary when altering the robot sensor positions:

**Distance travelled:** Since the simulations were all 30 s long, the robot that travelled the furthest distance in that time would have the highest mean speed and so would be considered better. This purely considers the speed the robot travelled and not the accuracy of line following which is addressed by the final two metrics.

**Energy consumed:** A robot that uses a smaller amount of energy would have lower running costs and so would be considered better.

**Deviation area:** Measuring the area between the line and the path the robot reveals if a robot deviated from the line for long periods of time.

**Maximum deviation:** Measuring the maximum deviation reveals if a robot design made a big departure at some point.

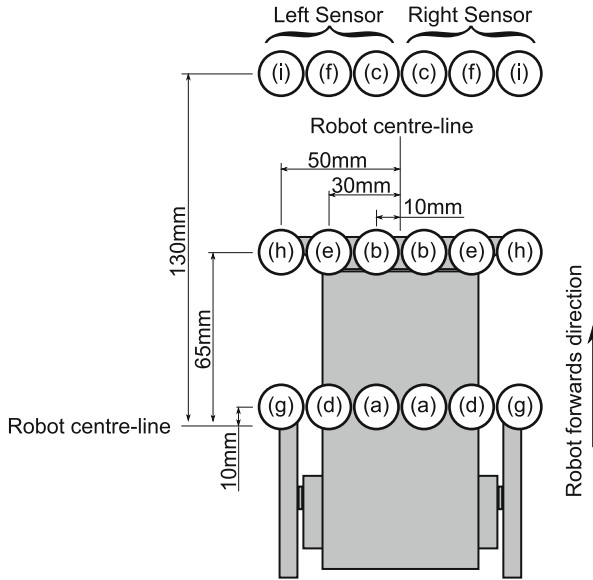
The model was set-up to provide the information required to facilitate making these measurements. To compute the energy used by a robot, we took advantage of the bond-graph basis of the robot model. There is a bond-graph element, an `EnergySensor`, that may be placed between any two elements and which outputs the energy that has been transferred between them. Placing one of these in each of the servo submodels made computing the total energy consumed by a design almost trivial. To compute the remaining metrics, we configured Crescendo to output the path followed by the robot as a sequence of co-ordinates. These sequences were processed after each simulation to obtain the distance travelled, deviation area and maximum deviation.

The experiment involved altering the positions of the sensors literally in both the lateral and longitudinal axis when looking at a plan of the robot. The actual positions of the sensors on the real robot are 70 mm in front of the wheel centre line and 10 mm either side of the robot centre line; this is represented by position “(B)” in Fig. 10.5 and Table 10.1. Two alternative values were selected for each axis, 10 and 130 mm longitudinal positions and 30 and 50 mm lateral positions. This means three values for two parameters and so there are nine different configurations to simulate and assess. Each combination of lateral and longitudinal was given an identifying label, (a)–(i); the positions of the sensors for each experiment are shown graphically in Fig. 10.4 and tabulated in Table 10.1.

To enable the experiment, the robot co-model was altered to make the lateral and longitudinal positions of the sensors a shared design parameter. Once done this allows us to make use of the ACA parameter sweep of the Crescendo tool. Figure 10.5 shows the `SDPs Sweep` panel in the ACA debug launch for the robot experiment. On the screen you can see the two parameters that are to be swept, their starting values, the increment between experiments and the final values.<sup>1</sup>

---

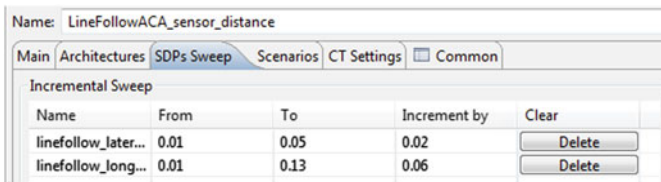
<sup>1</sup>The use of the terms “starting” and “final” here only refers to the sequence of values each parameter will adopt. The simulations themselves may not be performed in the same order.



**Fig. 10.4** Graphical positions of the sensors overlaid on a plan view of the robot

**Table 10.1** Lateral and longitudinal sensor offsets with their designated identifiers, (a)–(i)

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



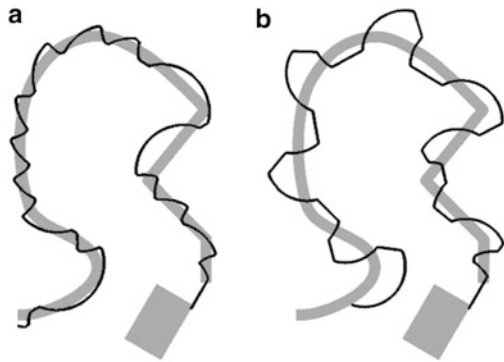
**Fig. 10.5** Defining the SDP parameter sweep in an ACA launch configuration

The results of the simulation are shown in Table 10.2. Here we can see the rankings each robot design achieved for each of the four metrics, and we can see that the designs perform differently for each metric. For example, design (b) attained the greatest distance during its simulation but was outranked by design (i) for energy consumed and design (f) in terms of maximum deviation. For this reason, we needed a way of combining the individual metrics to obtain an overall rank for the designs. For this experiment, the overall rank was trivially determined by taking the mean of the four individual ranks; this value appears in the final column and indicates that design (b), the original design, gives the best overall performance. This ranking

**Table 10.2** Ranking positions for each metric along with the mean ranking position to give an overall position. Metrics are: *A*: distance travelled; *B*: energy consumed; *C*: deviation area, and *D*: maximum deviation

Rank	Design	Metric				Mean rank
		A	B	C	D	
1	(b)	1	5	1	2	2.2
2	(f)	7	2	4	1	3.5
3	(a)	2	8	2	4	4.0
4	(e)	3	6	3	5	4.2
5	(i)	9	1	5	3	4.5
6	(c)	5	3	6	8	5.5
7	(d)	6	4	7	7	6.0
8	(h)	4	7	8	9	7.0
9	(g)	8	9	9	6	8.0

**Fig. 10.6** Paths followed by two robot designs during the ACA experiment. (a) Design (b). (b) Design (h)



assumes that all the metrics are of equal importance to the stakeholders, which is likely not to be the case and so the subject of multi-attribute decision making is discussed later in Sect. 10.8.

Saving the path of the robot allows it to be overlaid on the modelled line to help understand why one design performed differently to another. Figure 10.6 shows plots of the paths followed by designs (b) and (h). It helps to visualise why, for example, the deviation area for (h) is at the opposite end of the ranking scale to (b) and can be useful when explaining ACA results to non-technical stakeholders.

### 10.4 Candidate Parameters

It may be the case that the design space is more complicated than the simple positioning of two sensors that we saw in the previous section. In such situations, it is important to have a clear picture of what parameters will be varied and how many values each may adopt. The number of parameters to be varied and the number of values each parameter can take have a direct effect on the number of simulation experiments that will be needed and so a judgement should be made regarding the necessity of each new parameter or value.

Two characteristics may be used to classify parameters, one characteristic considers whether a parameter is part of the design, which can be controlled or part of the environment, which cannot be directly controlled by the designer (this can also include potential faults in case tolerance against these are desired). The other characteristic divides the nature of the parameter. These parameters can be divided into two groups: continuous parameters, such as the separation between two sensors, and parameters with discrete values/design alternatives, such as the selection of an electric motor from a catalogue. These characteristics are expanded below.

### ***10.4.1 Design or Environmental Parameters***

#### **10.4.1.1 Design Parameters**

These are the parameters that the designer defines, perhaps within a limited range, and which characterise the object being designed. These parameters may be represented by a single value, for example, the diameter of a wheel, but they may also represent choices of distinct design alternatives, such as the choice of either infrared or ultra-sonic sensors for a particular *purpose*.

#### **10.4.1.2 Environmental/Uncontrollable Factors**

A category of parameters that should be accounted for when designing an experiment are those that either represent the environment in the simulation or undesirable events such as faults occurring within the product being simulated. Environmental parameters could include the ambient temperature or light level during the test, the nature of the terrain a vehicle would need to traverse or the behaviour of other actors such as extra robots. Fault activation parameters include whether a fault becomes active during a particular simulation run, when the fault becomes active, the severity of the fault and the duration of the fault being active if it is transient.

### ***10.4.2 Nature of the Parameters***

#### **10.4.2.1 Continuous Parameters**

These are the parameters that can be varied according to the designer's will and can be placed in an ordered list. The values are either real (floating point) or natural (integer) numbers and represent the quantification of some property. Examples could include the length of a beam, the capacity of a battery, the sampling rate of a sensor or the number of sensors contained in a particular design.

For each of these parameters, there are two decisions that need to be made. The first determines the range of values, minimum to maximum, that each parameter will vary over and this must be guided by the designer's knowledge of the system at hand. The second decision is the choice of how many values to test with the chosen range. This second decision is a trade-off between choosing a higher number, which will better reveal the nature of the surface response to this parameter, and choosing a lower number, which will reduce the number of experiments that will need to be performed.

It may be the case that while a variable may be adjusted continuously, it may only be practical to produce a product with specific values within that range. A mechanical engineering example would be the use of aluminium extrude beams, which are generally available in a range of specific sizes from each supplier, an example from electrical engineering would be the available sizes of resistors. In such cases, choosing values that match the available components will yield simulation results that better represent the candidate design in reality.

#### **10.4.2.2 Parameters with Discrete Values/Design Alternatives**

There may be design parameters that do not naturally sit in an ordered list. Alternative designs for a product's control software would be an example here. There are two ways in which parameters of this type may be considered when designing a set of simulations to perform. The first way is to treat each design as a distinct value for some parameter, then each design can be represented equally within the set of experiments performed. The second way is to determine a set of properties that describe the differences between the set of designs. The next step then is to design a set of simulations that yield a good spread of these property values rather than one that tests each design alternative. A simple example would be the choice of a battery pack for a mobile device, and there may be many alternative batteries leading to a large number of simulations; however, it would be possible to describe each battery pack using its capacity and number of cells. In this way duplicating experiments employing, for example, 1,800 mAh three-cell battery packs of different manufacturers could be avoided.

### **10.5 Experimental Design**

The first step when designing an experiment is to define the *purpose* of that experiment as this may provide a guide as to which parameters must be varied and if any should remain static. For example, if the purpose of the experiment is to verify that a model complies with the requirements, then the parameters representing the design will be fixed, while parameters used to represent the environment and usage of the product under test will be varied, possibly focussing on the worst-case conditions.

After selection of the candidate parameters and the range of values each may adopt, the process of designing the simulation experiments to run can start. It is important at this stage to understand how many experiments can be performed as this is a major factor for the design of the simulation experiment.

A thorough simulation experiment would explore both environmental and fault dimensions as well as the dimensions relating to the design. If this is not possible due to the number of experiments to perform, then the effects of faults could be simulated on a reduced number of the better designs once the bulk of the design space has been eliminated by simulation and ranking.

### ***10.5.1 Screening Experiments***

Screening experiments can help to reduce the total number of experiments that will be performed. Their purpose is to filter out parameters that potentially have little effect on the observed results of a simulation. The simplest form of screening experiments will examine each property in turn, once with a “low” value and then again with a “high” value; those properties that show little or no effect on the results when compared to the effect of the others can be screened out of the main experiment.

### ***10.5.2 Fractional Factorial Experiments***

When it is not possible to perform a complete set of simulations testing all combinations of parameter values, a fractional experiment technique must be used. The goal of these techniques is to determine sets of simulation parameters that give a good coverage of the possible design space. Three alternative techniques are now described.<sup>2</sup>

#### **10.5.2.1 Orthogonal Matrices (Taguchi Methods)**

Orthogonal matrices for various combinations of numbers of parameters and numbers of values for each may be found in tables or generated algorithmically [92].

---

<sup>2</sup>There are currently two methods for executing fractional factorial experiments in Crescendo. The first is to manually launch individual simulations with the required SDP settings. The second involves making copies of a normal debug launch configuration file, one for each simulation required by the experiment. Each of these copied launch configuration files is then edited on the `destecs_launch_config_shared_design_param` line to contain the SDP settings for one simulation. The simulations can now be executed by creating a `Folder Launch` debug configuration and selecting the co-model project and the folder in which the launch files reside.

**Table 10.3** L25 Taguchi table for five parameters with five values

Experiment	Param 1	Param 2	Param 3	Param 4	Param 5
1	1	1	1	1	1
2	1	2	2	2	2
3	1	3	3	3	3
4	1	4	4	4	4
5	1	5	5	5	5
6	2	1	2	3	4
7	2	2	3	4	5
8	2	3	4	5	1
9	2	4	5	1	2
10	2	5	1	2	3
11	3	1	3	5	2
12	3	2	4	1	3
13	3	3	5	2	4
14	3	4	1	3	5
15	3	5	2	4	1
16	4	1	4	2	5
17	4	2	5	3	1
18	4	3	1	4	2
19	4	4	2	5	3
20	4	5	3	1	4
21	5	1	5	4	3
22	5	2	1	5	4
23	5	3	2	1	5
24	5	4	3	2	1
25	5	5	4	3	2

Each column in the tables represents one parameter, and the numbers in those columns represent the possible values the parameter can take. Experiments to be run (rows) are then defined by the values in each column for a particular row. Table 10.3 shows the five parameter, five value table used for the line-measuring robot experiment.

### 10.5.2.2 Space-Filling Search

If the number of simulation runs that can be performed differs from the number defined by an orthogonal matrix, then a space-filling search can be used. An important component of the space-filling approach is the concept of some *optimality criteria*. The criteria represent the experiment designer's purposes for the experiment, such as having an even spread across the design space, avoiding certain regions of the design space or ensuring that all corners of the design space are included. These criteria are represented as an algorithm that can compare two sets of simulation runs to see which best fits that criteria.

A space-filling search starts by randomly selecting parameters for the number of simulation runs that can be performed. An iterative process then begins. In each iteration, the parameters for a new simulation run are randomly generated and a simulation run in the existing experiment design is selected. The optimality of the existing experiment design is then compared to the optimality of that experiment design with the new simulation run substituted for the selected one. If the new experiment design is more optimal, then the substitution is retained, otherwise it is rejected. This process continues until repeated iterations yield no further improvements and the simulation process may begin.

### 10.5.2.3 Parameter Sweeping

It may be the case that the product of multiplying the number of values for each parameter is small enough that it is possible to perform a complete set of simulation runs. In this situation, a parameter sweep experiment, where all combinations of parameters are simulated, may be performed. This gives the most complete set of results and so will result in confidence that the optimal design from that set of parameter values will be found.

## 10.6 Using Folder Launch Configuration

As described in the previous section, there are times when it is not possible to run a complete set of experiments that explore the entire design space offered by the alternatives within a product; in these cases the ACA feature described in Sect. 10.2 can not be used. Instead, we need another mechanism that permits more control over the experiments that will be performed such as allowing the experiments defined by a Taguchi orthogonal matrix to be run.

The Folder launch configuration is the method currently supported by Crescendo that allows detailed control over the experiments to be run at the cost of manual or tool supported construction of launch files. The basic principle is that the user constructs debug configuration launch files, one for each simulation they wish to run, these files are placed into a single folder and Crescendo can then be instructed to run each of the simulations described and saves the results in the same way as for standard ACA. These launch configuration files may be named in any way so long as they have the `.launch` file type extension. Figure 10.7 shows the dialog box used to define a folder launch. It only required two pieces of information, the project the launch relates to so the results can be saved in the right location and path to the folder containing the launch configuration files. Once defined this folder launch configuration can be executed in the same way as any other by selecting the Debug button.

To create the launch configuration files, we suggest starting with an existing debug configuration launch file for the project of interest and making copies of



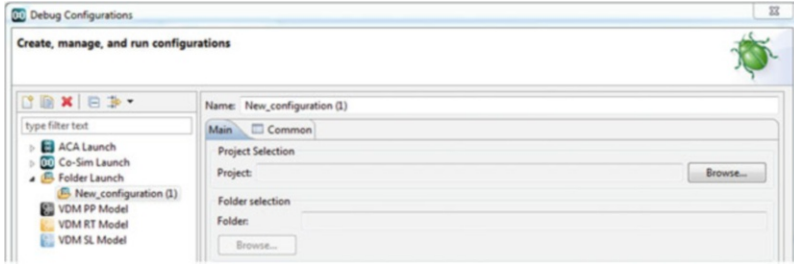


Fig. 10.7 Folder launch dialog box

it in the designated directory. These files have an XML structure and contain all the information required to launch a co-model simulation. While in principle any of the parameters can be changed in these files, it is only necessary to make changes to the `destecs_launch_config_shared_design_param` parameter to implement the experimental designs previously discussed. The value of this parameter contains a list of all the shared design parameters defined in the co-model and their values from the original launch configuration copied. The values associated with each shared design parameter for each experiment can be defined by editing them in this location in the file. Two other parameters can be changed to suit the experimental need. The `destecs_launch_config_simulation_time` parameter defines how long the simulation will run for and can be used to reduce the time spent running simulations if certain parameters are known to require longer or shorter simulations. Finally, the `destecs_launch_config_ct_model_path` can be changed if a project contains more than one CT model and the experiment requires simulating with them.

## 10.7 An Example Using the T1X Tractor

In Chap. 1, the fictional T1X GPS controlled tractor example project was briefly introduced. This example is based upon an independent research project modelling an experimental small tractor shown in Fig. 10.8 using the Crescendo technology.<sup>3</sup> The purpose of this research project is to develop an autonomous controller for the tractor to allow it to traverse fields, following a predefined path using satellite-based navigation systems, while taking into account the position of any loads being carried and in spite of the condition of the soil [23].

<sup>3</sup>See [http://www.froboMind.org/index.php/FroboMind\\_Robot:ASuBot](http://www.froboMind.org/index.php/FroboMind_Robot:ASuBot) for more detail about this project.



**Fig. 10.8** ASuBot: a Massey Ferguson 38-15 garden tractor retrofitted with a Topcon AES-25 steering system

The resulting experiment identified three parameters that will be varied. The first parameter is the speed at which the tractor moves; this has a range of 1–2 m/s and it is assumed that here faster is better. The second parameter to be varied is the position of the CoG of the tractor. As the position of the load carried by the tractor moves rearwards, this has the effect of moving the CoG backwards from its normal position in front of the rear wheels. This rearward CoG shift has a detrimental effect on the tractor's ability to steer, resulting in the potential for understeering or oversteering. In this experiment, the CoG offset ranges from 0 m, indicating the most forward position, to 0.4 m, representing the most rearward position. The third and final experimental parameter represents the coefficient of friction ( $\mu$ ) between the wheels and the surface upon which the tractor is running. A high value for  $\mu$ , 0.7, represents nearly dry conditions, while a lower value, 0.5, represents more wet and slippery conditions.

The simulations will be evaluated by computing the maximum Cross Track Error (XTE) from the ISO 12188-1:2010 standard. The simulation outputs the co-ordinates of the path the tractor took and these are compared with the co-ordinates of the track the tractor should have followed to find out the maximum distance the tractor deviated and this value becomes the XTE for that simulation. An XTE of 0.3 m or less is considered to be acceptable, and the results will be classified accordingly.

The design space has been separated into 21 speed values, 21 CoG offset values and 5 values for  $\mu$ , thus if we wanted to exercise all combinations of the experimental parameters this would require 2,205 simulations. In the case of this

**Table 10.4** Table of parameters used for the tractor experiment, based on Table 10.3, along with the resulting cross track error (XTE) for each experiment

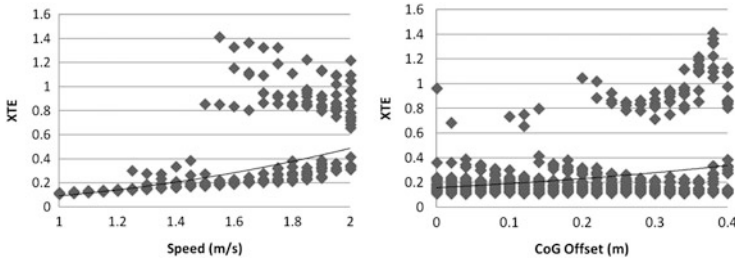
Experiment	Speed	CoG offset	$\mu$	XTE
1	1	0	0.5	0.10948
2	1	0.1	0.55	0.11614
3	1	0.2	0.6	0.11555
4	1	0.3	0.65	0.11124
5	1	0.4	0.7	0.11111
6	1.25	0	0.55	0.14498
7	1.25	0.1	0.6	0.14678
8	1.25	0.2	0.65	0.14528
9	1.25	0.3	0.7	0.13749
10	1.25	0.4	0.5	0.29989
11	1.5	0	0.6	0.18475
12	1.5	0.1	0.65	0.18099
13	1.5	0.2	0.7	0.17424
14	1.5	0.3	0.5	0.19995
15	1.5	0.4	0.55	1.4123
16	1.75	0	0.65	0.22096
17	1.75	0.1	0.7	0.21059
18	1.75	0.2	0.5	0.31918
19	1.75	0.3	0.55	0.80918
20	1.75	0.4	0.6	0.64782
21	2	0	0.7	0.29832
22	2	0.1	0.5	0.73324
23	2	0.2	0.55	0.83668
24	2	0.3	0.6	1.014
25	2	0.4	0.65	0.53127

particular co-model, each simulation took only a few tens of seconds to run and so the full set of experiments could be, and in fact was, performed. For the purpose of this example, two additional ACA experiments were performed to illustrate the principles and to allow comparison of results. The first of these experiments considered 5 values for each of the three experimental parameters and the second experiment considered 21 values for the speed and CoG parameters and retained the 5 values for  $\mu$  as in the other experiments. Complete experimental coverage for each of these additional experiments would require 125 and 2,205 simulations, respectively; however, when we produced Taguchi orthogonal matrices for each of these experiments (Tables 10.4 and 10.5), the number of simulations for these experiments reduced to 25 and 441, respectively. These experiments were executed using the technique described in Sect. 10.6.

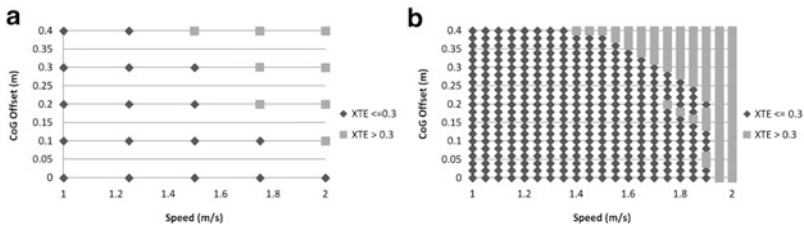
With the experiments performed the next step is to analyse the results. Generally a Taguchi-type analysis is used to determine such trends for multiple parameters and the graphs in Fig. 10.9 show the XTE for the 21-level ACA experiment plotted against both speed and CoG offset. Both tables are annotated with a trend line confirming the intuition that driving the tractor more slowly or reducing the CoG offset will reduce the XTE. It is interesting to note that as the ranking function in

**Table 10.5** Small sample of the parameters for the 21-level ACA along with the resulting cross track error (XTE) for each experiment

Experiment	Speed	CoG offset	$\mu$	XTE
⋮	⋮	⋮	⋮	⋮
19	1	0.36	0.65	0.11216
20	1	0.38	0.7	0.11726
21	1	0.4	0.5	0.11342
22	1.05	0	0.55	0.12259
23	1.05	0.02	0.6	0.11829
⋮	⋮	⋮	⋮	⋮



**Fig. 10.9** Individual parameter graphs for the 21-level ACA experiment



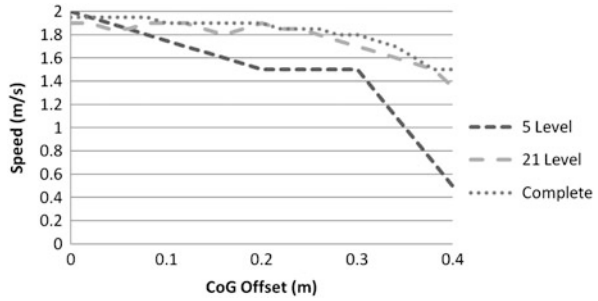
**Fig. 10.10** Scatter graphs indicating which combinations of speed and CoG offset achieved an XTE of 0.3 or less. (a) 5-level experiment. (b) 21-level experiment

this experiment only considers XTE, and that for this error a lower number is better, these graphs indicate that a tractor with 0 m CoG offset travelling at 0 m/s would perform best. This is intuitively untrue as such a tractor would not perform any useful work and highlights the need for either a ranking function that accounts for all relevant properties or a different presentation of the data.

In this case, we may present the data as shown in Fig. 10.10a, b. In these graphs, diamonds represent a pair of parameters, resulting in an XTE of no more than 0.3, while squares represent pairs that failed to meet this acceptance criteria. Here we can note that the results follow the trends indicated in the previous graphs where the tractor is better able to follow the path if it travels slower and also when the CoG offset is lower (more weight on the front wheels).

Parameters for the controller may be found by following lines, horizontally, from the CoG offset axis. Doing this yields the maximum speed that the tractor was able

**Fig. 10.11** Comparing results from the different experiments



**Table 10.6** Factorial experiment errors

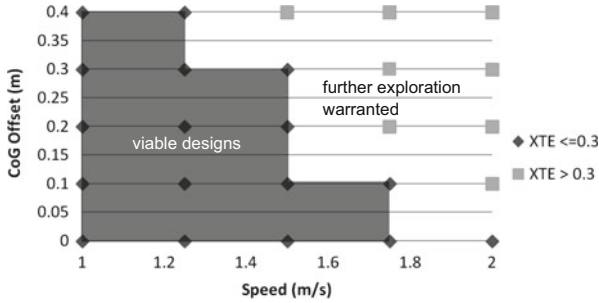
Experiment	Minimum (%)	Maximum (%)	Mean (%)
5 levels	0	66.7	20
21 levels	0	10	3.1

to follow the track for a given CoG offset. Here is where we find the benefit of performing the larger 21-level experiment compared to the 5-level experiment. If we take the CoG offset of 0.3 m as an example, in the 5-level experiment we find the highest speed that resulted in the tractor following the track correctly was 1.25 m/s, while in the 21-level experiment the result is 1.7 m/s. Similarly for a CoG of 0.1 m, the 5-level experiment gives a highest viable speed of 1.75 m/s while the 21-level experiment yields a speed of 1.90 m/s.

Figure 10.11 shows the tractor speed against CoG offset curves that result from both the 5- and 21-level experiments and also the complete sweep over all parameters. We can see that despite it only using 20% of the number of simulations the 21-level experiment is very close to the results found during the complete parameter sweep. Table 10.6 evaluates the errors between the Taguchi-based experiments and the complete sweep. It is to be expected that the 5-level experiments, which explored the design space using only 25 experiments, will have a greater error than the 21-level experiment, but the result of this experiment had a mean error of only 20% compared to the complete experiment while using only 1.1% of the simulations. This shows that such a technique could be fruitful in the earlier stages of design for the rapid selection between design alternatives. The 21-level experiment, on the other hand, achieved a mean error of 3.1% compared to the complete experiment at a cost of only 20% of the simulation time. Again this demonstrates the time savings that can be made using these techniques.

### 10.7.1 An Iterative Approach

In the case of the tractor project, we could use an engineering intuition to predict that increasing the speed of the tractor or shifting it backwards will reduce the ability of the tractor to follow the track. In such situations, it may have been possible to increase the accuracy of the result while maintaining a low simulation cost using an iterative approach.



**Fig. 10.12** Design space divided into areas known to work after the 5-level experiment and where further experiments at higher resolution might be fruitful

In such an approach, we might first perform an experiment such as the 5-level experiment to obtain an approximation of the relationship between CoG offset and the maximum tractor speed, a boundary between viable and possibly infeasible designs. We may then apply our engineering intuition and hypothesise that the optimal boundary between viable and infeasible designs is to the right (increased speed) of where the 5-level results show it to be. Based on this we divide the graph into two areas, a viable area where we have some confidence that all designs will work and an area where further exploration is warranted. These areas can be seen in Fig. 10.12.

If we then explored the identified region at the same resolution as the 21-level experiment, including the borders, this would require an additional 231 simulations. This would result in obtaining the same speed against CoG offset profile as in the 21-level experiment but at a cost of only 256 experiments, under 12 % of the complete experiment simulations.

## 10.8 Ranking of Results

Simulation experiments can produce large amounts of data both from the number of experiments performed and also the number of properties recorded during the simulation. In these cases, automated support for ranking of results can help reduce designer workloads. The first step towards defining this automated support is to understand what properties are significant for the stakeholder goals and from where in the model these properties can be derived. Then ensure that these properties are included in the simulation output.

At the same time as including results that permit the ranking of different designs, the simulation results should ideally allow the automatic rejection of a design that fails in some way. This additional data should relate to any invariants or safety requirements that a design should not violate.<sup>4</sup> Alongside this data it will

<sup>4</sup>When possible to express these directly in VDM, that is recommended, but in some cases, this is not natural.

be necessary to precisely define what it means to violate any of these conditions, such as what is the maximum current a motor can accept or what is the maximum force we can apply to a human user.

For automated ranking, there must be some form of *ranking function*. This function should be able to place the designs in at least a partially ordered set based upon their simulation results. If these results are being compared using only a single criterion, then the ranking function is trivial to describe, for example, ordering the results based upon their maximum speed. If the results are being compared on multiple criteria, then the function may be more complex. Three forms of ranking functions follow in the subsections below.

### 10.8.1 Simple Equation

The simplest form of ranking function is just an equation. Such an equation could be used to determine the ratio between two results or perhaps the statistical mean of a series of results. The numerical output from the equation then allows the results to be ordered.

### 10.8.2 Weighted Additive Method

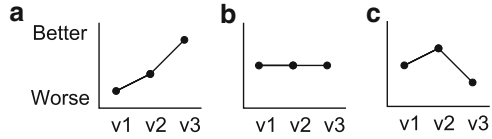
The Weighted Additive Method[9] (WAM) is based on equations that explicitly highlight the relative weight given to each measured result. In this method, each of the  $n$  metrics,  $x$ , are evaluated using a function  $v$  and then multiplied by a weighting  $w$ . The scores for each metric are summed together to give a final score  $V$  for that design. The equation for a design  $a$  is shown in Eq. (10.1)

$$V_a = w_1^a v_1^a(x_1^a) + w_2^a v_2^a(x_2^a) + \dots + w_n^a v_n^a(x_n^a) \quad (10.1)$$

### 10.8.3 Enumeration and Scoring

Enumeration and scoring [9] (ENUM) is based upon defining logical predicates, each of which describes some desirable condition in the results. These predicates may consider only a single metric from the simulation results, for example, the object travelled more than 5 m, or they may consider multiple metrics, for example, the object travelled more than 5 m and used less than 40 J of energy. These predicates are then placed into sets, where all predicates  $g_1 - g_n$  in a set are considered to be equally desirable. With the sets of predicates constructed, each predicate is assigned a score. The score for a particular design is the sum of the scores for all predicates

**Fig. 10.13** Example outputs showing possible trends for a parameter with three values



satisfied by the simulation results. The scores for each predicate are such that the score for a single predicate in set  $S_n$  outweighs the total scores for all predicates in all rows below that row. The general equation for this method is shown in Eq. (10.2)

$$\begin{aligned}
 &order_a = \\
 &\{ \\
 &\quad S_1^a = \{g_1^1 : g_2^1 : \dots g_n^1\}; score_{S_1^a} = w; \\
 &\quad S_2^a = \{g_1^2 : g_2^2 : \dots g_n^2\}; score_{S_2^a} = y; \\
 &\quad S \dots \\
 &\quad S_n^a = \{g_1^n : g_2^n : \dots g_n^n\}; score_{S_n^a} = z; \\
 &\}
 \end{aligned}
 \tag{10.2}$$

### 10.8.4 Analysis of Results

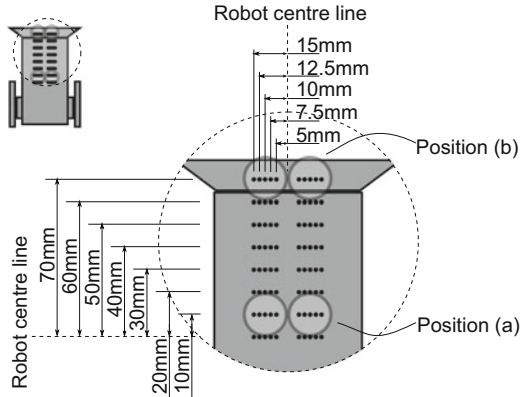
The results returned from any fractional factorial experiment will only cover a part of the design space. It is therefore possible that the best results returned are not in fact the best results possible within the scopes defined for the parameters. One way to approach this problem is to consider the effect of each parameter value on the results obtained individually, such as in sensitivity analysis. Figure 10.13 shows some example graphs that could be obtained if the mean simulation ranking for a parameter value is plotted against that parameter value. In graph a, there is a general move towards better values as the parameter changes from v1 to v3, in graph b the parameter value is found to have little or no effect and in c there is a peak at v2 and dips either side. If these graphs are produced for all parameters, then a suggestion for a potentially good design can be obtained by finding the values that give the maximum result for each parameter; such a prediction can be validated by performing a simulation run with exactly those parameters. The graph results may also be used to indicate a number of potentially valuable designs worthy of further investigation, and so these predictions form the focus for an additional series of simulations.

## 10.9 An Example Using the Line-Measuring Robot

In Sect. 10.3, we saw the effects of making dramatic changes to the sensor positions on the speed and accuracy with which the robot was able to follow a line. In this second visit to the line-following robot, we see it used in its line-measuring mode



**Fig. 10.14** Graphical positions of the sensors overlaid on a plan view of the robot. *Positions (a) and (b)* refer to the earlier ACA experiment in Sect. 10.3



**Table 10.7** Sensor offsets: letters indicate longitudinal offset and numbers indicate lateral offset

Lateral offset (m)	Longitudinal offset (m)							
	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07
0.0050	(a1)	(b1)	(c1)	(d1)	(e1)	(f1)	(g1)	(h1)
0.0075	(a2)	(b2)	(c2)	(d2)	(e2)	(f2)	(g2)	(h2)
0.0100	(a3)	(b3)	(c3)	(d3)	(e3)	(f3)	(g3)	(h3)
0.0125	(a4)	(b4)	(c4)	(d4)	(e4)	(f4)	(g4)	(h4)
0.0150	(a5)	(b5)	(c5)	(d5)	(e5)	(f5)	(g5)	(h5)

to demonstrate the use of the multi-attribute ranking functions, WAM and ENUM, described in Sect. 10.8.

If we assume that a design that is considered good for line following is likely to be good for measuring a line as well. With this in mind we will constrain the positions of the sensors to a region indicated to be “good” by the previous experiment. We will also constrain the search to include positions within the boundaries of the robot itself. The top four designs from the previous experiment were (b), (f), (a) and (c). Of those designs only (b) and (a) are actually within the boundaries of the robot and so we will use those two to guide the parameters for this exploration. Figure 10.14 shows the locations of sensor positions (a) and (b) from the previous experiment, overlaid with dots representing the positions that are explored in this experiment, and Table 10.7 gives names for each of the sensor positions.

The co-model of the line-measuring robot is instrumented to allow assessment of three potentially antagonistic properties, line-measuring accuracy, mean speed at which the line is measured and the energy consumed per meter of measured line. These three properties can then be used to rank the designs to determine which one performs the best. In the earlier experiment, each design was assigned a ranking for each of the four line-following measures identified and then an overall score for each design was found by taking the mean of each measure’s rank. This assumes that all of the measures are of equal importance to the stakeholders and ignores relative values of each measure. Below we present two instances of possible ranking functions that can be applied to this line-measuring version of the robot that do not suffer from these weaknesses.

$$V_a = 2\left(1 - \frac{\text{error}^a - \text{MinError}}{\text{MaxError} - \text{MinError}}\right) + \frac{\text{speed}^a - \text{MinSpeed}}{\text{MaxSpeed} - \text{MinSpeed}} + \left(1 - \frac{\text{energy}^a - \text{MinEnergy}}{\text{MaxEnergy} - \text{MinEnergy}}\right)$$

**Fig. 10.15** Weighted Additive Method ranking function for the line-measuring robot

**Table 10.8** Enumeration and scoring ranking function predicates for the line-measuring robot

Set	Predicates	Score per predicate	
$S_1$	$\text{error}^a < 0.7$	$\text{speed}^a > 0.075 \wedge \text{energy}^a < 17$	48
	$\text{speed}^a > 0.08$		
	$\text{energy}^a < 16$		
$S_2$	$\text{error}^a < 1 \wedge \text{speed}^a > 0.075$	$\text{speed}^a > 0.07 \wedge \text{energy}^a < 18.5$	7
	$\text{error}^a < 1 \wedge \text{energy}^a < 17$		
	$\text{error}^a < 3 \wedge \text{speed}^a > 0.07$		
$S_3$	$\text{error}^a < 3 \wedge \text{energy}^a < 18.5$	$\text{speed}^a > 0.055 \wedge \text{energy}^a < 20$	1
	$\text{error}^a < 5$		
	$\text{speed}^a > 0.06$		
	$\text{error}^a < 6 \wedge \text{speed}^a > 0.055$		
	$\text{error}^a < 19.5$	$\text{error}^a < 6 \wedge \text{energy}^a < 20$	

The first function uses the WAM. In this implementation, each of the three measures is first normalised against the complete set of simulation results to be on a scale of zero to one, where zero implies a result equal to the worst result, a one represents a result equal to the best and varying linearly in between. This represents the  $v_n^a(x_n^a)$  term in Fig. 10.1 for measure  $n$  of design  $a$ . The second part of the method is to multiply each of these terms by its weighting and sum the resulting terms to obtain a score for each design. In the case of a line-measuring robot, we can imagine that the accuracy of line-measurement might be of greater importance than the speed of measurement or the energy used and so the accuracy term is given a weighting of two while the other two terms have a weighting of one. The resulting equation is shown in Fig. 10.15. The *MaxError*, *MaxSpeed*, *MaxEnergy*, *MinError*, *MinSpeed* and *MinEnergy* refer to the maximum and minimum for each metric across all designs tested.

The second ranking function uses the ENUM method. In this implementation (Table 10.8), there are three sets of predicates,  $S_1$ ,  $S_2$  and  $S_3$ , with each of these representing progressively less desirable conditions for the robot design to satisfy. In each set there are three predicates containing only one metric and three predicates containing combinations of these metrics. The single metric terms represent a more narrow view of robot performance than the twin metric terms and so, to be similarly desirable, impose a higher requirement on performance. For example in  $S_1$ , it could be desirable that the robot measures the line at a speed of greater than 0.08 m/s, but it could be equally desirable for the speed to be greater than 0.075 m/s but if the energy consumed is less than 17 J/m. The ‘‘Score per predicate’’ column shows the score obtained for each predicate satisfied; these are such that, for example, satisfying even a single predicate in  $S_2$  outweighs satisfying all predicates in  $S_3$ .

The simulation results are presented in Table 10.9. Five of the designs failed to complete the line-measuring task and these designs are indicated within the table. For those designs that did complete the task, their performance is indicated in

**Table 10.9** The scores and resultant ranking from the robot metrics and both ranking functions

Design	Speed		Error		Energy		WAM		ENUM	
	Score	Rank	Score	Rank	Score	Rank	Score	Rank	Score	Rank
(a1)	Failed to follow line									
(a2)	Failed to follow line									
(a3)	0.0722	28	4.94	19	19.24	35	1.48	34	13	32
(a4)	0.0654	35	6.40	23	19.12	33	0.97	35	3	35
(a5)	Failed to follow line									
(b1)	0.0759	20	3.74	16	19.16	34	1.88	27	13	33
(b2)	0.0755	23	3.61	15	19.10	32	1.90	26	13	34
(b3)	0.0751	24	2.28	10	18.80	30	2.20	17	20	30
(b4)	0.0683	33	1.08	5	18.05	24	2.34	11	34	23
(b5)	0.0682	34	1.21	6	18.08	25	2.30	15	34	24
(c1)	0.0776	16	2.14	9	18.88	31	2.31	14	20	31
(c2)	0.0786	13	0.81	4	18.64	29	2.66	4	75	11
(c3)	0.0759	21	0.38	<b>1</b>	18.21	27	2.74	<b>2</b>	137	<b>1</b>
(c4)	0.0702	31	0.78	<b>3</b>	17.70	21	2.57	6	48	20
(c5)	0.0701	32	1.45	7	17.59	19	2.48	9	48	21
(d1)	0.0800	10	0.65	<b>2</b>	18.37	28	2.82	<b>1</b>	137	<b>2</b>
(d2)	0.0810	9	1.98	8	18.11	26	2.70	<b>3</b>	82	10
(d3)	0.0768	18	2.64	11	17.77	22	2.50	8	41	22
(d4)	0.0720	29	3.31	12	17.21	16	2.34	12	27	25
(d5)	0.0717	30	3.58	14	17.18	15	2.29	16	27	26
(e1)	0.0828	5	3.44	13	17.87	23	2.59	5	75	12
(e2)	0.0823	7	3.84	17	17.70	20	2.55	7	75	13
(e3)	0.0784	15	4.77	18	17.32	18	2.32	13	27	27
(e4)	0.0735	26	5.17	20	16.92	14	2.15	18	26	28
(e5)	0.0732	27	5.70	21	16.80	11	2.08	21	26	29
(f1)	Failed to follow line									
(f2)	0.0852	4	6.10	22	17.32	17	2.37	10	72	14
(f3)	0.0794	11	6.77	24	16.92	13	2.12	19	72	15
(f4)	0.0755	22	7.43	25	16.52	8	1.95	23	72	16
(f5)	0.0751	25	7.70	26	16.46	7	1.91	25	72	17
(g1)	Failed to follow line									
(g2)	0.0860	<b>3</b>	8.63	27	16.83	12	2.10	20	120	<b>3</b>
(g3)	0.0811	8	8.90	28	16.55	9	1.92	24	120	4
(g4)	0.0776	17	9.43	29	16.21	5	1.78	29	72	18
(g5)	0.0765	19	9.83	30	16.07	<b>3</b>	1.70	30	72	19
(h1)	0.0889	<b>1</b>	9.96	31	16.65	10	2.04	22	120	5
(h2)	0.0872	<b>2</b>	11.02	32	16.33	6	1.87	28	120	6
(h3)	0.0828	6	11.29	33	16.12	4	1.69	31	120	7
(h4)	0.0791	12	11.42	34	15.84	<b>2</b>	1.59	32	120	8
(h5)	0.0786	14	11.69	35	15.71	<b>1</b>	1.56	33	120	9

Top three results according to each ranking highlighted in bold

five different ways. The first six columns present the scores and resultant rankings within the group for the three metrics, speed of line measurement, accuracy of measurement and energy consumed per metre of line. It is not surprising to find that the rankings are different for each of the measures, for example, the fastest design, (h1), ranked 31st out of 35 for accuracy while the most accurate design (c3) ranked 27th for energy consumed. The final four columns contain the scores and rankings according to the two ranking functions. Both functions agree, although transposed, on the top two designs, (c3) and (d1), they also agree on the worst design (a4), though in between these the orderings are quite different. This highlights that ranking functions should be agreed by the stakeholders of a project or against the project requirements.

## 10.10 Conclusion

Co-simulation 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. In this chapter, we presented an approach to such design space exploration using experiment design. We started with a classification for the parameters that one may wish to vary during an experiment. We discussed methods for reducing the number of experiments performed, from screening experiments to find parameters that have little effect to the use of orthogonal matrices or a space-filling search to find actual parameter values to use in simulations. This was done using two different case studies which showed that it matters how well experiments are designed. The chapter concluded with a discussion of the ways in which experimental results may be ranked to permit automatic selection of the best designs simulated. The reader is invited to further explore the examples introduced in this chapter by defining additional experiments; all the relevant co-models can be imported to the Crescendo tool.

# Chapter 11

## Industrial Application of Co-modelling and Co-simulation Technology

Marcel Verhoef and Peter Gorm Larsen

### 11.1 Introduction

This chapter provides an overview of three industrial applications that have been carried out using the Crescendo co-modelling and co-simulation technology. The models have been developed by industry users and this presentation also includes a summary of the achievements and main lessons learned from the work performed in an industrial context. The applications include

- a dredging excavator system,
- a document handling system and
- a self-balancing scooter, the “ChessWay”, as introduced in Chap. 7.

The applications reported in this chapter have been produced by three different companies as case studies for the DESTTECS project [28] under which the Crescendo technology was originally developed. First, we provide a short overview of the companies involved:

**Verhaert New Products & Services N.V.** Verhaert is an integrated product development center delivering innovation, development and engineering services for state-of-the-art high-technology products. Verhaert is based in Kruikebeke, Belgium, and is active in many technology development programmes leading to innovative hardware, software and devices. For more information, we refer to <http://www.verhaert.com>.

---

M. Verhoef (✉)  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

P.G. Larsen  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

**Table 11.1** Case study context

Application	Self-balancing scooter	Document inserting system	Dredging excavator
Company	CHESSE	NEOPOST	VERHAERT
Disciplines	Mechanical, control, electrical, software	Mechanical, control, electrical, software	Mechanical, control, electrical, software
Fault source	Design faults	Error handling	Operator error
Challenge	Manage complexity	Concurrent design	Product robustness
Improvement	Reliability analysis	MIL simulations	Design exploration
Purpose	Trustworthy design	Raise product quality	Reduce product TTM
Approach	DE-first, CT-first and contract-first	DE-first	CT-first
Prior knowledge	VDM, 20-sim	20-sim	–

*MIL* model in the loop, *TTM* time to market

**Neopost Technologies B.V.** Neopost is one of the two world leaders in small- to middle-range mail handling equipment manufacturing. Its portfolio includes machines for flexible packing of multiformat documents, automated handling of incoming high volume mail and supporting information systems. Neopost Technologies B.V. is based in Drachten, the Netherlands, with a 75-head R&D force, which forms the “Document Systems” Competence Centre of the Neopost Group. For more information, see <http://www.neopost-technologies.com>.

**Chess WISE B.V. and Chess iX B.V.** The Chess group forms a design and development centre with core competences in both hardware and software. They also take operational responsibility and perform life-cycle maintenance of the electronic products and systems that they develop. Chess is based in Haarlem, the Netherlands, with a multidisciplinary engineering team of approximately 50 engineers. See <http://www.chess.eu> and <http://www.chess-ix.com>.

The applications described in this chapter are very different in nature and they use the alternative approaches introduced in Chap. 8. The case studies have been carefully selected to provide a range of embedded systems applications with different forms of complexity, involving engineering heterogeneity (so that collaborative approaches are of interest) and all having the need to provide a predictable level of fault tolerance. They were chosen to represent a state-of-the-art innovative design problem, but they also intended to be recognisable and acceptable to the industry at large, in order to ensure impact. They were proposed and taken on by individual partners in order to reduce risks and optimise resourcing but also to expose different working practices. Thus, they complement each other and illustrate the variety of situations in which the Crescendo technology is applicable. An overview is provided in Table 11.1.

Unfortunately, we are not able to present the full co-models from the case studies for commercial reasons. However, we are able to provide an overview of the approaches taken, to a certain extent the structure of the co-models and the main findings from the case studies. This chapter first introduces the dredging excavator

case study from Verhaert in Sect. 11.2. The Neopost document handling system is presented in Sect. 11.3. For confidentiality reasons, the details of the latter model are replaced by a paper path co-simulation model from the BODERC project [96] that has similar characteristics. Afterwards, Sect. 11.4 introduces the ChessWay application. Finally, Sect. 11.5 provides a summary of the chapter.

## 11.2 A Dredging Excavator

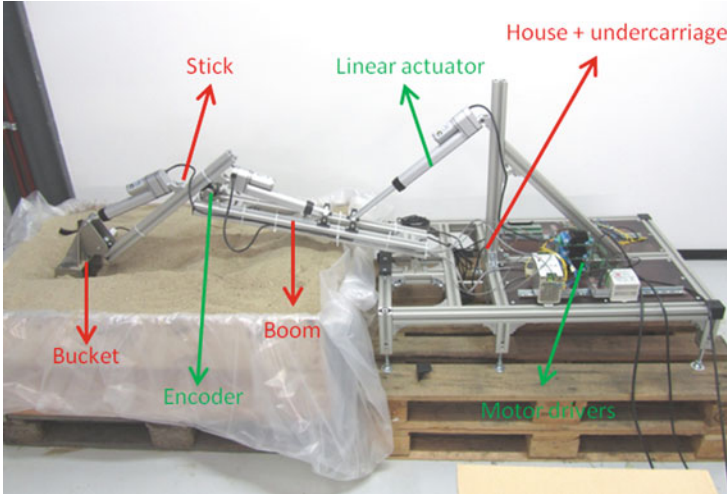
### 11.2.1 Case Description and Main Challenges

The Verhaert case study concerns the design of an excavator-based dredging system. The dredging process can be defined as the repositioning of soil from the seafloor for infrastructural or ecological purposes. This process typically involves expensive machinery and complex operator control as well as several days of operation, resulting in expensive and difficult processes. A dredging system is a complex application that involves a thorough logistical planning of the dredging process itself, besides the associated control problem of the elements involved in it.

As a general goal, it is Verhaert's intention to elevate the performance of dredging systems by introducing semi-automated control to improve productivity and also to reduce the down time due to repairs. At a lower (loop control) level, this includes a dynamic response function of the digging resistance to avoid overload of the machine. At a higher (sequence control) level, automation should provide predefined trajectory control (e.g. digging along a straight path) and optimisation of the digging pattern. The ultimate goal of this automation is to design an excavator which can operate completely autonomously.

Currently, the expertise of the operator has a large impact on the performance of the machine. In particular, the resistance encountered by the bucket during its path is crucial. For example, a novice operator tends to apply too much force with the arm, while an experienced operator is able to operate the excavator smoothly. By creating a system that assists the user, optimal operation can always be achieved. The goal of this case study is to investigate an automated control system as described above in order to increase the performance and the uptime of the excavator.

Verhaert based their work on a realistic excavator scale model with three axes of freedom (the *shoulder*, connecting the undercarriage with the boom, the *elbow*, connecting the boom with the stick and finally the *wrist*, connecting the stick and the bucket), which allowed for easy model validation as the setup can be used for realistic tests and measurements under controlled conditions. The scale model, which is shown in Fig. 11.1, is powered by electrical linear actuators instead of hydraulics, which are common in practice. The continuous time model of the excavator case study is introduced in Sect. 11.2.2 and the discrete event model is presented in Sect. 11.2.3. Afterwards Sect. 11.2.4 presents the co-simulation



**Fig. 11.1** The Verhaert excavator scale model and test setup

analysis conducted on this co-model and finally Sect. 11.2.5 provides an overview of the key results of this study.

### 11.2.2 *The Continuous Time Model*

A top-level overview of the CT model of the excavator is presented in Fig. 11.2. This figure uses the block diagram notation but represents a hierarchy of differential equations, as illustrated by the model snippet in Fig. 11.3. The electrical linear actuators are managed using encoders that send motor axle rotation measurements over to the controller. Based on the requested operator inputs, the controller determines the Pulse Width Modulation (PWM) signals to be sent to the power amplifiers driving the motors inside the linear actuators that make the excavator parts (boom, stick and bucket) move.

A special ground model was introduced to investigate the impact of ground material properties on the automated control, by comparing simulations with different ground material properties (e.g. mass density, drag). The ground model describes the forces experienced by the bucket when digging through the ground. Two different forces have been modelled: a constant cut force (velocity independent) that is required to break open the ground and a drag force (velocity-dependent) due to the bucket moving through the ground. The ground consists of several layers (along the  $z$ -direction) of different composition. The parameters of these layers (height, cut and drag force, mass density) can be set by reading in a table from a text file. It is also possible to activate an obstacle at a certain position in the  $x$ -direction.



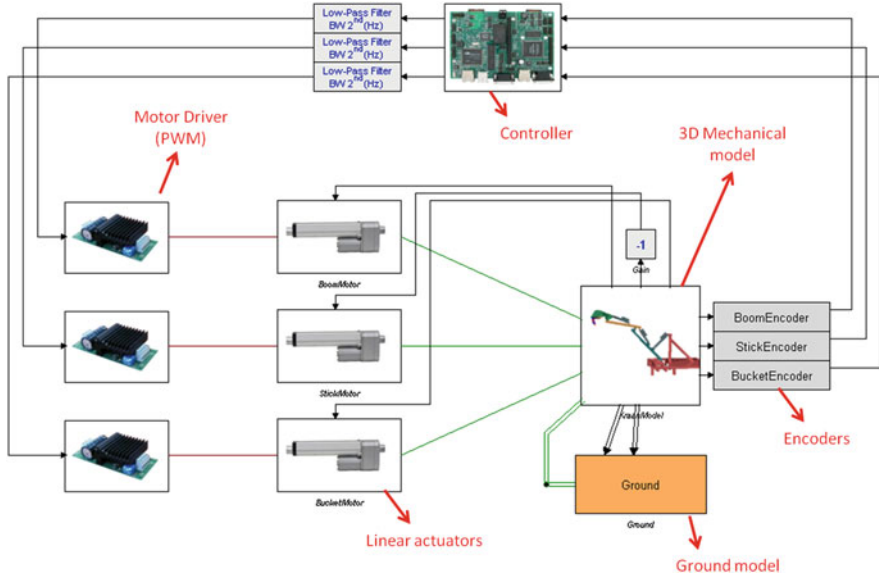


Fig. 11.2 The top-level continuous time model of the excavator

```

equations
// Motor equations
Tm = K * iMotor;
p1.u = K * omega + R * iMotor + 0.0000000001;

p1.i = iMotor + minCurrent * sign(p1.u);

// spindle speed relation
p2.v = omega * i;

// spindle force/torque relation
Tm = Tmax * Fspindel^2 / ( Fspindel^2 + Fx^2 );

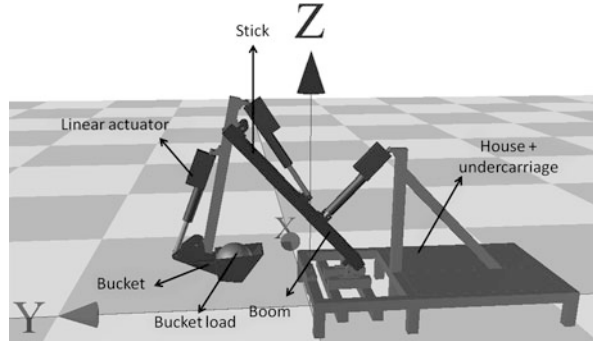
Fspindel = p2.F + col;

load = Tm / i;
overload = (not collision and abs(load) > maxLoad);
    
```

Fig. 11.3 Part of the 20-sim submodel for a linear actuator

This obstacle has a different composition than the rest of the layer. This feature can be used to model discontinuities, for example, a hard rock in the otherwise soft soil, at which point the excavator will experience a possible overload when trying to dig through it. The ground model also calculates the volume and mass of the ground that is dug up. This mass is then fed back into the mechanical model, to represent the varying mass and inertia of the bucket load while digging. In case of the dredging excavator, buoyancy effects can be added if the bucket is still under water.

**Fig. 11.4** The 3D mechanical model of the excavator



Visualisation techniques can be used in order to get a better understanding of the physical behaviour of the excavator. 20-sim provides a 3D mechanical editor that creates a 3D model with all its associated static properties (such as centre of gravity, inertia and so on). This 3D model is automatically and directly coupled to the differential equations describing the dynamical behaviour of the system. Animation is basically a time-lapse recording of the evolution of those model variables, and this is visualised directly in the 3D model. A screen dump of the excavator animation is shown in Fig. 11.4.

### 11.2.3 The Discrete Event Model

An overview of the structure of the VDM model of the excavator is shown in Fig. 11.5. The model consists primarily of three VDM classes, each allocated to their own CPU, to denote their independence. All three have a separate control loop running periodically, albeit at different intervals. The *operator* mimics the behaviour of the human operator in terms of using the controls at his disposal: the *joystick* and some *buttons*. The *controller* is the core of the model implementing the feedback control strategy, by reading from *sensors* and writing to *actuators*. The *safety unit* is concerned with independent assessment of the system safety state. It also observes the *sensors* and in case it detects an anomaly, it will shutdown the *controller*.

The main VDM classes *operator*, *controller* and *safety unit* are explained in more detail in the following sections. Also, the key operating modes of the excavator are described, as they are essential to understand the dependability analysis conducted in Sect. 11.2.4.

#### 11.2.3.1 Operator

The *Operator* class is the first of the three principal classes. Although in a physical sense the operator is not part of the controller hardware, the creation of this class allows us to model the interaction between the human operator and the

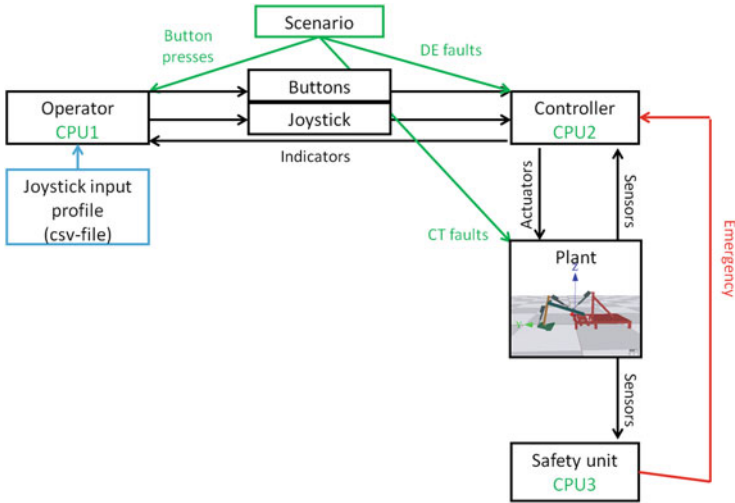


Fig. 11.5 A context diagram of the VDM model of the excavator

system. The operator receives input from the controller by means of indicators (lights and gauges on the dashboard), as well as visual feedback (the positions of the components of the excavator). Based on this feedback, he can manually operate the excavator using buttons and joysticks. The joystick handling at a given time is imported by reading a table from a comma separated text file using the CSV standard library.

**11.2.3.2 Controller**

The second principal class of the VDM model is the Controller class. The controller has a total of ten inputs: four buttons (power on/off, emergency on/off, start/stop and manual/assisted); three joysticks (one for each of the three Degrees of Freedom (DoF): boom, stick and bucket); and three sensor values (the angles between each of these components, measured with a relative encoder). Based on a fixed time step (typically 100 Hz), the controller reads these input values, processes them and provides three output signals to the actuators (again, one for each DoF). Using the co-simulation engine, the controller exchanges the sensor and actuator signals with the CT model of the excavator. The relationships from the Controller class are shown in a class diagram in Fig. 11.6.

**11.2.3.3 Safety Unit**

The third and final principal class is the SafetyUnit. This class has a thread that runs in parallel with the controller and also reads and processes the sensor values (the sensor signals are split in the software and then sent to both the controller

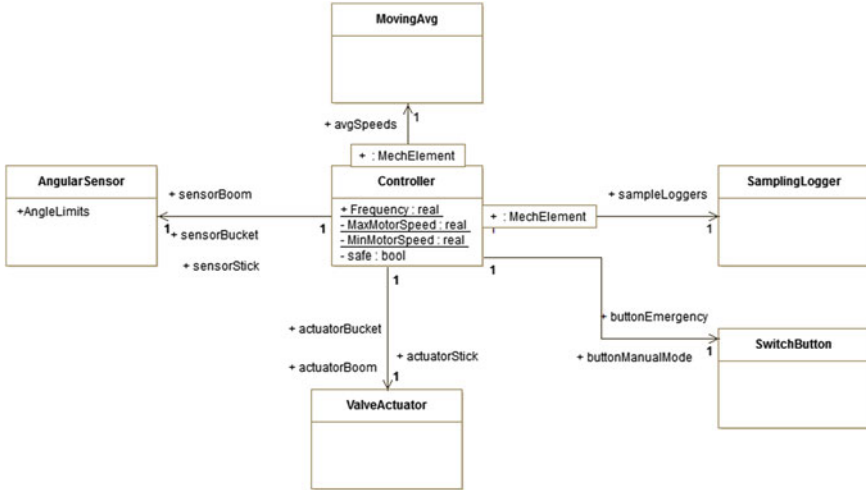


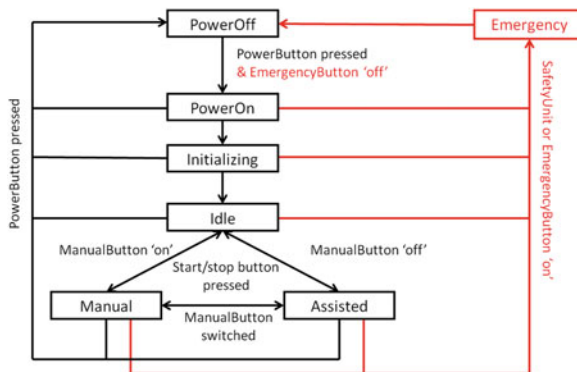
Fig. 11.6 An overview of the relationships from the Controller class

and the safety unit). The safety unit is a redundant system that under normal circumstances does not need to take any actions. It acts as a safety mechanism that only comes into action when the controller fails. There can be different reasons for such failures (e.g. a software bug, an unforeseen situation or a hardware failure), but this is irrelevant from the point of view of the safety unit.

One specific function of the safety unit is to prevent the actuators from crashing into their end stops, potentially resulting in breakdown of the entire excavator. Even though the controller contains the intelligence to slow down the motion to a soft stop before running into the end stop, the safety unit always makes an additional check to ensure that a crash does not occur. Since the safety unit only makes simple checks, it requires less processing power and thus runs on a lower-performance processor, but at a higher speed.

When action is required, the safety unit intervenes in several ways simultaneously. Firstly, it triggers the emergency state of the controller, resulting in a power shutdown of the excavator, thereby stopping all motion. This emergency state will only be reached when the controller is still running. An example of this is a controller failure due to an erroneous calculation of the limits. Secondly, the safety unit overrules the output of the controller and thereby stops all motion. Lastly, the safety unit opens three relays, by which power to all actuators is cut off. The system can be restarted by following an explicit reset procedure where the controller has only limited functionality at its disposal until the safe operating zone is reached again.

**Fig. 11.7** The different states of the controller class



### 11.2.3.4 Controller States

The controller class has seven different states, described by the informal state diagram in Fig. 11.7. The initial state is “PowerOff”, in which the excavator is unpowered. By pressing the “PowerButton”, the operator turns on power and the controller goes to the “PowerOn” state and then to the “Initialising” state. This state can be used to perform a homing procedure in which the actuators are moved to their starting position. This position is then used as a reference such that the read out of the relative encoders can be used to measure absolute angles between the components. When the initialisation is done, the controller turns to the “Idle” state. In the “Idle” state, the excavator is powered and the controller is running. By pressing the “Start/Stop button”, the controller moves either to the “Manual” or the “Assisted” state, depending on whether the “ManualButton” is switched on or off. When the controller is in either of these two states, switching the “ManualButton” turns the controller from the “Manual” to the “Assisted” state or vice versa. By pressing the “Start/Stop button” again, the controller moves back to the “Idle” state. When the operator pushes the “PowerButton” again, power to the excavator is shut down and the controller returns to the “PowerOff” state.

Besides these normal states, the controller also has an “Emergency” state. This state can either be triggered by the operator by turning on the “EmergencyButton”, or by the “SafetyUnit” when a dangerous situation is detected. When the “Emergency” state is triggered, power is shut down and the controller moves to the “PowerOff” state. As long as the “EmergencyButton” is on, power cannot be restored and so the controller remains in the “PowerOff” state.

### 11.2.3.5 Modes of Operation

In normal operation, the controller runs in either “Manual” or “Assisted” mode. These are the only two modes in which output signals can be sent to the actuators to move the excavator. To prevent overload of the actuators by sudden changes in input

```

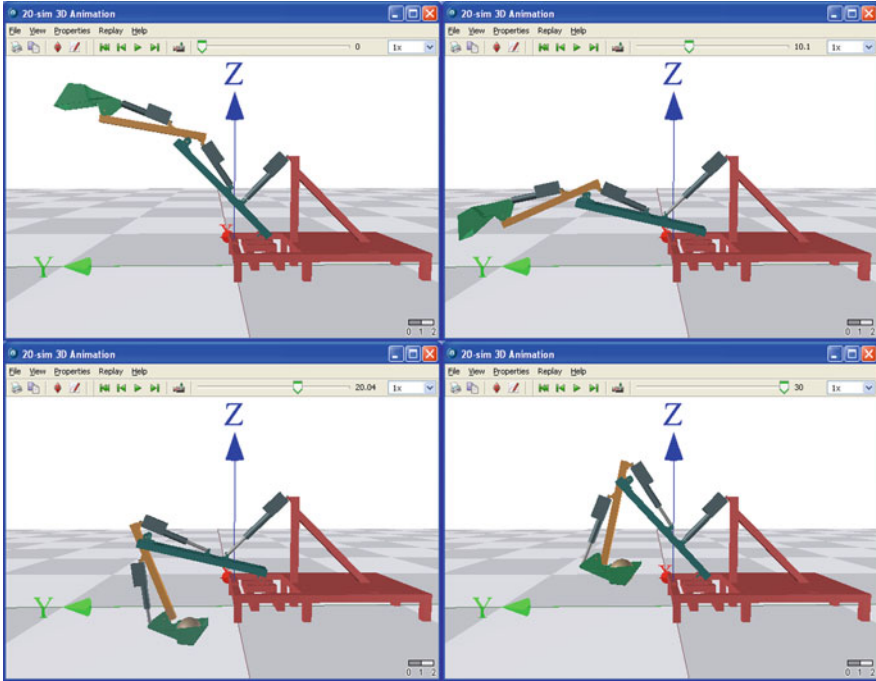
speedPercentageAtAngle: MechElement * real * real ==> real
speedPercentageAtAngle(element, angle, speed) ==
(...
  let softLimit = sensors(element).getSoftLimit(useMinAngleLimits),
      hardLimit = sensors(element).getHardLimit(useMinAngleLimits)
  in
  if slowdownLimiter = 1.0
  then ratio := 1.0 - (angle - softLimit) / (hardLimit - softLimit);
  return if hardLimit > softLimit
    then if angle > hardLimit
      then 0.0
      elseif angle > softLimit
        then ratio
        else 1.0
    elseif hardLimit < softLimit
      then if angle < hardLimit
        then 0.0
        elseif angle < softLimit
          then ratio
          else 1.0)

```

**Fig. 11.8** A VDM model snippet of the motion limiter

by the operator, the controller smoothes out the input signals from the joysticks. This is done by defining a slope, which limits the maximum rate by which the signal can vary over time. Since the actuators have a limited range, we must also prevent them crashing into their end stops. Therefore, the controller reads in the encoder signals which are then used to determine the angles between the components. By setting a limit to these angles, we can limit the movement of the actuators and prevent these crashes. For both endpoints (beginning and end) three limits are defined. Two of these, the soft and hard limit, are used by the controller. As soon as an actuator passes the soft limit, the controller limits the output signal to this actuator. When the actuator passes the hard limit, the signal is turned to zero. In between both limits, the actuator signal is limited by multiplying the signal with a factor that scales linear between 0 (hard limit) and 1 (soft limit). A VDM model snippet of this function is shown in Fig. 11.8. This limiting is only activated when the actuator is moving towards the end stop. When it is moving away from it, its motion is not limited. The third limit that is set for this range limiting lies behind the hard limit. It is used by the safety unit and when the actuator passes this limit the “Emergency” state is triggered and power to the excavator is shut down. This safety mechanism is a last resort that only kicks in when the controller fails.

In “Manual” mode, the operator drives each of the actuators directly by controlling the rotation speed between the components. In “Assisted” mode, the operator drives the movement of the bucket (the translation in the  $x$  and  $z$  direction and the angle between the bucket and the ground), while the controller translates the requested motion into the individual rotation speeds of the components. It uses a reverse kinematics calculation based on the measured angles between boom, stick and bucket to generate setpoints for the linear actuators dynamically such that perfectly straight and smooth movements can be made.



**Fig. 11.9** The standard digging motion used in some of the simulations (*from top-left, top-right, bottom-left to bottom-right*)

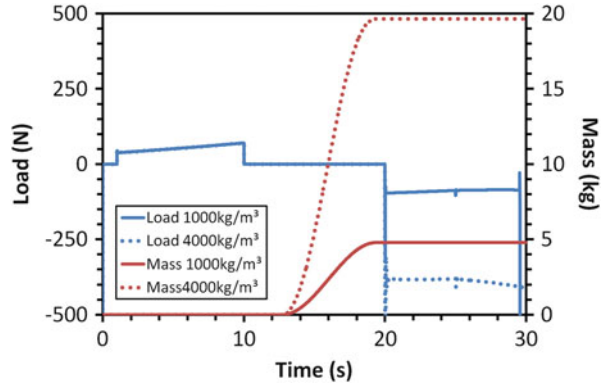
### 11.2.4 Co-simulation Analysis

Using the co-model, different co-simulations have been executed investigating several aspects of the behaviour of the excavator. This information was then used to improve the functionality of the controller and to fine-tune the parameters of the control algorithm. The most interesting results of these co-simulations are presented in this section.

#### 11.2.4.1 Ground Model

In a first experiment, a standard digging motion was performed (see Fig. 11.9) and the mass of the ground load in the bucket was taken into consideration, as well as the load on the actuators. This experiment was repeated for different values of the mass density of the ground, to validate the impact of the ground model on the performance of the system. Results of these simulations are shown in Fig. 11.10. Considering the load of the boom actuator, it is possible that first there is a positive load (bucket goes down), then there is no load (bucket moves towards the actuator) and then there is a

**Fig. 11.10** The mass of the load in the bucket and the load on the boom actuator



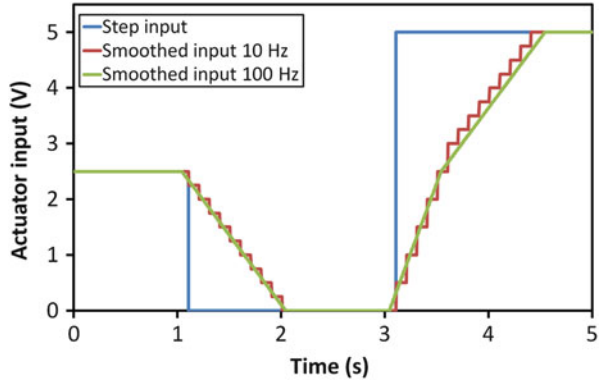
negative load (bucket goes up). Looking at the graph of the mass, we see that after 13 s, the bucket hits the ground and starts digging, thereby accumulating mass in the bucket. The mass keeps increasing, until the bucket is retrieved from the ground, at 20 s. We can clearly see that the mass in the bucket scales with the mass density of the ground. When the mass density in the bucket is higher, this has a significant effect on the performance of the excavator. This effect only appears when the bucket is pulled up again, thus after it has been filled, which demonstrates that the ground model works appropriately.

#### 11.2.4.2 Overload

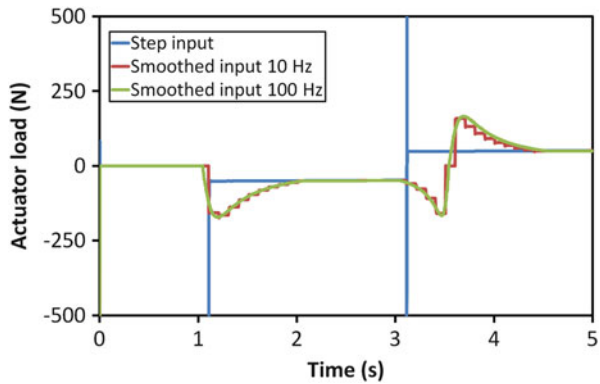
One of the aims of this case study is to design a controller that assists the operator so that overloads of the system are avoided. In the controller model, two types of overload protection are included. The first type is protection against abrupt changes to the input of the actuators, resulting in high accelerations and thus loads. An example of this is the situation where the operator pulls the joystick rapidly from one end to the other, resulting in an abrupt switching in the direction of the actuator's motion (i.e. from full speed backward to full speed forward). This abrupt change in input and the resulting load on the actuator are shown in Figs. 11.11 and 11.12. Due to inertia, this results in a very high load on the actuator (the maximum load on the actuator in the test setup is specified at 500 N), enough to cause severe damage to it. To prevent this overload, the controller needs to smooth the joystick input before passing it to the actuator. This smoothing is implemented by a ramp function, defined by a maximum slope (per increase in input voltage per second). A different slope can be set for the forward and backward movement. The figures show the result for an input smoothing with the controller running at 10 Hz and at 100 Hz. Using this smoothing function, we see that the load on the actuator becomes much smaller, thus preventing an overload.



**Fig. 11.11** The input signal to the actuator, as function of time. The step input represents the abrupt signal that is received without an intervention of the controller, as opposed to the signals that are smoothed by the controller



**Fig. 11.12** The load on the actuator, for a step and a smoothed input



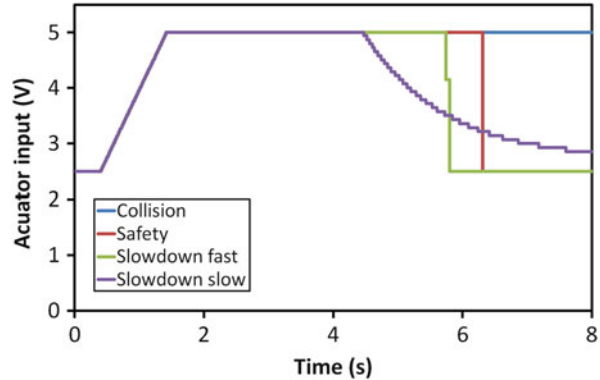
**11.2.4.3 Endstop Protection**

The second protection mechanism prevents the actuator from actually crashing into its physical end stops. In order to achieve this, two additional protection mechanisms have been included, one in the controller and one in the safety unit. The input signal and the resulting actuator load are shown in Fig. 11.13. Without any protection mechanism, the input signal to the actuator remains at full speed and the actuator crashes into the end stop, resulting in a very high load.

The protection mechanism of the safety unit terminates motion immediately when such a crash is imminent. Although this mechanism prevents a crash with the end stops, it still does this in an abrupt way which also introduces a high load on the actuator. However, this is acceptable as the hard stop will not be used in everyday operation but only in case of emergency, that is, when the controller fails. In that case, the emergency stop is more important than potential damage to the system, for example a person might be in harms way, which requires immediate action leading to a full stop as soon as possible.

The mechanism modelled in the controller also terminates the motion, but it does this in a smooth way, by slowing down the motion between a soft angular limit and

**Fig. 11.13** The input signal to the actuator as function of time. The collision signal corresponds to the simulation without any protection, safety refers to the protection mechanism implemented in the safety unit and slowdown refers to the mechanism implemented in the controller



a hard one. The distance between both limits defines how fast the motion is slowed down, which clearly has an impact on the load on the actuator. When choosing this distance, a compromise has to be made between the maximum allowed load, the performance of the system and the useful angular range of each component. This was determined using the Automated Co-model Analysis feature of the Crescendo technology presented in Sect. 10.2.

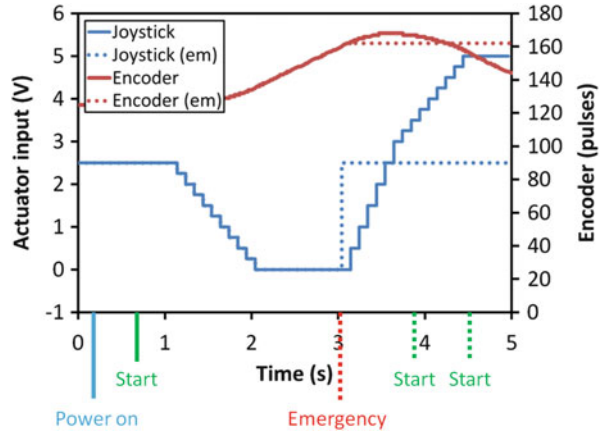
#### 11.2.4.4 Emergency Switch

In a final collection of co-simulation experiments, the effect of the emergency switch has been validated. Two co-simulations were performed, where first the system was powered on and then started. Then a certain input signal was given by the joystick and the encoder value of the actuator was logged, as shown in Fig. 11.14. In the second co-simulation, the emergency switch was pressed after 3 s, after which the excavator came to a halt. With the emergency activated, we then tried to restart the system, but the system did not respond, thus indicating that the emergency switch works correctly. Here the system can only be restarted after an entire power-down.

### 11.2.5 Key Results and Observations

Based on the experiences in the case study, Verhaert concluded that the Crescendo approach has added great value for modelling complex systems like the dredging excavator. They report that it enables teams with different technical backgrounds to work easily together on the same project. Interfaces can be managed by models instead of writing and maintaining lengthy documents (typically tens to several hundreds of pages). The software team can develop and verify the control software in combination with a relevant physical (yet virtual) model. The machine design

**Fig. 11.14** The joystick input and the encoder value as function of time. The different button presses are indicated below the time axis. The *dotted lines* represent the additional button presses in the second co-simulation, the one in which the emergency was activated and the operator tried to restart the system



team can verify the behaviour of the machine as driven with the control software and it can improve the mechanical (or hydraulic) design where needed.

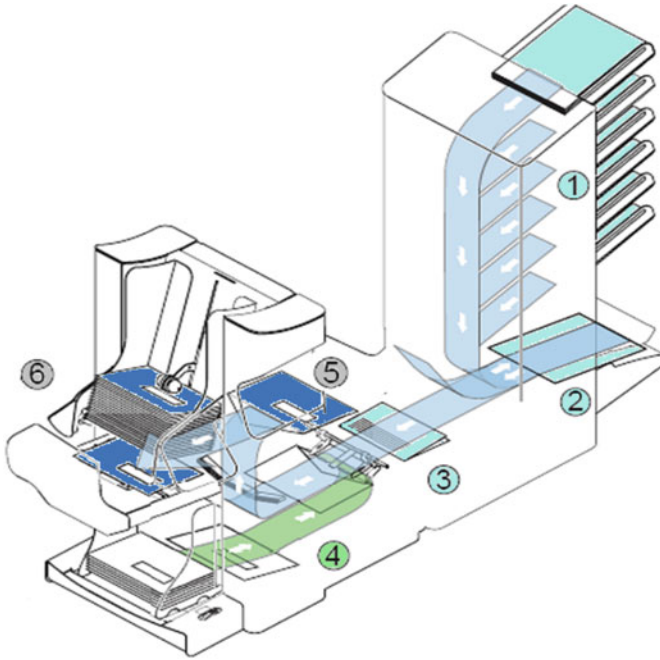
In their opinion, this is particularly interesting for complex controls, for example, multiple states of the excavator and inverse kinematics as part of the control, and for the combination of mechanics and hydraulics. More scenarios than in real life can be tested, and more parameters can be monitored. Our approach also allows for off-nominal or critical simulations that are difficult or dangerous to perform with real hardware. Upscaling the model from a test setup to full scale also allows the designer to check for differences in behaviour between lab scale and full scale. That information can be used to guide the design of the full scale machine and control.

By their detailed modelling of the controller, Verhaert discovered some shortcomings in the initial implementations of the controller that would have been hard to find with other simulation tools. For example while testing the end stops, Verhaert discovered that the assisted mode control behaved in a strange way. After this observation, some changes were made to their controller to accommodate this shortcoming. These changes were first checked in the model and then validated on the setup. Our approach also enabled Verhaert to efficiently validate their emergency supervisor. Last but not least, Verhaert modelled, implemented and successfully demonstrated a fully working version of their “assisted mode” excavator operation on their scale model test setup.

## 11.3 A Document Handling System

### 11.3.1 Case Description and Main Challenges

The Neopost case study concerns the model-based design of a document inserting system, whereby different sheets of paper need to be folded and inserted into envelopes. Such a document inserting system, as shown in Fig. 11.15, has a number



**Fig. 11.15** Layout of a Neopost document inserting system: (1) Document bins, (2) collator area, (3) folding area, (4) inserting area, (5) closing area and (6) document exit

of bins containing printed documents and a bin containing empty envelopes. Depending on settings for the application job, the system will separate documents from the available piles (1) in the right order, transport them sequentially to a collator area (2) until the document set is complete. Then, the set will be folded (possibly more than once) and transported to the insert area (3). Meanwhile, an envelope has been separated from the pile of envelopes. This envelope is transported to the insert area and opened mechanically (4). At the insert area, the folded set of documents will be inserted into the envelope (5). Finally, the filled envelope will be glued, closed and transported to the exit of the system (6).

Co-models were developed by Neopost to study the misalignments of documents with respect to each other and/or the heart line of the paper path. This is important, as large misalignments may lead to uncontrolled collisions between documents and envelopes which is critical to quality for document inserting systems, as these collisions may lead to paper jams and hence significant loss of productivity. Understanding how the different causes of misalignment contribute to the total misalignment of a set of folded documents during the insertion process might give direction to how to distribute the available effort over the different possible improvement activities, and this was the main research topic, using our approach.

Due to commercial confidentiality, we cannot show the actual DE and CT models of the document inserting system produced by Neopost. But fortunately, to a large

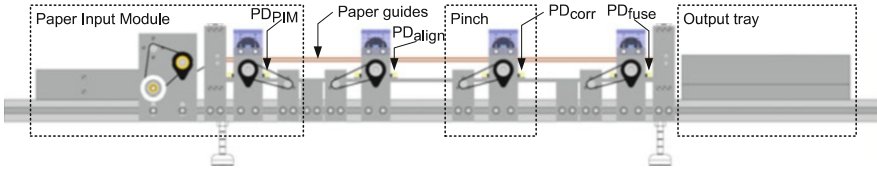


Fig. 11.16 Schematic overview of the experimental setup from [3]

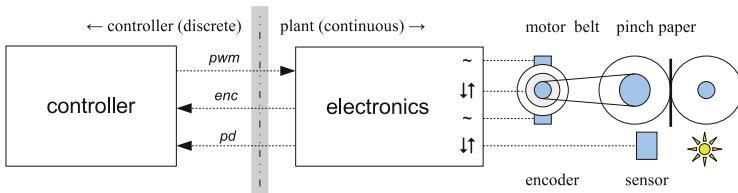


Fig. 11.17 Overview of the controller–plant I/O interface

extent the modelling of this device has great similarity to the paper path control for a digital printer that has been studied in the BODERC project [96]. Thus, we will describe the main characteristics of the BODERC paper path and its experimental setup instead here. Figure 11.16 presents a side view of the paper path. It consists, from left to right, of a single paper input module with a built-in sheet separation device, and four pinches to transport the paper towards the output tray. Each pinch is driven by an electric motor through a belt and each motor is equipped with a rotary encoder. Simple paper guides are used to lead the paper from pinch to pinch and optical sensors are available to detect the edges of the sheet.

Figure 11.17 presents the overall controller architecture, for a single motor–pinch pair. The digital controller produces a PWM signal which is connected to a power amplifier. The power amplifier produces an electric current that is proportional to the PWM signal. The motor torque is in turn proportional to the current supplied by the amplifier. The motor torque is transferred to the pinch through a belt, while rotation of the motor axle (and *not* the pinch axle) is measured by a rotary encoder. Each pinch consists of two rubber rollers which are mounted such that their surfaces touch. Paper can then be transported due to friction by inserting a sheet in between the rubber rollers. The position of the leading and trailing edge of a sheet can be measured with an optical sensor, the paper detector, which triggers as soon as the light levels are changed by the passing sheet.

The paper path setup emulates the behaviour of a high-volume digital printer, whereby each motor–pinch pair has a specific control function to perform. The first pinch is used to stabilise the speed of the paper coming from the paper input module. The second pinch is used to decelerate, stop and accelerate each sheet in order to simulate the alignment process. The purpose of the third pinch is to deliver the sheet at exactly the right time and at the right constant speed at the fourth pinch. The fourth and final pinch represents the printing process, where the image is put on the sheet. This pinch delivers each sheet in the output tray. The overall control goal is to

maximise throughput while ensuring that individual sheets never touch each other while in transit, and ensuring that alignment of each image on each sheet remains within the set accuracy requirement. We will first introduce the continuous time model in Sect. 11.3.2 and then the discrete event model in Sect. 11.3.3. Afterwards Sect. 11.3.4 presents the co-simulation analysis conducted on this co-model and finally Sect. 11.3.5 provides an overview of the key results of this study.

### *11.3.2 The Continuous Time Model*

The top-level bond graph model of the plant is shown in Fig. 11.18. At the bottom of the figure, we see the interface towards the controller. There is a pair of PWM and encoder signals connected to each motor-belt-pinch icon.<sup>1</sup> These icons represent lower level bond graph models, or submodels, which we will present later in more detail. The plant model has four motor-belt-pinch submodels while our experimental setup has five. The first motor in the setup is only used to inject new sheets into the paper path. Since its operation is only of minor importance to the total system behaviour it is only abstractly represented in the plant model by means of the `FeedSheet` signal. The behaviour of the individual sheets is represented by the photographic icon. This submodel maintains the state of each sheet in the paper path, such as for example its current speed and position. The state of the `PaperDetectors` signal is automatically derived from this information. If the position of a paper detector is in between the leading and trailing edge of at least one sheet then it will yield 1 else 0.<sup>2</sup> Similarly, the sheet is under the control of a pinch if the position of the pinch is in between the leading and trailing edge of the current sheet position. The animation icon is used as a monitor which allows us to visualise the simulation graphically.

The pinches drive the sheets and this transfer of energy is influenced by friction. In kinematic models the friction is assumed to be zero but this is usually not very realistic. The friction force which is imposed on each sheet is a function of the mass of the sheet and the speed difference between the sheet and pinch. Of course, the friction force is imposed if and only if the sheet is in control of a pinch. What happens if a sheet is under the control of two pinches? In our model, we assume that the pinch near the leading edge of the sheet dominates the pinch near the trailing edge. The assumption is that the force imposed by the leading edge pinch will cause the sheet to slip in the trailing edge pinch. This abstraction can be used if and only if the speed of the leading edge pinch is equal to or slightly higher than the speed of the trailing edge pinch. This condition can be checked at simulation time. Only a very trivial friction model is used here, but it can simply be replaced by more complex hybrid friction models if the need arises, without affecting the plant model architecture demonstrated here.

---

<sup>1</sup>The Neopost document handling system also makes use of pinches for transporting the paper.

<sup>2</sup>Similar paper detectors are used in the Neopost document handling system.

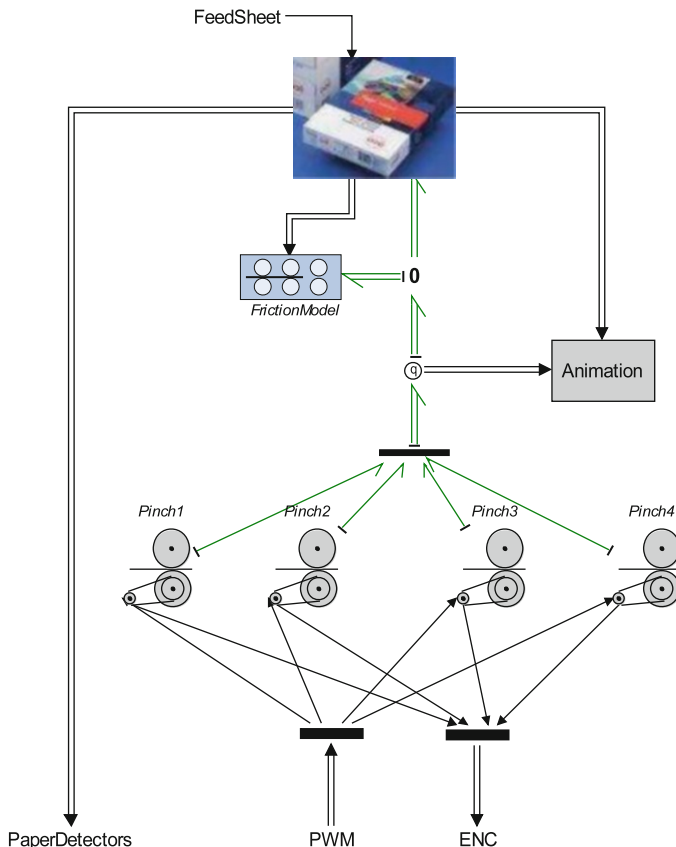


Fig. 11.18 Top-level bond graph of the plant model

The bond graph submodel for the motor, belt and pinch is presented in Fig. 11.19. This iconised diagram demonstrates at a very high level of abstraction how the control signal relates to the movement of the sheet. For example, observe that the behaviour of the power electronics, the so-called H-bridge, has been modelled as a simple multiplication (or gain) factor. In other words, the amount of power provided to the motor is linearly proportional to the duty cycle of the PWM input signal obtained from the controller. The motor converts this electrical energy into torque. The torque causes the belt to rotate and the belt in turn drives the pinch, whereby the “Belt and Gear” submodel simply multiplies the rotational speed of the motor by the gear ratio. And finally, the pinch transfers its energy towards the sheet of paper as described previously. The angular velocity is measured at the motor axis and this value is multiplied by  $2\pi$  to obtain the number of rotations per second. We will see later how this value is converted into encoder values in the detailed I/O interface model.





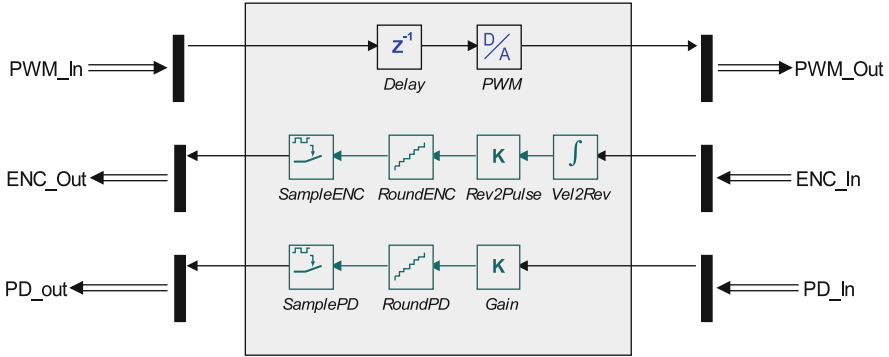


Fig. 11.21 Detailed overview of the I/O interface model

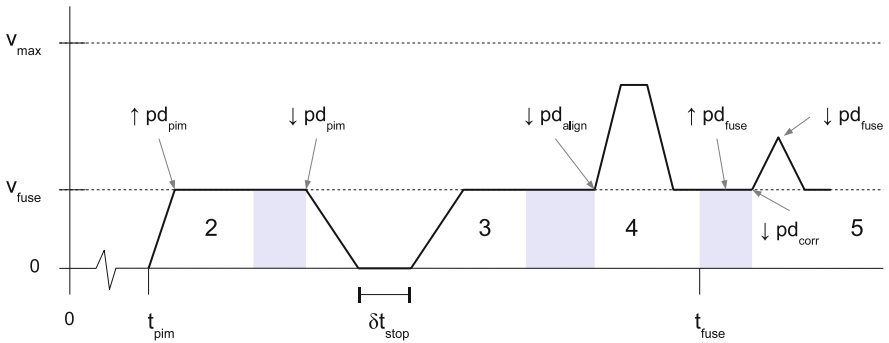


Fig. 11.22 Overview of a typical sheet velocity profile

This velocity  $\omega$  is determined by the required system throughput performance as described in the following equations:

$$V_{nom} = (pagesize + isd) \cdot tp/60 \quad (\text{mm/s}) \quad (11.1)$$

$$\omega = V_{nom}/2\pi \cdot r_{pinch} \quad (\text{rad/s}) \quad (11.2)$$

whereby *pagesize* represents the length of a sheet (in mm), *isd* represents the inter-sheet distance (in mm), *tp* represents the throughput (in pages per minute) and finally  $r_{pinch}$  represents the radius of the pinch (in mm). The inter-sheet distance is defined as the distance between the trailing and leading edge of two consecutive sheets. The primary task of the paper path subsystem is to deliver each sheet on time and at the right speed at the pinch.<sup>3</sup> This requirement has two implications:

<sup>3</sup>For the Neopost document handling system, similar requirements are present for preventing skewing of paper when folding.

1. With respect to *on time*. The inter-sheet distance shall be maintained in order to meet the required system performance and to ensure the correct alignment of the image on the sheet. A maximum deviation of 0.5 mm is allowed exactly at the pinch but the inter-sheet distance may vary elsewhere as long as two consecutive sheets do not collide or overlap.
2. With respect to *right speed*. The leading edge of each sheet shall have the nominal speed  $V_{\text{nom}}$  just before it is in control of the pinch. A maximum deviation of  $V_{\text{nom}}$  of 2 % is allowed.

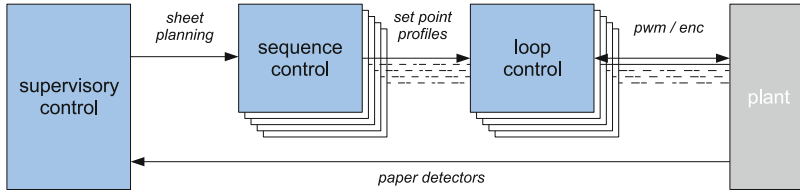
Now the main control goal has been identified, we can look at the secondary tasks to perform by the control application. Considering the pinches in the experimental setup, we have the following additional requirements:

1. The first pinch is part of the paper input module and it is used to retrieve sheets from the tray. The challenge is to ensure that single sheets are separated. The solution is to control this motor belt pinch subsystem in open loop. Basically the motor is told to accelerate as fast as possible for a very short period of time and then immediately decelerate. The friction force between the pinch and the top sheet is larger than the friction force between the top two sheets, which will cause clear separation of a single sheet.
2. The purpose of the second pinch is to get the sheet under control by moving it down the paper path at the nominal speed  $V_{\text{nom}}$ .
3. The purpose of the third pinch is to decelerate, stop and accelerate the sheet. The length of the stop period can be defined by the super-user setting up experiments on the paper-path.
4. The purpose of the fourth pinch is to ensure that the sheet is delivered with the correct inter-sheet distance and speed to the exit of the paper path. It will have to compensate for the time lost during alignment of the sheet at the previous pinch.

The control application will have to satisfy all these sub-goals simultaneously. The behaviour of a sheet, in terms of its speed through the paper path, is graphically presented in Fig. 11.22. The numbers 2–4 correspond to the pinch that is in control of the sheet at a given point in time. The grey areas indicate where the paper is in control of two pinches simultaneously.

We will take a step-by-step look at the lifetime of a sheet during its travel through the pinches in order to get a feeling for the control complexity involved. The events mentioned are also shown in Fig. 11.22 (similar to the DE-first approximations shown in Fig. 8.16b).

1. A new print job arrives and the image processing starts. Meanwhile, pinches 2–4 are booted up until they run at  $V_{\text{nom}}$  and then the first sheet is requested from the paper input tray. The sheet is separated by pinch 1 and it is inserted into the paper path. It will hit pinch 2 with some force and at the wrong speed since we use a rather brute force separation method.
2. Pinch 2 accepts the first sheet and will try to stabilise its speed to  $V_{\text{nom}}$ . The timing tolerance caused by the brute force separation is known when the leading edge of the sheet is detected by  $\text{PD}_{\text{pim}}$ , as shown by the  $\uparrow$ -arrow.



**Fig. 11.23** An informal overview of the three tier control application architecture

3. The sheet continues to move downstream and gets into joint control of pinches 2 and 3 (the first grey area in the figure). The control application knows that the sheet has left the control of pinch 2 when the trailing edge of the sheet is detected by  $PD_{pin}$ , as shown by the  $\downarrow$ -arrow. The alignment phase of the sheet can now start because pinch 3 is in full control. The deceleration needs to be quick enough to ensure that the leading edge of the sheet does not reach pinch 4.
4. The sheet is accelerated to  $V_{nom}$  after the user-defined alignment time  $\delta\tau_{stop}$  has expired. The acceleration must be performed quickly to ensure that the nominal speed has been reached before the leading edge of the sheet hits pinch 4.
5. The sheet continues to move downstream and gets into joint control of pinches 3 and 4 (the second grey area in the figure). The control application knows that the sheet has left control of pinch 3 when the trailing edge of the sheet is detected by  $PD_{align}$ , as shown by the  $\downarrow$ -arrow. The correction phase of the sheet can now start because pinch 4 is in full control. The sheet is accelerated to compensate for the time lost in the alignment phase and the tolerances caused by the brute force sheet separation.

The informal description of the requirements for the control application listed above gives us some inspiration for the control application architecture that is required to address these challenges. From the engineering point of view, it is usually a good idea to apply the “separation of concerns” principle, for example to divide the time critical parts from the less time critical parts of the application. In fact, we will provide each motor-belt-pinch subsystem its own controller in our architecture because the requirements differ and they may be deployed on different hardware in the final implementation. In contrast, the timing requirements for the high-level sheet flow control, as presented in the sheet life-cycle are far less demanding and a single application may suffice for this purpose. The linking pin between this high-level supervisory control layer and the real-time controllers is a set of so-called sequence controllers, one per real-time controller. These sequence controllers generate so-called setpoint profiles ahead of time, based on the planning information received from the supervisor. An informal overview of this well-known three tier control application architecture is shown in Fig. 11.23.

An extract of the VDM-RT models for the controller application architecture is presented in a bottom-up fashion. We start at the plant model interface and the real-time loop controller and work our way up towards the supervisory control. Each motor-belt-pinch subsystem has an interface consisting of a PWM input

and encoder output (ENC). This interface is well suited for feedback control. A standard PID control strategy will be used for pinches 2–5. The real-time controller will periodically sample the encoder value. This value is a measure for the distance covered and it is compared to the so-called setpoint, which represents the intended value. The difference between the two is called the error and with the PID algorithm we calculate a new PWM value to compensate for this measured error, by accelerating or decelerating the motor accordingly. The four PID loop controllers will operate at 1 kHz in our controller models. Open loop control is used for pinch 1. Basically, the setpoint is forced upon the motor-belt-pinch system by writing the correct PWM value but the encoder value is ignored. For convenience, the loop controller for pinch 1 will also run at 1 kHz.

### 11.3.3.1 The Loop Controller

Consider the VDM-RT model for the loop controller shown below. The constructor of the active class `LoopController` takes two arguments. The first argument, `ptp`, determines whether the calculated output value is sent to the plant model at the start of the next iteration or immediately. The second argument, `pfb`, is used to distinguish the control strategy: closed loop or open loop.

```

class LoopController

instance variables
  -- time-triggered (true) or immediate output (false)
  private hold : bool := true;
  -- closed loop (true) or open loop (false)
  private feedback : bool := true

operations
  public LoopController: bool * bool ==> LoopController
  LoopController (ptp, pfb) ==
    ( hold := ptp; feedback := pfb )

```

The instance variable `output` is used to temporarily store the calculated PWM value. The operation `CtrlLoop` implements the real-time control strategy and this operation is periodically executed.

The operation `getSetpoint` used inside `CtrlLoop` retrieves the setpoint from the passive `SetpointProfile` object for the current local time `ltime`. Setpoint profiles are the interface between the loop and sequence controllers. The setpoint profile can be updated by the sequence controller by calling the asynchronous `addProfileElement` operation. The operations `getSetpoint` and

```

instance variables
  private output : real := 0;
  private ltime : real := 0

operations
  public CtrlLoop: () ==> ()
  CtrlLoop () ==
    -- first retrieve the current encoder value
    ( dcl enc : real := getEnc();
      -- update the old output if time-triggered
      if hold then setPwm(output);
      -- calculate the new PWM value
      output := if feedback
                then limit(calcPID(enc)) -- closed loop
                else limit(getSetpoint()); -- open loop
      -- update the output if not time-triggered
      if not hold then setPwm(output) )

thread
  -- execute the controller at 1 kHz
  periodic (10E6, 0, 0, 0) (CtrlLoop)

```

addProfileElement are declared mutual exclusive to prevent data corruption by simultaneous access to the profile instance variable.

```

instance variables
  profile : SetpointProfile := new SetpointProfile()

operations
  private getSetpoint: () ==> real
  getSetpoint () == profile.getSetpoint(ltime);

  async public
  addProfileElement: real * real * real ==> ()
  addProfileElement (px, py, pdt) ==
    profile.addElement(px, py, pdt)

sync
  -- access to the profile is mutual exclusive
  mutex (addProfileElement, getSetpoint);
  mutex (addProfileElement)

end LoopController

```

### 11.3.3.2 The Setpoint Profile

The passive class `SetpointProfile` is used as a container to collect all knowledge about manipulating so-called setpoint profiles. A setpoint profile is an ordered collection (a sequence) of left-closed, right-opened, line elements which together define the evolution of the setpoint over time similar to the strategy followed in Sect. 8.5.4. Each line element, or `ProfileElement`, is defined by three real numbers. The first number defines the domain: the starting time  $t$  at which this element is valid. The second and third number define the range: the current value at time  $t$  and the direction coefficient that is valid from this point in time onwards respectively. Setpoint profiles are defined from some point in time to infinity, since the last element in the profile is right-opened. The invariant of the profile instance variables ensures that the domain is strictly monotonically increasing but it does allow discontinuities in the range. The operation `addElement` can be used to extend the current setpoint profile.

```

class SetpointProfile

types
  private ProfileElement = seq of real
  inv pe == len pe = 3

instance variables
  profile : seq of ProfileElement := [];
  inv forall i, j in set inds profile &
    i < j => profile(i)(1) < profile(j)(1)

operations
  public addElement: real * real * real ==> ()
  addElement (t,v,a) ==
    profile := profile ^ [[t,v,a]]
  pre len profile > 0 => profile(len profile)(1) < t

```

The operation `getSetpoint` is used to compute the actual setpoint at some specific point in time based on linear interpolation of the abstract continuous time description maintained in the `profile` instance variable. Consider the example

```
[ [0,0,0], [1,0,1], [2,1,0], [4,1,-1], [5,0,0] ]
```

This trapezoid setpoint profile ramps up from  $\langle 1, 2 \rangle$  and ramps down from  $\langle 4, 5 \rangle$ . Hence, `getSetpoint(1.5)` would yield the value 0.5.

```

operations
public getSetpoint: real ==> real
getSetpoint (t) ==
  if len profile = 0
  then return 0
  else ( dcl prev_pe : ProfileElement := hd profile;
         for curr_pe in tl profile do
           if curr_pe(1) > t
           then return calcSetpoint(t, prev_pe)
           else prev_pe := curr_pe;
         return calcSetpoint(t, prev_pe) )
pre t >= 0 and len profile > 0 => t > profile(1)(1)

functions
private calcSetpoint: real * ProfileElement -> real
calcSetpoint(t, [px, py, pdydx]) ==
  py + pdydx * (t - px)
pre t >= px

end SetpointProfile

```

### 11.3.3.3 The Sequence Controller

The active class `SequenceController` contains the knowledge to translate high-level paper path planning commands into setpoint profiles that are used by the loop controllers. Each sequence controller is associated with exactly one loop controller `loopctrl`.

```

class SequenceController

instance variables
  public loopctrl : [LoopController] := nil

```

The operation `initNominal` is used to power-up the pinches until they reach the nominal paper path speed  $v_{\text{nom}}$ . The motors are not started at full throttle immediately, but they are ramped up gradually. The user can influence the power-up time by setting the acceleration parameter  $a_{\text{nom}}$ .

The operation `setStopProfile` is used to bring the sheet in the paper path to a complete stand still for  $d_{\text{stop}}$  seconds. The procedure will start at  $t_1$  with speed  $v_1$  mm/s and the sheet will accelerate and decelerate with  $\text{acc}$  mm/s<sup>2</sup>.

**operations**

```

async public initNominal: real * real ==> ()
initNominal (v_nom, a_nom) ==
  ( -- ramp up the motor to the nominal paper speed
    loopctrl.addProfileElement(0, 0, a_nom);
    -- and maintain a constant speed indefinitely
    loopctrl.addProfileElement(v_nom/a_nom, v_nom, 0))
  pre v_nom > 0 and a_nom > 0 and loopctrl <> nil;

async public initPeak: real ==> ()
initPeak (tpeak) ==
  -- give the sheet a good kick for 60 msec
  ( loopctrl.addProfileElement(tpeak, -40, 0);
    loopctrl.addProfileElement(tpeak+0.060,0,0) )
  pre loopctrl <> nil

```

**operations**

```

async public
setStopProfile: real * real * real * real ==> ()
setStopProfile (t1, v1, acc, dstop) ==
  def dt = v1 / acc in
    ( loopctrl.addProfileElement(t1, v1, -acc);
      loopctrl.addProfileElement(t1+dt, 0, 0);
      loopctrl.addProfileElement(t1+dt+dstop, 0, acc);
      loopctrl.addProfileElement(t1+dt+dstop+dt, v1,0))
  pre acc <> 0 and loopctrl <> nil

end SequenceController

```

### 11.3.3.4 The Supervisory Controller

The active class `Supervisor` represents the supervisory control in our architecture. It has four instance variables of type `SequenceController`. The links to these objects are created at model instantiation time. Each sequence controller takes care of one motor-belt-pinch subsystem.

The core functionality of the supervisory control application is captured in the operations that respond to the paper detectors. For example, the `pimDownEvent` operation will be called whenever a trailing edge of a sheet has been detected by paper detector `PDpim`. This event signals the start of the alignment process which will bring the sheet to a complete stand still, in our case for 100 ms.



```

class Supervisor

instance variables
  public ejectSeqCtrl : [SequenceController] := nil;
  public pimSeqCtrl   : [SequenceController] := nil;
  public alignSeqCtrl : [SequenceController] := nil;
  public corrSeqCtrl  : [SequenceController] := nil;
  ...

operations
  -- operation to initiate the alignment procedure
  async public pimDownEvent: () ==> ()
  pimDownEvent () ==
    -- start decelerating in 10 msec from now
    def dectime = time + 0.01 in
      alignSeqCtrl.setStopProfile(dectime, 50, 500, 0.1)
    pre alignSeqCtrl <> nil

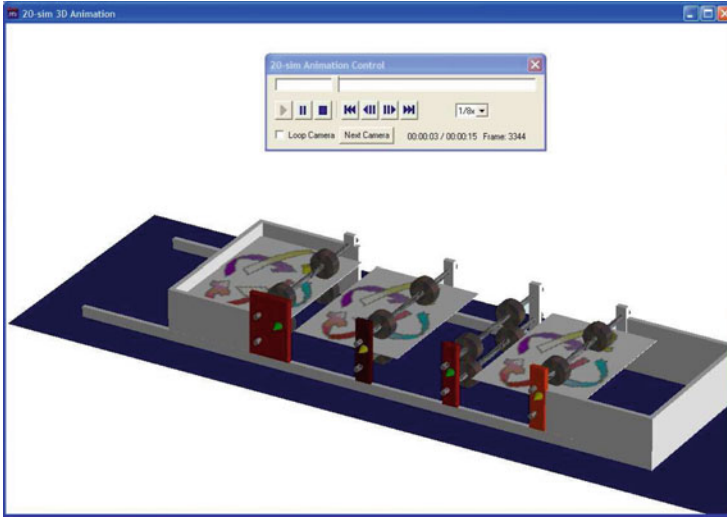
```

### 11.3.4 Co-simulation Analysis

In the case of the paper path models presented in the previous section, we encountered two significant problems that were only exposed during co-simulation. Interestingly, neither problem was found during CT-only analysis nor during DE-only analysis, which demonstrates the added value of our approach.

The first problem was a simple mistake with potentially severe consequences. The VDM-RT controller model used mm/s as the unit of measure in the setpoint profiles, but the loop controller measured the distance covered in radians. Hence, the integrated setpoint values provided to the PID controller were incorrect, causing the wrong output values to be calculated because the error was off the chart at every iteration, leading to the constant spin-up of the motor at maximum speed. The root cause of this problem was easily identified since it is very simple to monitor model parameters during simulation. It would have taken substantially more time and effort if the cause of the problem had to be investigated on the embedded target (implementation).

The second problem was slightly more complex but is also due to a misinterpretation of the informal requirements. The designers of the plant model assumed that the time earmarked for the alignment of the sheet also included the time required to decelerate the paper. However, the designers of the controller model followed a strict interpretation of the requirement: the time needed to decelerate is not included in the alignment time. The designers of the plant model performed a simulation using a simplified controller model and claimed that an inter-sheet distance of 50 mm was feasible at a productivity rate of 50 pages per minute and an alignment time of



**Fig. 11.24** 3D visualisation of the paper path co-simulation in 20-sim

200 ms. However, when the experiment was performed on the experimental setup it turned out that their controller model was incorrect and they circumvented the problem by increasing the inter-sheet distance to 100 mm, reduced the alignment time to 100 ms and operated the alignment motor with maximum acceleration and deceleration values.

As sometimes happens in real life, these lessons learnt were not properly documented and communicated. Therefore, the designers of the controller model based their design on the wrong data. This issue became very clear when the controller model was tested in combination with the plant model. Both the different assumption about the same requirement as well as the lack of communication of the insight gained from the plant experiments were easily identified using the visualisation capabilities of 20-sim. The plant model designers developed a three dimensional model of the paper path as a plug-in to their plant model. This interface is shown in Fig. 11.18. The visualisation runs in parallel with the co-simulation, whereby the virtual prototype is fully synchronised with the simulation state. It also provides the ability to stop, rewind and replay the visualisation such that the system behaviour can be inspected in detail. Using this facility it was demonstrated convincingly that two consecutive sheets would always collide if the original parameters were used. An example of the visualisation is shown in Fig. 11.24.

Both examples illustrate a key problem occurring in industrial practice: errors are made in critical design parameters (unvalidated assumptions) due to lack of communication in the design team. Co-simulation exposes these issues early in the development life-cycle.

### ***11.3.5 Key Results and Observations***

Neopost Technologies confirms these findings from the BODERC project as they had similar experiences as those listed in the paper path case study. The co-modelling approach forced the CT- and DE-groups of engineers to communicate right from the start of the project, which was experienced as being of great value for the mutual understanding and overall project progress.

With respect to the system under development, Neopost identified and solved several flaws in the system design due to the co-modelling work performed. Actually, these defects were discovered even before physical parts of the system were realised, which saved several weeks of development time. In fact, the entire software implementation was written and tested using a system mock-up that was directly based on the Crescendo co-simulation models. This allowed parallel development of both hardware and software and also significantly reduced the time required to perform software integration when the physical system became available. This integration process was also smoother than usual, saving again several man months in development lead time. But even when the hardware became available, the Neopost engineers kept using and maintaining the development approach based on the Crescendo models, because of the following:

- In the early stages of the project, the simulator was working more reliably and was more flexible than the real physical system. This meant that by using the simulator instead of the real machine, the engineers could concentrate on the development of their software instead of keeping the mechanics running.
- The availability of the simulator was greater than the availability of the real machine. The system prototype turned out to be under constant development and near constant maintenance led to low availability for the software engineers. This became of crucial importance when the development team grew and only a few systems were available.

## **11.4 The ChessWay Self-balancing Scooter**

### ***11.4.1 Case Description and Main Challenges***

The ChessWay case study and its main control challenges were introduced in Sect. 7.3, and models of the ChessWay have been used to illustrate the DE-first approach to co-model creation (Sect. 8.5.4), and approaches to the modelling of faults and fault tolerance (Sect. 9.8). In this section, we report on the actual development of the ChessWay study within Chess.

One of the main motivations for the study was the need to explore the range of potential faults. In fact, before following the DE-first approach presented in Sect. 8.5.4, an abstract model was produced of the full system with the purpose of identifying interfaces, considering safety aspects and analysing potential fault

handling strategies as early as possible. The emphasis of this model has been on initial exploration of the system complexity, rather than on model structuring and analysis from a traditional object-oriented point of view, or co-simulation. However, this model provided valuable insights that have been transferred over into the DE-first approach shown in Sect. 8.5.4. An overview of this initial model can be found in Appendix D.

### 11.4.2 *The Continuous Time Model*

The initial CT model described the dynamic behaviour of the ChessWay self-balancing scooter limited to two-dimensional space, so it could only ride forward and backwards, and assumed only a single-driven wheel, whereby the digital (closed loop) control was embedded deep within the CT model. When co-simulation with the initial DE model was acceptable, the CT model was expanded to a 3D setting, with two independent driven wheels including steering, and potential faults were also incorporated. Moreover, the closed loop control was partly moved from the CT model to the DE model. This top-level CT model is presented in Fig. 11.25. At the top, we see two independent PWM signals, coming from the DE model, that drive the power amplifiers, one for each wheel. Each wheel has an explicit contact model which reflects the energy transfer between the wheel and surface, which in turn might depend on its actual location. This allows the execution of experiments with different surface models, independent from the system model. Last but not least, the CT model provides the current forward speed and the angle of the handlebar as input back to the DE model.

### 11.4.3 *The Discrete Event Model*

A distributed controller architecture was chosen for the ChessWay, whereby each wheel has its own controller. This was done on purpose in order to demonstrate and expose typical reliability challenges that occur in these type of architectures. Each motor controller is guarded by a *safety*, which has the task to intervene and put the system in a fail-safe state if a fault is detected. The fail-safe state condition for the ChessWay is the situation whereby both motors are not actuated and free running. Furthermore, the system should be able to recuperate from such an intervention and return to normal operating mode if the root cause of the fault has been removed, for example, due to the user's intervention.

An overview of the distributed controller architecture of the ChessWay is shown in Fig. 11.26. Each controller has its own safety monitor and a motor controller. The architecture seems mirrored, but note that only one of the safety monitors has direct physical access to the safety key, the other safety monitor needs to communicate over the bus connecting the two controllers to access the device remotely, see Fig. 7.5. If the safety key is removed, then the safety monitor will move to the

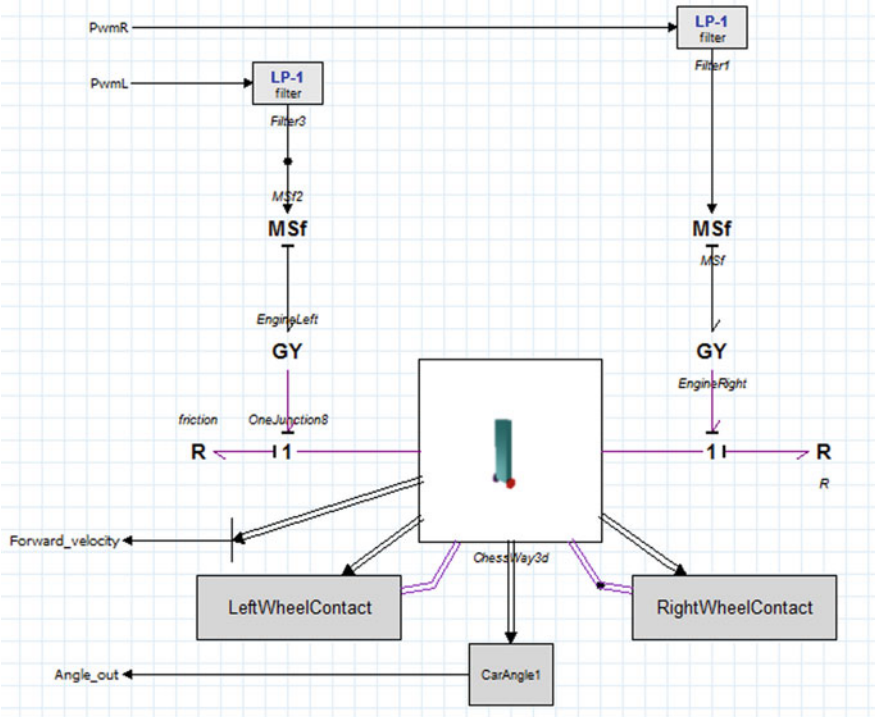


Fig. 11.25 Top-level CT model of the ChessWay

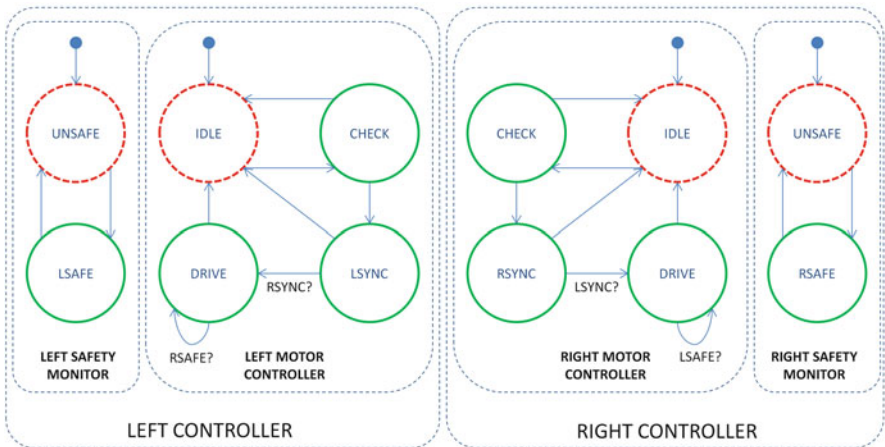


Fig. 11.26 Distributed DE controller architecture

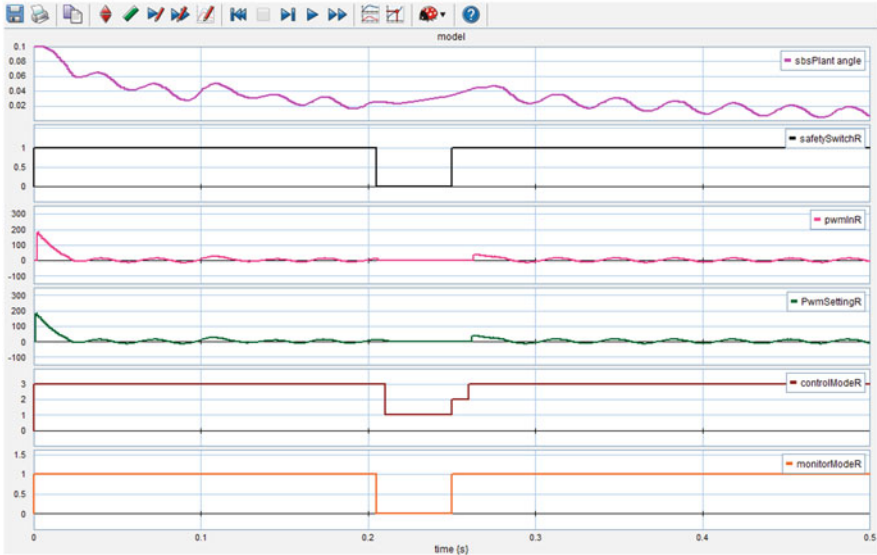
UNSAFE state and the wheel will be undriven. In that case, the safety monitor will force the motor controller to go to the IDLE state. Note that the safety monitor only affects the local motor controller so some synchronisation is required with the other controller. If the safety key is inserted then the motor controller will move from IDLE to CHECK mode. The motor controller will move from CHECK to SYNC mode, whenever the ChessWay has been kept upright for at least 3 s (this test is not shown here). This is to prevent the user being hit by the handle bar when the ChessWay is lying flat when the power is turned on. The motor controller will move from SYNC to DRIVE mode if and only if the other motor controller is also in the SYNC state. The general idea being that both motor controllers will proceed to DRIVE mode at the same time. The motor controller will execute the control algorithm to keep the ChessWay upright and steer only while in DRIVE mode. It will continue to remain in DRIVE mode for as long as the safety monitors on both controllers are in the SAFE state. Again, note that the motor controller needs to communicate to its neighbour to inquire its current safety state. This implies that the communication path between the left and right controller potentially affects the system reliability, as communication may corrupt or delay data, or even fail entirely.

#### 11.4.4 Co-simulation Analysis

Results of a co-simulation are shown in Fig. 11.27, with, from top to bottom:

- The pole angle (in radians).
- The state of the safety switch (1 = CLOSED, 0 = OPEN).
- *PWMInR*, the right-hand-side PWM value, being the equivalent to the average current (in A) to the motor. This should be the same as the following curve, multiplied with the safety switch. That means that as soon as the safety switch is opened by the safety monitor, the *PWMInR* will become zero, which is indeed the case.
- *PwmSettingR* the right-hand-side PWM setting as it is transferred from DE to CT. This signal continues until the state machine of the right-hand-side controller has changed its state from DRIVE to CHECK. This is shown in the graph.
- Status of the right-hand-side controller (0 = IDLE, 1 = CHECK, 2 = SYNC, 3 = DRIVE).
- Status of the right-hand-side safety monitor. (0 = UNSAFE, 1 = SAFE).

We show here a specific simulation to demonstrate the cooperation between the DE and CT part of the co-simulation. It shows the closed loop control, stabilising the vehicle and handling of a simulated error. The vehicle starts in an off-balance position of 0.1 rad in the controller state DRIVE and safety state SAFE. The real-time controller generates a PWM signal to bring the pole angle to zero rad. Initially the PWM signal peaks to generate a relatively large current, to force the pole angle towards zero to prevent the vehicle from falling. At 0.205 s, a fault was introduced by pulling the safety key. Immediately the safety monitor changes from SAFE to



**Fig. 11.27** Simulation result: (1) the controller forces the vehicle to change from 0.1 rad approaching zero, (2) handling of an error that is introduced at 0.205 s and (3) recovery of that error from 0.25 s onwards

UNSAFE and the safety switch changes from CLOSED to OPEN. The PWM signal (average current to the motor) should become zero here as well, to simulate open loop wires to the motor (safety switch open). The control status changes to CHECK as well, but a little later. This is to show that the emergency fault handling acts faster and therefore overrules the slower running controller. The safety key is inserted again at 0.25 s, which causes the safety monitor and safety switch change to SAFE and CLOSED, respectively. The controller detects the safe state and changes to SYNC and after both sides are in SYNC, it changes state to DRIVE again after which the motor is powered by a PWM signal.

The diagram nicely shows the relationship between the dynamic behaviour of the vehicle and the state changes of the controller, monitor and safety switch. It also clearly demonstrates the purpose of having the graphical presentation of the DE states in the same diagram as the CT graphs.

### 11.4.5 Key Results and Observations

The DE model helped the stakeholders to differentiate precisely the real-world phenomena that take place and their interpretation (observation) in the control software. Originally, a fault was injected in the DE model, and therefore directly seen by the DE model and reacted to instantaneously. However, in real products, there is always a delay between the occurrence of a fault to its detection by sensors

and followed by the response of the control software. This leads to a difference between the software's picture of the physical system, and the physical system's real state.

It is important to take account of this delay because too large a detection time can cause hazards to users. This distinction has been modelled by the explicit definition of distinct plant (real world) and software states. The software interacts with the plant via sensors and actuators; errors and faults in the plant are discovered by the software only via sensors, just as in real systems. The software transforms one state instance to the other. Several processes that handle such actions have been identified, for example

- An actuator representing the PID controlled pulse with modulator of the motor current.
- A controller, being a state machine for discrete control, such as startup, shut down, handling of faults.
- A safety monitor to detect errors and perform an emergency reaction independent from the controller.

The DE modelling has helped to identify these independent processes early and they have been maintained throughout the evolution of the models, leading to a very high correspondence between the abstract initial models and the elaborated models used for co-simulation. This has added to the confidence in the model correctness and likewise for the (manual) implementation that followed. The only errors that were found in the implementation had to do with the “glue code” between the computing hardware and the handwritten implementation of the model. As the structure of the implementation followed the structure of the co-simulation models, it was relatively easy to spot and fix the mistakes made, which reduced the test- and integration effort significantly.

The ChessWay case study contributed a great deal to identifying the *conceptual* boundaries between the DE and CT models, as reflected in Chap. 8, including the insight that the interface between them is actually a layer in its own right, as shown in Figs. 3.40 and 11.21. The co-simulation contract basically reflects an arbitrary “knife-cut” boundary drawn right through this I/O layer. Consequently, some elements of the interface end up as part of the DE model while the rest becomes part of the CT model. This not only affects the way the system is described but it also affects the performance of the co-simulation, sometimes even significantly. Hence, making this “knife-cut” demands careful consideration in relation to the required fidelity of the model.

## 11.5 Conclusion

We have briefly presented three industrial case studies that have used our approach. The ChessWay case has been discussed in detail in earlier chapters. For the new studies presented here—Verhaert's dredging excavator and Neoposts's document



handling system—the use of co-modelling had positive technical consequences when compared to alternative approaches. These results are, of course, encouraging, but we also know from experience that the successful deployment of modelling technology requires more than a technical capability [84]. In Chap. 12, we record the lessons learned by Chess, Verhaert and Neopost about the impact on industrial development practice of introducing co-modelling and co-simulation using the Crescendo technology.

**Part III**  
**Advanced Topics**

# Chapter 12

## Deploying Co-modelling in Commercial Practice

Sune Wolff, Peter Gorm Larsen, and Marcel Verhoef

### 12.1 Introduction

We have gained experience in the first industry deployments of collaborative modelling and co-simulation technology in a variety of settings. In this chapter, we describe the “life stories” of model construction, analysis and maintenance from those studies and include suggestions for integrating the technology into established development processes.

However strong the potential benefits of a new modelling or design technology, the process of industrial deployment brings many interesting challenges relating to cost-effectiveness, overcoming barriers to entry, the strength of tool support and the characteristics of the engineering skills base. This chapter records the experience gained applying collaborative modelling and co-simulation technology in the development of Neopost’s document handling system and Verhaert’s dredging excavator case studies introduced in Chap. 11, as well as Chess’ ChessWay case study introduced in Chap. 7.

The text here is based on interviews with engineers from all three companies: we introduce them in Sect. 12.2. In Sect. 12.3, the engineers describe how the cases would have been developed using traditional means, and in Sect. 12.4 they describe how the co-modelling and co-simulation integrated with existing development methods. Section 12.5 gives an overview of the resources used in developing the case studies, and Sect. 12.6 describes the challenges encountered. Section 12.7 discusses the main benefits of using the Crescendo technology. Finally, in Sect. 12.8, the future of co-modelling is examined.

---

S. Wolff (✉) • P.G. Larsen  
Aarhus University, Aarhus, Denmark  
e-mail: [swo@eng.au.dk](mailto:swo@eng.au.dk); [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

M. Verhoef  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

## 12.2 Company Introductions

Neopost employs around 70 R&D engineers with an equal split between engineers with backgrounds in mechanical and software engineering. Approximately 30 % of the engineers have a master's degree in computer science or mechanical engineering. Four engineers were involved in the development of the Crescendo model of the document handling system: one with a PhD in control engineering; one with a master's degree in mechanical engineering; one with a master's degree in electronics; and Peter van Eijk, who was interviewed for this chapter, is a PhD in physics.

Chess employs development engineers with a bachelor's or master's degree and a few with a PhD in either electrical engineering, computer science, mathematics or physics. Five engineers were involved in the development of the Crescendo model of ChessWay: one with a bachelor's degree in control engineering, two with a bachelor's degree in electrical engineering, one with a PhD degree in discrete-event modelling and Bert Bos, who was interviewed for this chapter, is a PhD degree in continuous-time system modelling.

Verhaert employs engineers with a wide range of backgrounds: product innovation, product development, conceptual design, mechanics, embedded electronics, embedded software and physics. Five engineers were involved in the development of the Crescendo model of the excavator: one with a physics background who had previous experience in university research on computer tomography algorithms; one with background in embedded electronics, embedded software and data processing; one with a software background (embedded, PC and mobile platforms); and two with background in electro-mechanical engineering. Koenraad Rombaut, who was interviewed for this chapter, had management rather than technical responsibilities in the creation of the co-model.

## 12.3 Traditional Development

We asked the engineers to describe how the case study systems would typically have been developed using the technology and processes then in place in their organisations. Two of the companies would traditionally have made use of physical prototypes. Koenraad Rombaut from Verhaert reported:

Initially, measurements would be made on a real-life excavator to determine the characteristics of the physical behaviour and the existing controller. Following this, a basic standalone model of the excavator would be built in Matlab/Simulink to be used for controller development. A downscaled test setup of the excavator would be built in a lab environment for tuning and verification of the main loop controller. The final production code for the controller would be based on code generated from the Matlab/Simulink control model.

Bert Bos from Chess reported a similar traditional approach:

Traditionally, the design effort would be smaller and lead to building and testing to be carried out on several physical prototypes. Fault handling is traditionally very difficult to test thoroughly, and errors appear during initial use. This can lead to required repairs after the product has been delivered, which increases overall cost.

At Neopost, software simulators would normally be used to fine-tune the software controllers of embedded systems. Peter van Eijk reports:

Without the co-simulation technology we would use pure discrete-event based simulators built using C++.

All three companies already use modelling to some extent in the development of embedded systems, and so the step onward to collaborative modelling and co-simulation would not require revolutionary change in their technology and process base.

## 12.4 Integrating Co-modelling and Co-simulation with Existing Processes

We asked the engineers to discuss how well the use of the co-modelling technology integrated with their development process. Koenraad reported the biggest change in practice, especially in the timing of the phases of system development:

Using the co-modelling technology forced our engineers to use a more structured approach to system development. We experienced a slight shift in timing and development approach compared to the traditional embedded systems development process. Fortunately, this did not conflict with the flexible development approach used at Verhaert. Detailed verification of the system shifted to an earlier stage of the development process by using the CT and DE models instead of physical hardware. The development of the software controller could also be started a lot earlier because of the availability of the CT model.

Engineers from Neopost and Chess adapted the methodology to their existing processes. Bert commented:

Chess traditionally follows a development process similar to IEEE 12.207: Initially the operational concept is defined, followed by defining the architecture before developing the full project. It is an iterative process where details are added over time. We have experienced no difficulties integrating this process with the co-modelling technology. After an initial idea phase (operational concept) a first model and physical prototype were developed to demonstrate the fidelity of the co-model. Collaborative modelling could be seen as part of the architecture phase and gives substantial and objective feedback on the architectural ideas. Normally this feedback does not arise until during the integration testing of the full system.

These comments suggest that it is possible to adapt co-modelling methodology to existing development processes without too many alterations. Shifts in the timing of some phases are bound to happen, since our method requires that more time is invested in early design and modelling to enable earlier verification ensuring rapid feedback on critical system properties.

## 12.5 Resources

We asked the technology users how resources were spent on developing the case studies. Peter van Eijk reported that Neopost had spent less than one full-time equivalent on developing the case study. He envisaged saving some time from the training effort in subsequent projects, but did not anticipate other major effort reductions.

The introduction of new methods and tools necessarily entails investment of time and effort, but second and subsequent uses of the same tool chain are likely to incur lower training costs. If the same tool-chain and methodology are reused in later projects, further training requirements will be minimal. Koenraad made similar observations:

Approximately 26 man-months of engineering effort were spent on developing the case study. We estimate that all tool training could be saved in subsequent case studies. In addition, we spent a lot of time testing intermediate versions of the tools and adjusting the models to be compatible with new tool features. This effort could also be saved in subsequent case studies using a stable tool-chain. On the other hand, this approach enabled us to have a first time right test setup with loop control.

The case studies reported here were conducted over the 3 years of a research project, with the participating companies allocating between two thirds and one full-time equivalent staff. During the project, the methods and tools for co-modelling and co-simulation were necessarily developed “from scratch”, in parallel with case studies. Consequently, a considerable proportion of the time was invested in evaluating intermediate versions of the tools. Adapting existing models to tool feature updates can be very time-consuming. Bert Bos from Chess reports a similar experience:

The models were built by two persons: one with experience in DE modelling and one with experience in CT modelling. A significant part of the resources were spent on evaluating the tool chain; to explore the capabilities of the tool; and to pin down errors in the tool. Different methodological approaches were also considered: how should the technology be embedded into larger projects; when should fault modelling be initiated; and how to gradually add details in an iterative manner. These considerations gave valuable input to the development of the methodological guidelines. In total, 23 man months were used on the modelling activities: 9 for model development; 9 for tool and methodology evaluation; and 5 for tool training and error finding.

With more mature and stable tools, all three companies agree that they would be able to allocate a much larger proportion of staff time to new case studies.

## 12.6 Challenges Encountered

We asked the engineers to identify the difficulties encountered during the development of the three case studies. Koenraad reported:

Our non-software engineers had no experience with the Eclipse platform on top of which the tool-chain is built. Also, VDM was unknown to our engineers which lead to steep learning curves with regards to both the language and the modelling philosophy. The main functionality missing from the tool-chain is code generation. This would make the final step towards physical verification on a test setup less labor intensive and remove the potential of errors introduced in the implementation of the final production code of the controller.

Using a model-based approach to embedded systems development requires a paradigm shift: engineers must be trained in using new tools and languages, and there are often steep learning curves. Bert experienced similar difficulties in gaining a common understanding of the semantics of co-simulation in Crescendo:

The main difficulty encountered was how to correctly scope the model: what should the model represent; and which parts of the model should be modelled in DE and which in CT? The timing model used in the tool is still understood by only a select few persons in the project. In particular how data exchange between CT and DE interacts with the differential equation solver within CT is not fully understood which leads to uncertainty during the model development.

Peter observed the effects of tool instability:

The main difficulties encountered had to do with the stability of the tool. In early phases of the project the tool changed a lot and there were frequent feature additions. This required us to adjust our model to be fully compatible with the tool. At later stages of the project more stable versions of the tool were released.

The difficulties reported were predominantly in early phases and were caused by the need to develop skill at using the tools, tool instability and the need to update models to stay in synch with new tool releases. At the time of writing, work on the provision of code generation has been initiated. The semantics of the co-simulation have been described in a clearer style, as demonstrated in Chap. 13.

## 12.7 Key Benefits

What were the benefits of using co-modelling and co-simulation? Peter reported:

We saved a lot of time by locating design error prior to creating a physical prototype. We estimate that an entire prototype production cycle was saved due to the early and rather complete analysis of the system.

Reducing the need for physical prototypes can be very beneficial: saving an entire prototype production cycle greatly reduces the overall cost of a product. Peter also identified the use of the DE model for testing the hand-coded software:

The VDM simulation model that was developed for the feeder folder has been manually translated to C++ and this code has been used to develop and pre-integrate the embedded control software before the feeder/folder hardware was available. Even when the hardware of the feeder folder was available the engineers continued to use this simulator for two reasons:

- First of all in the early stages of the project the simulator was working more reliably than the real system. This meant that by using the simulator instead of the real machine

the engineers could concentrate on the development of their software instead of keeping the mechanics running.

- A second reason was the availability of the simulator compared to the availability of the real machine. The real machine turned out to be under development constantly leading to a low availability for the software engineers. When during the project the number of embedded software engineers was raised from one to three and the fact that we had only one physical model the advantage of using the simulator became of crucial importance.

Koenraad similarly reported resource savings as a result of using computerised rather than physical prototypes:

Since a CT model of the physical dynamics of the excavator was created, a lot of time was saved on building physical prototypes. This ensures much faster iterations on physical models compared to traditional approaches. The Crescendo technology enabled us to easily swap between different design solutions (e.g. hydraulic vs. electrical drives). Such design space exploration would be extremely time consuming on real-life prototypes. The control software development could start from day 1 in parallel with the physical model development — traditionally the development of the control software would not be initiated before the physical test set-up had been completed. This ensured that several controller errors were found very early in the development process which saved development time.

Computerised models of the physical plant can be used to test embedded software at an early stage, avoiding some of the issues commonly encountered during an integration test. Bert told us that engineers from Chess used the co-model in a similar way:

Debugging in the co-simulation environment is much quicker than debugging real-time embedded control software. Debugging the final implementation of the system was supported by the model. A technical error in the sensor interface was found by comparing its output with what was expected from the model. After solving minor technical issues such as Hardware-Abstraction-Layer (HAL) sensor wiring and a driver problem, the initial system implementation worked the first time. This was an exceptional experience for us, since fault handling usually takes several cycles to work properly. Also, the functional behaviour of fault handling was evaluated in the co-model, and after implementation the system worked as foreseen. This ensured that integration overhead was kept to a minimum. The co-modelling and co-simulation technology allowed us to explore alternative designs quickly without having to do changes to physical prototypes. The collaborative model served as a means to communication between the system architect and the programmer developing the DE model.

It was notable that, even though this was the first time the engineers from the three companies used co-modelling approaches for embedded systems development, all experienced time savings. In all three case studies, the users felt that the additional time invested in earlier phases would be recovered in later stages.

This claim is given credibility by the thorough evaluation that was performed [97]. At the start of the DESTECs project, explicit tool and methodology requirements were formulated by the industrial partners. These requirements can be perceived as a statement to the lack of support of some feature or process, or, in other words, a deficiency that was explicitly identified which hampers product development as perceived by an end-user. They implicitly represent a benchmark of the state-of-the-art and state-of-practice at the time, based on similar experiences with other tools and techniques. Hence, when a requirement is realised, we can



conclude that we have made a step forward if its evaluation is also positive. The evaluation was performed at the end of the 3 year research project for all requirements that were implemented in Crescendo. This evaluation was based on concrete acceptance criteria that were formulated at the time when the requirement was established. Moreover, the industrial partners also rated the perceived impact on productivity. Out of the 25 implemented requirements that have been evaluated, no less than 21 reached the *improvement to productivity* threshold, and 12 of those also passed the *significant productivity improvement* mark. Overall, this implies that a 30 % increase in productivity is within reach when using Crescendo technology. One can argue that the data is statistically irrelevant with only three independent assessors, but we feel confident about this claim since the experiences from the challenges of the Industrial Follower Group<sup>1</sup> and the DESTTECS Summer School have also demonstrated that impressive results can be achieved.

## 12.8 The Future of Co-modelling

Peter, Bert and Koenraad identified several benefits and some drawbacks of the co-modelling methods and tools. Peter van Eijk reflected on the future of model- and simulation-based development of embedded systems in Neopost:

Simulation will become increasingly important for Neopost. Test setups and physical prototypes will become more costly and have less availability. The Crescendo technology can be one of the critical enablers for a more model-based approach to embedded systems development. The most critical current drawback of the Crescendo technology is the incompatibility with the tools we use for generating the code for the embedded platform. Code generation within Crescendo will mitigate this drawback.

Bert and Koenraad agreed on the potential benefits of code generation. Bert also offered an alternative view on the use of the technology:

The complexity of embedded systems is ever increasing with the increased focus on: error handling, safety demands, and added functionality. This complexity cannot be overseen by a single lead engineer or by a single engineering team. To avoid making expensive mistakes, modern embedded systems cannot be developed anymore without utilising modelling during project initiation and architecture phases. When following this sort of concurrent and holistic model-based development principles, a technology like Crescendo is a critical enabler.

Koenraad saw considerable potential in the approach:

We see great potential in the Crescendo technology during the development of complex systems combining some of the following: complex multi-physics (e.g. multiple interacting subsystems where behaviour is hard to predict); multiple controls (e.g. multiple interacting subsystems); multiple inputs (e.g. users, data inputs, supervisory controls, safety systems); high reliability and safety requirements and systems that are difficult to test due to high risk, cost, timing constraints.

---

<sup>1</sup>For more information, we refer to <http://www.destecs.org/deliverables.html>.

Koenraad also offered some ideas for Crescendo usability improvements:

Adding a GUI shell on top of Crescendo (similar to the way bond graphs are encapsulated in 20-sim using the iconic diagrams) could broaden the use of the tool to non-software engineers. Graphical tools for visual validation of the discrete behaviour of the Crescendo model would be beneficial as well: monitoring discrete values, state changes, etc. Finally, an increase in simulation speed would of course be a welcome improvement.

Chess have embarked on several new projects using the Crescendo technology:

The persons using the technology need some experience in abstracting from implementation details. Learning the modelling metaphor and the VDM syntax is a big hurdle as well. With that said, we are currently using the technology in the development of a taxi on-board computer, to validate the sensor system to determine the driven distance and to detect fraud in distance measuring. We also use the technology to evaluate state machines used in a payment system. Payment systems used in e-commerce follow various state flows. A model is under investigation to unify the states over several payment means. State sequences and timing are important here as, in practice, critical races occur in reporting the payment result.

With stable tools available and no need for training, it is anticipated that the added time investment in the early design phases will easily be saved in later implementation and integration phases.

## 12.9 Conclusion

Peter, Koenraad and Bert were pioneers: the first to apply Crescendo for co-modelling and co-simulation in an industrial environment. They had to contend with the instability of initial research versions of tools and only basic initial training. It is noticeable, however, that they retained a pioneer's clear view of the technology's potential to improve product development in their organisations. Their experience suggests that several factors might motivate the decision to use these methods and tools in commercial practice. These include the need for rapid early-stage innovation, the need for dependability and the cost of physical prototyping and testing.

Innovation in embedded systems is being driven by rapidly evolving capabilities of computing platforms and networks. This is leading to the invention of mechatronic applications in areas like robotics, medicine and transport that rely on novel and complex control software or in areas such as agriculture that may previously have seen little use of computing systems or software for control. Where innovation is rapid, it becomes imperative to eliminate infeasible or suboptimal designs quickly, and it would appear, for example, from experience with the ChessWay, that co-modelling is valuable as a means of gaining confidence in the appropriateness of a design direction without having to build physical prototypes of products that may be out of the ordinary for the organisation.

Increasingly, users come to rely on the correct functioning of complex embedded devices. The achievement of dependability demands both fault avoidance and fault tolerance. The experience in both Neopost and Chess suggested that avoiding

defects by early detection could save design iterations; there is also a pay-off in using early-stage models as an oracle or design aid for later-stage implementation tests. Modelling could be seen as a form of early virtual prototyping, reducing the risk of failing tests when expensive or dangerous physical prototypes have to be built. In Chap. 14, we examine ways in which the methods developed here could be extended to more complex products and cyber-physical systems.

# Chapter 13

## Semantics of Co-simulation

Joey W. Coleman, Kenneth Lausdahl, and Peter Gorm Larsen

### 13.1 Introduction

This chapter provides an overview of the semantics for the whole co-simulation framework, covering the co-simulation engine and the scripting language. It is presented in a combination of styles, including graphical representations and Structural Operational semantics (SOS), see [77, 78]. This chapter assumes a background in formal semantics and is intended primarily for readers seeking a full understanding of the semantics underlying the Crescendo technology.

The mathematical foundations for bond graphs and the VDM-RT notation are substantially different. In this chapter, we present the formal foundations that enable coupling of these diverse formalisms to permit co-simulation. The Crescendo tool implements the semantics presented here following a traditional master/slave architecture. The master is the co-simulation engine that manages all communication between the DE and CT simulators which act as slaves and progress the simulation according to the directives coming from the master co-simulation engine.

The overall structure of the co-simulation engine, a basic explanation of the synchronisation of the two “slave” simulators, and the semantic constraints for the simulators are presented in Sect. 13.2. Section 13.3 provides the actual co-simulation semantics. Then, Sects. 13.4 and 13.5 explain how the co-simulation can be extended using the semantics for a small language Crescendo Scripting Language (CSL) that enables scenarios to be exercised by a co-simulation. Section 13.6 ends the chapter with a short summary.

---

J.W. Coleman (✉) • K. Lausdahl • P.G. Larsen  
Aarhus University, Aarhus, Denmark  
e-mail: [jwc@eng.au.dk](mailto:jwc@eng.au.dk); [lausdahl@eng.au.dk](mailto:lausdahl@eng.au.dk); [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

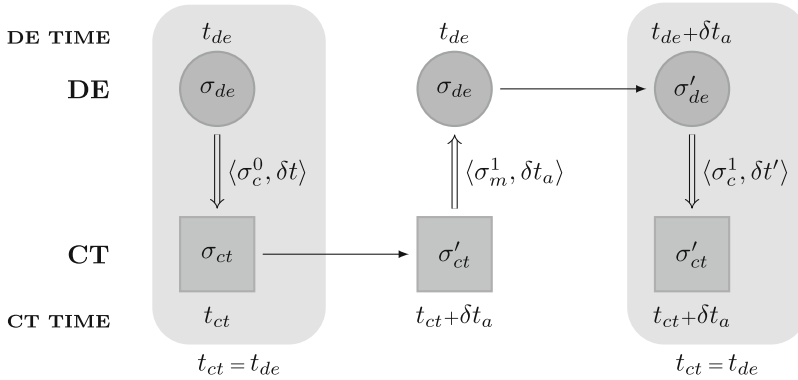


Fig. 13.1 Example of the synchronisation scheme for DE-CT co-simulation

## 13.2 Structure of Co-simulation

Figure 13.1 illustrates the synchronisation scheme underlying co-simulation between a DE simulation of a controller (top) and a CT simulation of the plant (bottom). It is an expanded diagram of Fig 2.3. The DE and CT simulators are coupled through a co-simulation engine that explicitly synchronises the shared variables, events and the simulation time in both linked simulators (the co-simulation engine is not shown explicitly in Fig. 13.1).

Each simulation maintains its own local state and time at which the state is valid. Thus, let  $\sigma_{de}$  be the internal state of the DE simulation at simulation time  $t_{de}$ , and let  $\sigma_{ct}$  be the internal state of the CT simulation at simulation time  $t_{ct}$ . The controlled variables defined in the co-simulation contract (whose values are defined in  $\sigma_c$ ) are set by the DE controller and read by the plant; the monitored variables ( $\sigma_m$ ) are set by the plant model and read by the controller.

Consider a synchronisation cycle which starts with the two simulators having a common simulation time ( $t_{ct} = t_{de}$ ). On each cycle, the DE controller simulation sets the controlled variables and proposes a duration  $\delta t$  by which the CT simulation should, if possible, advance. As the CT simulation of the environment advances, it may encounter a state in which one of the event predicates defined in the contract becomes true. The state of the monitored variables  $\sigma_m^1$  and the actual time that it reached  $\delta t_a$  are communicated back to the DE side. If no events occur in the CT simulation during this interval,  $\delta t_a = \delta t$ . While the CT simulation has been progressing, the DE simulation remains unchanged, so its local simulation time remains at  $t_{de}$  and state  $\sigma_{de}$ . The DE simulation then advances by  $\delta t_a$  so that both DE and CT are again synchronised at the same simulation time, and the controlled variables are updated ( $\sigma_c^1$ ) and the next time step is proposed to CT. The performance of the DE state change takes place in two stages, with the calculations being performed first, separately from advancing the DE simulation time. The granularity of the synchronisation time step is always determined by

the DE simulator. The scheme does not require resource-intensive roll-back of the simulation state in either of the simulators, though roll-back may occur inside the CT simulator in order to catch the precise time requested. The CT simulator typically performs a binary search technique to determine zero crossings for event signalling. This process is called microstepping, which is even performed when fixed time step solvers are used.

The overall structure of the co-simulation semantics presented in the remainder of this chapter consists of a central semantic model of the co-simulation engine. It consists of the necessary semantic models for the individual pieces of a co-model. This means that we have three semantic models instantiated for a typical co-model: the co-simulation engine; the Discrete-Event (DE) semantic model of the formalism (VDM) used to describe an embedded digital controller; and the Continuous-Time (CT) semantic model of the formalism (bond graphs) that is used to describe the environment that the embedded controller interacts with. The semantic model of the co-simulation engine is necessarily generic: it needs to be able to support the opaque embedding of other heterogeneous semantic models. In addition, the semantic models of the two simulators are only characterised in this chapter, and we rely on opaque embeddings of them in the semantics of the co-simulation engine.

The semantics of the co-simulation engine can only embed semantic models that conform to certain constraints. Some of these constraints are common to both DE and CT models and these are described in Sect. 13.2.1. The constraints of the CT and DE simulator semantic models are explained in Sects. 13.2.2 and 13.2.3, respectively.

This structure consisting of a central engine coordinating the simulated time and shared state of the overall co-model provides a clear modular division between the generic co-simulation properties and the formalism-specific subject model needs. As a part of this, the actual subject model simulations are delegated to separate semantic models, allowing them to be fit for the needs of those subject models. This, in turn, enables an implementation of the co-simulation engine that allows the use of a variety of different specific simulation engines, as has been done in the Crescendo tool.

### ***13.2.1 Common Semantic Constraints***

There are a few common constraints on the semantics of the simulators involved in co-simulation that must be respected. The constraints common to both simulator semantics are as follows:

- C1:** It must be possible to have the semantics “step” from a given state at a given time to a successor state at some future time, and it must be able to do so in relatively small time increments.

**C2:** It must calculate the next state of the subject model in a given simulator semantics, taking into account the values from the shared state that have been changed by other simulator semantics.

The first constraint, C1, is required to ensure that it is possible to interleave the simulation of the subject models. The ambiguity that it is able to do so in relatively small increments relates to the subject models (in the Crescendo tool the scale is nanoseconds).

For CT simulator semantics, this effectively means that it must be possible to determine the observable state of a model for any arbitrary (future) point in time, given the granularity defined by the “small time increments” noted above.

Constraint C2 is required to support a shared state between the various subject models in the co-simulation engine. Without this, one may as well just run the individual subject models independently.

### ***13.2.2 Continuous-Time Simulation Semantics***

The first type of semantic description that may be used as a simulator semantics in the co-simulation framework is labelled “continuous-time”. Semantic descriptions of this type are characterised by a model that is based on a single model state with no hidden or transactional variables. As such, simulators with this sort of semantic model are typically used to model physical systems.

Given the basic assumption that any semantics model in the overall co-simulation framework must make finite steps, there are four primary requirements on a CT semantic model:

- CT1:** It must be possible to observe the complete state of the semantics for a given point in time.
- CT2:** It must be possible to set bounds on the maximum duration of each simulation step of a subject model in the semantics.
- CT3:** It must always produce the actual duration of the simulation step that was taken.
- CT4:** It must be able to produce “events” as part of a simulation step of a subject model.

The first constraint, CT1, means that there must be no hidden state: the successor state of a CT semantics must be dependent only on the observable state and the time. This lack of pre-determination, in turn, in principle allows changes to shared state variables to have immediate effect on the simulated model.

The second constraint, CT2, is necessary to support the interleaving of CT and DE semantic models. Because it is possible for a DE model to determine that its observable state will change at some given point in the future, we then know that we will need to synchronise the simulator semantic models at that point. Thus, it must be possible to ensure that a step of the semantic model will be no later than that known synchronisation point.

The third constraint, CT3, is required as it may be possible for the step to have taken less time than the maximum allowed by the bound. It may happen that the subject model's successor state in a step is earlier than that bound; this can represent something occurring in that subject model, altering shared state, that requires synchronisation with the other simulation semantics.

Constraint CT4 allows for a mechanism to indicate that something of interest has happened in the simulation of the subject model—an event—without the need to record the event directly in the shared state. This is a sort of side-band signalling mechanism enabling both time-triggered as well as event-triggered behaviour.

### ***13.2.3 Discrete-Event Simulation Semantics***

The second type of semantic model for simulation is called the “discrete-event” model. Semantic descriptions of this type are characterised by a model that allows portions of the overall subject model state to be hidden. These hidden portions of the overall model state may be the result of determining the future value of state elements, but not allowing them to be observed until the appropriate point in time. Not surprisingly, this sort of model is typically used to model computational systems, such as embedded controllers.

The requirements on a DE semantic model are concerned with managing the subject model's state and the bounds on the duration of steps of other simulators.

- DE1:** Every step must indicate the time at which hidden state will next be revealed.
- DE2:** It must be possible to have the semantics perform a synchronisation step that only exposes hidden state that is ready to be revealed up to a given point in time and updates the internal subject model state with the shared state from the co-simulation engine.
- DE3:** It must be able to update to a point in time prior to the next point at which hidden state will be revealed.
- DE4:** It should accept the notification of events that occurred in other simulators.

The first constraint, DE1, on DE semantic models is necessary to indicate to the co-simulation engine the point at which there will next be a change to the shared state due to this subject model. It allows the co-simulation engine to set a bound on the maximum duration of the next step of CT simulation semantics and is necessary to keep the various simulator semantics in the co-simulation engine in step with respect to time.

The second constraint, DE2, anticipates the addition of a fault injection framework and is not strictly necessary for the co-simulation engine itself. In a CT semantic model, as it has no hidden elements, the control and value state of the subject model are always synchronised with the time reported to the co-simulation engine. The DE model, in contrast, may allow hidden elements in its internal state to contain values which are not valid until future points in time. This means that for



the subject model to be affected by changes to the shared state from other subject models, it must have a mechanism to update the present internal state before the next internal state is calculated. The initial semantic rules presented in Sect. 13.3.3 do not entirely conform to constraint DE2; however, Sect. 13.4 depends upon the constraint and alters the semantic rules in the required manner.

The third constraint, DE3, allows for this simulation semantics to gracefully handle the other simulation semantics taking steps shorter than expected. The expectation is that a DE model would not normally change any state on such a step. Connected to this is constraint DE4, which requires that the simulation semantics accept the notification of events from the CT model; these events give the subject model an opportunity to react to their presence.

### 13.3 Co-simulation Semantics

We use the SOS format [77, 78] to present the semantic definitions in this chapter. An SOS description consists of two major elements: a set of type definitions that describe the static structure of the system; and the definitions of the transition relations that describe the behaviour of the system.

The semantics is described using a collection of *transition rules*. Each of these has a number of hypotheses (typically one per line) over a horizontal line. Below the line the conclusion can be reached if all the hypotheses above the line can be reached. It is also possible that such transition rules have side conditions that also need to be satisfied to ensure the validity of applying a specific transition rule. Many of the hypotheses and the conclusion are typically described as transitions which are indicated as something that matches a configuration followed by a line (from left to right with a name above it) and followed again by the new configuration that a system is transformed into.

#### 13.3.1 Structural Operational Semantics

The logical notation used for the assertions in the semantic descriptions is the basic VDM-SL type system and expressions [20, 27].<sup>1</sup> This notation is used to define the static structure of the co-simulation engine and give its behaviour.

In an SOS definition, the static state of a system is modelled as a *configuration* that contains all of the information needed to capture the complete state of a system at any given point. This includes, for the co-simulation semantics we present, information about which simulator semantics was used for the previous step, the

---

<sup>1</sup>The mathematical syntax will be used for these definitions in order to clearly distinguish it from the use of VDM-RT in the models used in this book.

shared variable state, the current simulated time and the complete internal states of both simulators' semantics (treated opaquely). Configurations are typically given as tuples.

The behaviour of a system is defined through the use of *transition relations*, at least one of which must involve the complete static state configuration. In a fine-grained SOS definition, the overall system behaviour is typically defined using a transition relation from configuration to configuration, though this is not strictly necessary.

The transition relations are defined through the use of inference rule schemata, where each rule's conclusion defines a subset of the entire transition relation. The least relation that satisfies all of the inference rules is taken to be the relation defined.

Consider the following rule:

$$\text{Example} \frac{P(a, b, a', b') \quad Q(a, b) \quad R(b')}{(a, b) \xrightarrow{s} (a', b')}$$

The Example rule here would be a (partial) definition of the  $\xrightarrow{s}$  transition relation. The conclusion below the line has four free variables— $a, b, a'$  and  $b'$ —into which values may be substituted. For this rule to be true of a given pair of pairs— $(a, b)$  and  $(a', b')$  in this case—then the three hypotheses above the line must be satisfied.

In general, the rule applies for any set of values that may be substituted into the free variables so long as all of the hypotheses of the rule are satisfied. This substitution is similar to the pattern matching as done in VDM models and, where a free variable is equated to some other structure, the resolution of the possible values for the free variable is done in a way that allows the constraints in the other structure (concrete values, restrictions to certain types and so on) to hold of the value in the substitution. In practice, however, the rules are often used to determine a successor configuration, that is, the configuration on the right of the transition relation, based on some given predecessor configuration.

For the Example rule, that means that we would start with known values of  $a$  and  $b$  and then proceed to determine values for  $a'$  and  $b'$  that satisfy  $P(a, b, a', b')$  and  $R(b')$ . However, we would first check that this rule applies to the given  $a$  and  $b$  by checking any hypotheses that apply only to  $a$  and  $b$ ; here we check  $Q(a, b)$ .

The task of finding values for the successor configuration,  $a'$  and  $b'$  in this case, may be one of simply calculation, or a more complex constraint satisfaction problem. It may also involve choosing between equally valid alternatives for some values—these cases can introduce non-determinism into the semantic model,<sup>2</sup>

---

<sup>2</sup>Or under-definedness, rather than non-determinism, depending on the perspective and interpretation of the semantic model.

which can be useful but must be resolved during implementation. Finally, it may not be possible to find values for a successor configuration: this may indicate that the predecessor configuration is not a part of the system modelled; that the wrong rule is being applied; that there's an error or missing rule in the semantic model; or that the semantic model has deliberately chosen that the predecessor configuration has no successor.

### 13.3.2 *Co-simulation Static State*

Given a co-simulation system that has a CT-type simulator and a DE-type simulator, the overall co-simulation system configuration is given as

$$Config = DE \times CT \times \Sigma \times Time \times Time \times Event\text{-set} \times Tag \quad (13.1)$$

where

- *DE* is the type of representations of the DE-type simulator, covering all of the possible states that it may reach;
- *CT* is the type of representations of the CT-type simulator, also covering all of the states that it may reach;
- $\Sigma$  is the representation of the variables shared between the two simulator semantic models and is defined below;
- the first *Time* component of the tuple represents the current simulated time of the overall co-simulation system;
- the second *Time* component represents a time bound that must be respected by CT simulator semantics;
- *Event-set* is the set of *Events* generated by CT semantic models for the DE models to react to; and
- *Tag* is a token that records which of the two semantic models took the last step.

The definitions of the *DE* and *CT* representations are left undefined here; it suffices that they conform to the constraints given in Sect. 13.2. These representations are the overall configurations from their respective semantic models.

The type of the shared variables is a map from an identifier (represented by the inexhaustible set  $Id_v$ ) to a pair of a value and a tag indicating which simulator semantics controls the value of that variable.

$$\Sigma = Id_v \xrightarrow{m} (Value \times Tag) \quad (13.2)$$

Note that a shared variable ownership is immutable over the course of a co-simulation run. Changing the ownership of a shared variable would imply some significant change to the structure of the subject models in the co-simulation.

The tag is simply a structure-less token, either  $\langle DE \rangle$  or  $\langle CT \rangle$ .

$$\begin{array}{c}
\text{DE Step} \left[ \frac{(de, \sigma, \tau, events) \xrightarrow{de} (de', \sigma', \tau'_b)}{\sigma'' = mergeStates(\sigma, \sigma', \langle DE \rangle)} \right. \\
\left. \frac{(de, ct, \sigma, \tau, \tau_b, events, \langle CT \rangle) \xrightarrow{cs} (de', ct, \sigma'', \tau, \tau'_b, events, \langle DE \rangle)}{} \right. \\
\\
\text{CT Step} \left[ \frac{(ct, \sigma, \tau_b) \xrightarrow{ct} (ct', \sigma', \tau', events')}{\sigma'' = mergeStates(\sigma, \sigma', \langle CT \rangle)} \right. \\
\left. \frac{(de, ct, \sigma, \tau, \tau_b, events, \langle DE \rangle) \xrightarrow{cs} (de, ct', \sigma'', \tau', \tau_b, events', \langle CT \rangle)}{} \right]
\end{array}$$

Fig. 13.2 Inference rules for the behaviour of the co-simulation engine

$$Tag = \langle DE \rangle \mid \langle CT \rangle \quad (13.3)$$

### 13.3.3 Co-simulation Behaviour

With the static state defined, we can now define the transition relation for the co-simulation semantics,  $\xrightarrow{cs}$ .

$$\xrightarrow{cs} : Config \times Config \quad (13.4)$$

Note that the colon, above, is a type assertion;  $\xrightarrow{cs}$  is a relation over the type of configurations, *Config*.

We give two inference rules to define the high-level behaviour of the co-simulation semantics in Fig. 13.2. The first rule, DE Step, describes the behaviour that is dependent on the semantics of the discrete-event simulator; the second rule, CT Step, handles the corresponding case when the behaviour is dependent on the semantics of the continuous-time simulator.

Starting with the DE Step rule, consider first the *Config* tuple on the left of the  $\xrightarrow{cs}$  relation. This tuple is given in terms of its components to allow us to name the components for use in the rule.

Moving our focus to the first hypothesis of the rule, we see some of the components of the *Config* tuple being used in a new transition relation,  $\xrightarrow{de}$ . This transition relation is defined as

$$\xrightarrow{de} : (DE \times \Sigma \times Time \times Event\text{-set}) \times (DE \times \Sigma \times Time) \quad (13.5)$$

and it represents a single step in the discrete-event semantics. Informally, the left side of the  $\xrightarrow{de}$  transition is a tuple of a DE simulator state, shared variable state, a new simulated time for the DE to update to and a set of events that happened between the last point at which the DE semantics took a step and the new simulated time; the right side is a tuple of the new DE simulator state, the new values of shared variables and a new time bound.

$$\begin{aligned} \text{mergeStates}: \Sigma \times \Sigma \times \text{Tag} &\rightarrow \Sigma \\ \text{mergeStates}(\sigma_o, \sigma_i, \text{tag}) &== \sigma_o \dagger \{id \mapsto \sigma_i(id) \mid id \in \text{dom } \sigma_i \wedge \sigma_o(id).\#2 = \text{tag}\} \end{aligned}$$

**Fig. 13.3** Function to merge two shared states, taking only the values paired with a specific tag value

The second hypothesis merges the new values in the shared variable mapping what were produced by the DE semantics into the overall co-simulation shared state. The details of the merge are encapsulated into the *mergeStates* function, defined in Fig. 13.3. This hypothesis is separate from the DE semantics to emphasise that this synchronisation is, properly, the responsibility of the overall co-simulation semantics and not the individual simulator semantics. One important property that is maintained by this separation is that each simulation semantics may only affect the values that it “owns”, as marked by the control tag in the state.

So, the DE Step rule defines the behaviour of the co-simulation semantics when executing a DE step directly in terms of the behaviour of the DE semantics, with a bit of extra mechanism to ensure that the overall shared state is updated.

The second rule in Fig. 13.2, CT Step, is structured in a similar manner to the first, but uses the  $\xrightarrow{ct}$  transition relation to model the behaviour of the CT semantics as it performs a single step. This transition relation is defined as

$$\xrightarrow{ct}: (CT \times \Sigma \times \text{Time}) \times (CT \times \Sigma \times \text{Time} \times \text{Event-set}) \quad (13.6)$$

where, informally, the left side of the transition relation is the CT simulator state, the shared variable state and a time bound; and the right side of the transition relation is the new CT simulator state, the modified shared variables, the time up to which the CT semantics actually reached and a (possibly empty) set of events generated during that step.

The primary advantage of this semantic description of co-simulation is that it isolates the semantics of the two simulators as much as possible, allowing the semantics for each simulator to be defined independently.

### 13.3.4 Simulator Properties and Their Transition Relations

For the common constraints, C1 and C2, it is clear that both the  $\xrightarrow{de}$  and  $\xrightarrow{ct}$  transition relations are satisfactory. The ability of the semantic models to step from state to state is inherent in SOS descriptions, thus satisfying C1. Satisfaction of C2 is actually handled internally within the individual definitions of  $\xrightarrow{de}$  and  $\xrightarrow{ct}$  that we must be satisfied with the simple structural compliance present. Specifically, the shared co-simulation state is in the signature of the transition relation, allowing a definition of the transition to use and update it.

Focusing on  $\xrightarrow{ct}$  transition, it must satisfy the four CT constraints, CT1–CT4. Two, CT1 and CT3, are dependent on the internal definition of the  $\xrightarrow{ct}$  transition, and the external structure cannot speak to this. However, CT2 requires that the internal definition of  $\xrightarrow{ct}$  be aware of a maximum duration and this appears structurally in the left-hand configuration. Finally, CT4 requires that the semantic description be allowed to produce events, and this appears directly in the structure of the right-hand configuration.

For the  $\xrightarrow{de}$  transition, the four DE constraints, DE1–DE4, must be satisfied. Constraints DE1, DE3 and DE4 are satisfied as far as possible by the structure of the signature of  $\xrightarrow{de}$ ; what remains is the responsibility of the internal definition of  $\xrightarrow{de}$  to handle. For constraint DE2, the signature of  $\xrightarrow{de}$  can support this, as it is possible to indicate that a synchronisation-only step is required by defining an event with such a special meaning. Beyond the structure, again, the rest is handled by the internal definition of  $\xrightarrow{de}$ .

Possible internal definitions for the  $\xrightarrow{de}$  transition may be found in [61], and a description for the  $\xrightarrow{ct}$  may be found in [24]. In this book, the  $\xrightarrow{de}$  transition corresponds to the semantics of the executable subset of VDM-RT while the  $\xrightarrow{ct}$  transition corresponds to the simulation of the bond graphs.

## 13.4 Adding Fault Injection Semantics to the Co-simulation

The co-simulation engine used in the Crescendo tool presents an opportunity to create a fault injection framework that is independent of any of the constituent simulators. This framework only has access to variables that have been explicitly shared between the simulation engines, but this is sufficient for the creation of fault injection and external event scripting.

The CSL is introduced in the Crescendo user documentation and is a minimal language intended to allow for changes to the shared state of the simulators when certain conditions are met. The basic structure of a script—a series of commands in CSL—is just a sequence of triggers, each with a firing condition, body and optional reversions.

The CSL is essentially a reactive language, and a script is a sequence of triggers. A trigger consists of a test expression and a set of assignments to the shared state and may also optionally have a list of variables to revert to their values prior to the trigger. A trigger may be set to react every time its test expression is true, or it may react only once during the entire co-simulation run.

In the basic co-simulation cycle, the DE simulator runs first, followed by the CT simulator, and execution continues, alternating between the two simulators. To add CSL into this cycle, we need to split the DE step and add a new CSL step. We must also extend the configuration, *Config*, with a representation of the CSL script, and the *Tag* type of tokens with  $\langle \text{CSL} \rangle$  and  $\langle \text{SYNC} \rangle$  to include tags for the CSL.

$$Config = DE \times CT \times CSL \times \Sigma \times Time \times Time \times Event\text{-}set \times Tag$$

$$\begin{array}{l}
 \text{DE Step} \left[ \begin{array}{l}
 (de, \sigma_o, \tau, events) \xrightarrow{de} (de', \sigma_s, \tau'_b) \\
 \sigma'_o = mergeStates(\sigma_o, \sigma_s, \langle DE \rangle) \\
 \tau''_b = \min(\tau_b, \tau'_b) \\
 \hline
 (de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle CSL \rangle) \xrightarrow{cs} (de', ct, csl, \sigma'_o, \tau, \tau''_b, events, \langle DE \rangle)
 \end{array} \right. \\
 \\
 \text{DE Sync} \left[ \begin{array}{l}
 (de, \sigma_o, \tau, \{ \langle SYNC \rangle \}) \xrightarrow{de} (de', \sigma_s, \tau'_b) \\
 \sigma'_o = mergeStates(\sigma_o, \sigma_s, \langle DE \rangle) \\
 \tau''_b = \min(\tau_b, \tau'_b) \\
 \hline
 (de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle CT \rangle) \xrightarrow{cs} (de', ct, csl, \sigma'_o, \tau, \tau''_b, events, \langle SYNC \rangle)
 \end{array} \right. \\
 \\
 \text{CT Step} \left[ \begin{array}{l}
 (ct, \sigma_o, \tau_b) \xrightarrow{ct} (ct', \sigma_s, \tau', events') \\
 \sigma'_o = mergeStates(\sigma_o, \sigma_s, \langle CT \rangle) \\
 \hline
 (de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle DE \rangle) \xrightarrow{cs} (de, ct', csl, \sigma'_o, \tau', \tau_b, events', \langle CT \rangle)
 \end{array} \right. \\
 \\
 \text{CSL Step} \left[ \begin{array}{l}
 (csl, \sigma_o, \tau) \xrightarrow{csl} (csl', \sigma'_o, \tau'_b) \\
 \tau''_b = \min(\tau_b, \tau'_b) \\
 \hline
 (de, ct, csl, \sigma_o, \tau, \tau_b, events, \langle SYNC \rangle) \xrightarrow{cs} (de, ct, csl', \sigma'_o, \tau, \tau''_b, events, \langle CSL \rangle)
 \end{array} \right.
 \end{array}$$

**Fig. 13.4** New semantic rules for the behaviour of the simulators in the co-simulation semantics

When splitting the DE Step rule from Fig. 13.2, we replace it with the version in Fig. 13.4. The difference between the two rules is that the extended version requires that the *Tag* in the left-hand configuration be  $\langle CSL \rangle$  instead of  $\langle CT \rangle$  and that the new time bound,  $\tau''_b$ , must be the least between that generated by the DE semantics, in  $\tau'_b$ , and last execution of the CSL script, in  $\tau_b$ .

We also add a new rule, DE Sync, that does a  $\xrightarrow{de}$  transition with a SYNC event marked; the intent is to signal to the internal definition of the  $\xrightarrow{de}$  transition that it should perform synchronisation of hidden state, and not perform any new computation, as per Constraint DE2. This rule also takes the least available time bound for  $\tau''_b$ .

For completeness, Fig. 13.4 also shows the updated CT Step rule and definition of the updated *Config*.

The transition,  $\xrightarrow{csl}$ , for the CSL can be defined as

$$\xrightarrow{csl} : (CSL \times \Sigma \times Time) \times (CSL \times \Sigma \times Time) \quad (13.7)$$

where *CSL* is the type representing CSL scripts. The two *Time* values correspond, respectively, to the simulated time in the subject model at which the CSL script is evaluated and a time bound indicating the next simulated time at which the CSL may be triggered.

The necessary semantic rule to use the  $\xrightarrow{csl}$  transition for the CSL is the last rule given in Fig. 13.4. This rule follows the pattern of the rest of the rules in Fig. 13.4, except that it does not need to merge states as it operates on the shared state directly. Further, given that the CSL is intended to allow changes to the shared state of both simulators, there is no need to handle ownership of variables.

Taking the rules in Fig. 13.4 as a whole, we can see that the round-robin cycle is preserved; in order, the rules proceed from the DE Step rule to CT Step to DE Sync to CSL Step and then back around.

## 13.5 Semantics of the CSL

The intended execution of a CSL script is straightforward: for a setpoint in time, each trigger is sequentially processed, modifying the shared state as necessary. The semantic model given below reflects this as the transition relation recursively processes the sequence of triggers, producing a new shared state when given an empty sequence.

### 13.5.1 Top-Level CSL Structures

The active structure of the CSL in the co-simulation semantics is the *CSL* construct.

$$\begin{aligned} \text{CSL} :: & \quad ts : \text{Script} \\ & \quad ms : \text{Marker}^* \end{aligned}$$

The *CSL* construct contains the script of CSL triggers and a sequence of the same length to hold a minimal amount of state for each trigger.

$$\text{Script} = \text{Trigger}^*$$

Each trigger state—a *Marker*—is either **nil**, a constant DONE or a pair of a time value and a state.

$$\text{Marker} = [\text{DONE} \mid \text{Time} \times \Sigma]$$

The initialisation of the *CSL* construct is handled by the  $\xrightarrow{cslinit}$  transition,

$$\xrightarrow{cslinit} : \text{Script} \times \text{CSL} \tag{13.8}$$

with its definition given in Fig. 13.5. The initialisation rule simply transforms a fault injection script into the *CSL* object used by the main CSL rules. The top-level rule is given in Fig. 13.6

The *Trigger* construct is defined as



**Fig. 13.5** CSL initialisation rule

$$\boxed{\text{CSL Init}} \frac{\begin{array}{l} \text{elems markers} = \{\text{nil}\} \\ \text{len markers} = \text{len script} \end{array}}{\text{script} \xrightarrow{\text{cslinit}} \text{mk-CSL}(\text{script}, \text{markers})}$$

**Fig. 13.6** The top-level semantic rule for CSL execution

$$\boxed{\text{CSL Top}} \frac{\begin{array}{l} \sigma = \{id \mapsto \text{value} \mid \sigma_o(id) = (\text{tag}, \text{value})\} \\ (ts, ms, \tau, \sigma) \xrightarrow{\text{trig}} (ms', \sigma') \\ \sigma'_o = \{id \mapsto (\sigma_o(id), \text{value}) \mid \sigma'(id) = \text{value}\} \\ \tau_b = \text{calculateCSLTimeBound}(ts, ms') \end{array}}{(mk\text{-CSL}(ts, ms), \sigma_o, \tau) \xrightarrow{\text{csl}} (mk\text{-CSL}(ts, ms'), \sigma'_o, \tau_b)}$$

*Trigger* :: *once* :  $\mathbb{B}$   
*test* : *Expr*  
*dur* :  $[Time]$   
*body* : *Stmt*\*  
*rev* : *Id<sub>v</sub>-set*

A *Trigger* has a flag, *once*, indicating whether or not it may occur only once or many times; a condition, *test*, which is evaluated to determine if the trigger should fire; an optional duration field, *dur*, which if present requires that the condition hold over the given duration; a body field, *body*, that holds assignment and print statements to be executed when the trigger fires; and a field, *rev*, that stores a set of variable names to be reverted to their values prior to the trigger firing.

The basic execution of a CSL script is modelled by the  $\xrightarrow{\text{csl}}$  relation,

$$\xrightarrow{\text{csl}}: (\text{CSL} \times \Sigma_o \times \text{Time}) \times (\text{CSL} \times \Sigma_o \times \text{Time})$$

and that rule delegates most of the actual work of executing the triggers to the  $\xrightarrow{\text{trig}}$  transition relation. Of the rest of the rule, the first and third hypotheses deal with the flattening and rebuilding of the shared state, and the final hypothesis calculates a time bound for the co-simulation based on the triggers that are actively testing a condition over a duration.

The  $\xrightarrow{\text{trig}}$  transition relation,

$$\xrightarrow{\text{trig}}: (\text{Trigger}^* \times \text{Marker}^* \times \text{Time} \times \Sigma) \times (\text{Marker}^* \times \Sigma)$$

only alters the sequence of *Markers* and the shared state; the actual triggers are never altered. The three basic rules for a trigger with a simple test (no duration) are shown in Figs. 13.7, 13.8 and 13.9.

Reading the *Exec* rule in Fig. 13.7, we see that the configuration is structured to pattern match the heads of the sequences of triggers and markers, ensuring that we only execute a trigger that is not already active. Then, the first hypothesis pattern matches the first trigger on the sequence to ensure that it is one without a duration. The second hypothesis ensures that the trigger's condition has been met; evaluation of the expression *test* is done using the semantic evaluation function  $\llbracket \cdot \rrbracket$ , and the result is, in turn, given a  $\tau$  and a  $\sigma$  to evaluate the expression in the time and state context down to its value. The third hypothesis delegates the execution of the

**Fig. 13.7** The Exec semantic rule for executing triggers

$$\begin{array}{l}
 \text{trigger} = \text{mk-Trigger}(\text{once}, \text{test}, \mathbf{nil}, \text{body}, \text{rev}) \\
 \llbracket \text{test} \rrbracket \tau \sigma = \mathbf{true} \\
 (\text{body}, \sigma) \xrightarrow{\text{tstm}} \sigma' \\
 \hline
 \text{Exec} \quad \frac{(ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'')}{([\text{trigger}] \frown ts, [\mathbf{nil}] \frown ms, \tau, \sigma) \xrightarrow{\text{trig}} ([(\tau, \sigma)] \frown ms', \sigma'')}
 \end{array}$$

**Fig. 13.8** The Skip semantic rule for executing triggers

$$\begin{array}{l}
 \text{trigger} = \text{mk-Trigger}(\text{once}, \text{test}, \text{dur}, \text{body}, \text{rev}) \\
 \llbracket \text{test} \rrbracket \tau \sigma = \mathbf{true} \\
 \text{dur} \in \text{Time} \Rightarrow \tau - \tau_0 > \text{dur} \\
 (ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'') \\
 \hline
 \text{Skip} \quad \frac{(ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'')}{([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{\text{trig}} ([(\tau_0, \sigma_0)] \frown ms', \sigma'')}
 \end{array}$$

**Fig. 13.9** The After semantic rule for executing triggers

$$\begin{array}{l}
 \text{trigger} = \text{mk-Trigger}(\mathbf{false}, \text{test}, \text{dur}, \text{body}, \text{rev}) \\
 \llbracket \text{test} \rrbracket \tau \sigma = \mathbf{false} \\
 \text{dur} \in \text{Time} \Rightarrow \tau - \tau_0 > \text{dur} \\
 \sigma' = \sigma \dagger (\text{rev} \triangleleft \sigma_0) \\
 (ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'') \\
 \hline
 \text{After} \quad \frac{(ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'')}{([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{\text{trig}} ([\mathbf{nil}] \frown ms', \sigma'')}
 \end{array}$$

body of the trigger to the  $\xrightarrow{\text{tstm}}$  transition relation, producing a potentially altered shared state. The last hypothesis executes the rest of the sequence of triggers.<sup>3</sup> The conclusion of this rule places a pair containing the present time—that is, when the trigger fired—and the initial state at the head of the sequence of markers in the resulting configuration. This indicates to future evaluations of this trigger that the trigger has fired and is waiting for the condition to become false.

The Skip rule in Fig. 13.8 just skips over triggers that have a pair as their marker and a true condition; for triggers with durations, the rule only applies when the duration has been exceeded (third hypothesis). When the marker is a pair, this indicates that the trigger has been active; furthermore, that the condition is still true means that it is not yet time to revert any variables.

The After rule in Fig. 13.9 handles the case where the trigger's condition has become false and there are shared variables to revert. This rule applies to all repeating triggers, regardless of whether or not they have a duration. The first two hypotheses pattern match the trigger at the head of the sequence and that the condition is false. The third hypothesis is a guard for triggers with a duration, to ensure that we do not revert shared variables when we should instead be cancelling a trigger with a duration that was still waiting for the necessary time to pass. The fourth hypothesis does the necessary reversion of the state, pulling the values of the variables to be reverted from the state at the start of the trigger,  $\sigma_0$ ; note that if the trigger did not have any reverts specified, then the set  $\text{rev}$  will be empty and

<sup>3</sup>The base case—an empty sequence of triggers—is handled by the rule **Base** in Fig. 13.10.

$$\begin{array}{c}
\boxed{\text{Base}} \frac{}{([], ms, \tau, \sigma) \xrightarrow{\text{trig}} (ms, \sigma)} \\
\\
\boxed{\text{Done}} \frac{(ts, ms, \tau, \sigma) \xrightarrow{\text{trig}} (ms', \sigma')}{([\text{trigger}] \frown ts, [\text{DONE}] \frown ms, \tau, \sigma) \xrightarrow{\text{trig}} ([\text{DONE}] \frown ms', \sigma')} \\
\\
\begin{array}{l}
\text{trigger} = \text{mk-Trigger}(\mathbf{true}, \text{test}, \text{dur}, \text{body}, \text{rev}) \\
\llbracket \text{test} \rrbracket \tau \sigma = \mathbf{false} \\
\text{dur} \in \text{Time} \Rightarrow \tau - \tau_0 > \text{dur} \\
\sigma' = \sigma \dagger (\text{rev} \triangleleft \sigma_0)
\end{array} \\
\\
\boxed{\text{Once after}} \frac{(ts, ms, \tau, \sigma') \xrightarrow{\text{trig}} (ms', \sigma'')}{([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{\text{trig}} ([\text{DONE}] \frown ms', \sigma'')}
\end{array}$$

**Fig. 13.10** The remaining semantic rules for executing triggers

$\sigma' = \sigma$ . The fifth hypothesis is the usual recursive step of executing the rest of the triggers.

Those are the three basic rules for executing triggers in the CSL semantics. There are ten semantic rules in total and they have been constructed so that there is no situation where multiple rules apply.

Of the remaining semantic rules, some of the base cases are given in Fig. 13.10. The first, Base, is only necessary to handle the base case of processing a sequence: that is, when the sequence is empty. Two rules, Done and Once after, deal with the non-repeating triggers: the former skips non-repeating triggers that have already triggered; and the latter places a DONE constant in the marker after the trigger completes.

The last four remaining rules in Fig. 13.11, Duration init, Duration pending, Duration cancel and Duration exec, handle the specific cases where a trigger with a duration on its conditional must, respectively, start monitoring the duration in which the conditional has been true; do nothing between the condition becoming true but before the duration has passed; stop monitoring the duration if the condition goes false before the duration passes; and to execute the trigger body when the duration is met.

### 13.5.2 CSL Statement and Expression Semantics

The rules for statements and expressions are included only to give a complete semantic model of the CSL. There are only three statements—assignment, output and quit—and the expression evaluation is essentially a subset VDM's expressions (with the singular addition of a special identifier to obtain the current time value).

$$\begin{array}{c}
\text{Duration init} \\
\hline
\begin{array}{l}
trigger = mk-Trigger(once, test, dur, body, rev) \\
\llbracket test \rrbracket \tau \sigma = \mathbf{true} \\
dur \in Time \\
(ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \\
\hline
([\text{trigger}] \frown ts, [\mathbf{nil}] \frown ms, \tau, \sigma) \xrightarrow{trig} ([(\tau, \sigma)] \frown ms', \sigma')
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{Duration cancel} \\
\hline
\begin{array}{l}
trigger = mk-Trigger(once, test, dur, body, rev) \\
\llbracket test \rrbracket \tau \sigma = \mathbf{false} \\
\tau - \tau_0 \leq dur \\
(ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \\
\hline
([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{trig} ([\mathbf{nil}] \frown ms', \sigma')
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{Duration pending} \\
\hline
\begin{array}{l}
trigger = mk-Trigger(once, test, dur, body, rev) \\
\llbracket test \rrbracket \tau \sigma = \mathbf{true} \\
\tau - \tau_0 < dur \\
(ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \\
\hline
([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{trig} ([(\tau_0, \sigma_0)] \frown ms', \sigma')
\end{array}
\end{array}$$

$$\begin{array}{c}
\text{Duration exec} \\
\hline
\begin{array}{l}
trigger = mk-Trigger(once, test, dur, body, rev) \\
\llbracket test \rrbracket \tau \sigma = \mathbf{true} \\
\tau - \tau_0 = dur \\
(body, \tau, \sigma) \xrightarrow{tstmt} \sigma' \\
(ts, ms, \tau, \sigma) \xrightarrow{trig} (ms', \sigma') \\
\hline
([\text{trigger}] \frown ts, [(\tau_0, \sigma_0)] \frown ms, \tau, \sigma) \xrightarrow{trig} ([(\tau_0, \sigma_0)] \frown ms', \sigma'')
\end{array}
\end{array}$$

Fig. 13.11 Duration rules for the CSL

### 13.5.2.1 Structure

Statements in CSL are represented by the *Stmt* type and consist of assignment statements, output statements to print message to the log and the quit statement.

$Stmt = Assign \mid Output \mid QUIT$

Assignment statements have the state variable that will be modified and an expression that is evaluated to a value assigned.

$Assign :: id : Id_v$   
 $e : Expr$

Output statements may print a message to one of three logs: the regular message log, PRINT, the warnings log, WARN, or the error log, ERROR.

$Output :: target : PRINT \mid WARN \mid ERROR$   
 $message : String$

Expressions in CSL are represented by the *Expr* type and consist of the special variable *TIME*, time values, *Time*, Boolean values, real number values, identifiers, unary expressions and binary expressions.

$$Expr = \text{TIME} \mid \textit{Time} \mid \mathbb{B} \mid \mathbb{R} \mid Id_v \mid \textit{UnaryExpr} \mid \textit{BinaryExpr}$$

Unary expressions in CSL include the arithmetic operators for negation and absolute value, the rounding functions floor and ceiling and the boolean negation operator.

$$\begin{aligned} \textit{UnaryExpr} &:: op : - \mid \text{ABS} \mid \text{FLOOR} \mid \text{CEIL} \mid \text{NOT} \\ &e : \textit{Expr} \end{aligned}$$

Binary expressions in CSL include the usual logical operators for conjunction, disjunction, implication and bi-implication; the basic arithmetic comparison operators; the usual arithmetic addition, subtraction, multiplication and division; and the integer modulo and division operators.

$$\begin{aligned} \textit{BinaryExpr} &:: op : \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \mid < \mid \leq \mid \geq \mid > \mid = \mid \neq \mid + \mid - \mid \times \mid \div \mid \text{MOD} \mid \text{DIV} \\ &a : \textit{Expr} \\ &b : \textit{Expr} \end{aligned}$$

Finally, the transition relation for statement execution in CSL is defined as  $\xrightarrow{tstmt}$ , which relates a tuple containing a sequence of statements, a current time value and an initial state,  $\Sigma$ , to a final state.

$$\xrightarrow{tstmt} : (\textit{Stmt}^* \times \textit{Time} \times \Sigma) \times \Sigma$$

### 13.5.2.2 Statement Rules

There are four rules to define  $\xrightarrow{tstmt}$ , giving the behaviour of statements in CSL.

$$\boxed{\text{Stmt base}} \frac{}{([], \tau, \sigma) \xrightarrow{tstmt} \sigma}$$

The first rule defines the base case behaviour for sequences of statements, just giving the state on the left of the transition relation as the final state.

$$\boxed{\text{Stmt Assign}} \frac{\begin{aligned} \sigma' &= \sigma \uparrow \{id \mapsto \llbracket e \rrbracket \tau \sigma\} \\ (rest, \sigma') &\xrightarrow{tstmt} \sigma'' \end{aligned}}{([mk\text{-Assign}(id, e)] \curvearrowright rest, \tau, \sigma) \xrightarrow{tstmt} \sigma''}$$

The rule for assignment statements evaluates the expression given the current time and state, and then executes the remaining statements in an appropriately modified state.

$$\boxed{\text{Stmt Print}} \frac{\text{output}(\text{target}, \text{message}) \quad (rest, \sigma) \xrightarrow{tstmt} \sigma}{([\text{mk-Print}(\text{target}, \text{message})] \curvearrowright rest, \tau, \sigma) \xrightarrow{tstmt} \sigma}$$

The function  $\text{output}()$  in the Stmt Print rule is left undefined; the intent is that it maps to some implementation-dependent logging facility.

The third type of statement, QUIT, does not have a semantic rule defined here. The quit statement is implementation-dependent as it is intended to stop simulation of the entire co-model.

### 13.5.2.3 Expression Evaluation

Expression evaluation in the CSL is defined in the usual way, though we include access to the current time value using the special identifier TIME. Note, also, that the set of time values, *Time*, is a subset of the real numbers.

The semantics of expression evaluation is given as a function,  $\llbracket \cdot \rrbracket$ , over three parameters that results in either a Boolean value or a real number. The first parameter is the expression to be evaluated, the second is the time at which the expression is evaluated and the third is the state in which it is evaluated.

$$\llbracket \cdot \rrbracket : Expr \times Time \times \Sigma \rightarrow (\mathbb{B} \mid \mathbb{R})$$

The specific cases in the definition of  $\llbracket \cdot \rrbracket$  are given below.

$$\begin{aligned} \llbracket \mathbf{true} \rrbracket \tau \sigma &= \mathbf{true} \\ \llbracket \mathbf{false} \rrbracket \tau \sigma &= \mathbf{false} \\ \llbracket \mathbf{TIME} \rrbracket \tau \sigma &= \tau \\ \llbracket r \rrbracket \tau \sigma &= r && \Leftrightarrow r \in \mathbb{R} \\ \llbracket id \rrbracket \tau \sigma &= \sigma(id) && \Leftrightarrow id \in \mathbf{dom} \sigma \\ \llbracket \text{mk-UnaryExpr}(op, e) \rrbracket \tau \sigma &= op(\llbracket e \rrbracket \tau \sigma) \\ \llbracket \text{mk-BinaryExpr}(op, e_1, e_2) \rrbracket \tau \sigma &= (\llbracket e_1 \rrbracket \tau \sigma) op (\llbracket e_2 \rrbracket \tau \sigma) \end{aligned}$$

The first four cases deal with constant values: the Boolean and real values evaluate to themselves (the first, second and fourth lines), and the special constant TIME evaluates the given time value,  $\tau$ . The fifth case deals with variable identifiers by returning their corresponding value from the state. The last two cases deal with operators by applying the operation to the result(s) of evaluating their operand(s).

## 13.6 Conclusion

In order to be able to trust the outcomes of model-based analyses, we need languages and tools that have a formal semantics, and the semantics must form the basis of the tooling. For co-simulation, the semantic basis is perhaps more intricate than

that of monodisciplinary modelling because of the need to be precise about the management of communication and the time in the collaborating simulations. The overall manner in which the Crescendo co-simulation works was presented at the start of this chapter, and that was used as a starting point to give the general semantic constraints under which a co-simulation framework must operate. With those constraints in mind, we described a modular operational semantic model that gives the behaviour of co-simulation while leaving the behaviour of the individual simulators abstract. We then presented an extension to the co-simulation semantic model that includes support for fault injection. This was presented as an extension to the whole co-simulation semantics to allow the fault injection to be abstract with respect to the simulators. We then gave the operational semantics of CSL, as that was the fault injection language implemented for the Crescendo tool. Readers interested in a detailed presentation of the VDM-RT simulator are invited to review the execution semantics [24].

# Chapter 14

## From Embedded to Cyber-Physical Systems: Challenges and Future Directions

John Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef

### 14.1 Introduction

The embedded systems market is now well established, with some estimates placing it in excess of \$120 billion, and expected to grow by over 6% per annum. This growth is driven in part by declining prices for multi-core devices, but also by increasing demand for smart devices that can be networked to create new applications in areas such as power grids, medical monitoring, automotive and transport. This rise in networked applications suggests that embedded systems of the types considered in this book are only the first generation of a larger class of *Cyber-Physical Systems (CPSs)* which consist of many collaborating computational elements interacting with the physical world [65, 81]. Examples might include the dynamic selection of electrical power sources in a grid, coupled with dynamic control of consumer-side devices so as to select the cheapest appropriate power source at a given time; collation and processing of data from multiple non-invasive sensors in the home to identify and respond to changes in a person's state of health; or cooperative platooning of road vehicles to manage fuel consumption.

The challenges facing developers of CPSs are significant [19, 72]. CPSs combine the characteristics of embedded products with the features of Systems of Systems (SoSs) such as the autonomy of the constituent systems, their distribution, capacity to evolve and—crucially—the reliance on the delivery of a behaviour

---

J. Fitzgerald (✉)  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk)

P.G. Larsen  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

M. Verhoef  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)



that emerges from the interactions of these independent constituents. The presence of multiple distributed, mobile and heterogeneous components and the need to accommodate change and reconfiguration make it difficult to demonstrate the levels of dependability required in many applications. CPSs will typically interact with other systems and human beings, so that they may be considered as an SoS involving diverse stakeholders and engineering disciplines in their design and operation. For example, development of a CPS for rail transport management might require collaboration between control engineers, mechanical engineers, power transmission specialists and software developers.

The pace of innovation in both the market and the technology of CPSs means that multidisciplinary methods and tools are needed to support rapid but accurate multidisciplinary exploration of design alternatives. Although model-based methods and tools require formal foundations in order to support consistent analysis, models must nevertheless be accessible to engineering practitioners and domain experts, be capable of integration with established techniques and processes [100] and support views covering different system facets.

Like embedded systems, CPSs feature close coordination between computational and physical elements [81]. Imagine, for example, automobiles moving under computer control in platoon. Platooning allows each vehicle to travel close enough to the car in front to benefit from the reduction in air resistance, saving fuel and increasing road capacity. It can only be done with computer software, but understanding the consequences of a software bug requires a model of the software in each vehicle, communications mechanisms available between cars and of course the physics of the vehicles and their surroundings. This is not merely an embedded systems design problem: it is essential to take account of multiple vehicles and their controllers (all from different manufacturers), as well as the communications infrastructures. Can co-modelling and co-simulation help us to explore the emergent behaviour of these larger-scale networked CPSs? If co-models represent reality with sufficient fidelity, such an approach could be a major cost saver in industries where assurance of CPS behaviour is needed [46].

A full-fledged CPS is a *network* of interacting elements rather than stand-alone devices. In contrast, the collaborative modelling presented in this book has been between *one* DE model and *one* CT model. In order to conveniently model and analyse networks of DE and CT elements, it is necessary to generalise the technology with support for *multiple* DE and CT models and a description of their connections. Such DE and CT elements may even be owned and managed by different stakeholders, leading to what is known also as SoS [70]. In our platooning example, we have the obvious vehicle-based constituent systems being independently managed by their drivers and/or control software, as well as the less visible constituent systems managing communications between vehicles, for example. Crucially, in a CPS as in an SoS, it is critical that attention is paid to the behaviours exhibited by the collective in addition to the behaviours of the individuals. In some CPSs, reliance is placed on some of these emergent behaviours—for example, the platoon is expected to provide a certain level of traffic

density—while other emergent behaviours may be undesirable, such as frequent and rapid changes in inter-vehicle spacing as a result of interactions between the features of the control algorithms operating in the different vehicles. The analysis of emergence is particularly significant when we consider exceptional behaviour related to faults and error recovery.

In this book, we have presented an approach to the multidisciplinary design of embedded systems. In this chapter, we examine the prospects for scaling co-modelling and co-simulation up to the challenge of CPSs. We begin an overview of co-modelling approaches (Sect. 14.2). Section 14.3 considers design flow for CPS and the role of co-modelling within it. Section 14.4 then considers how the limitation of one DE and one CT model can be generalised to include a collection of such models. Section 14.5 looks at the generalisation of co-simulation to incorporate other simulators and Sect. 14.6 considers the future about distributed control and intelligence. Afterwards, Sect. 14.7 discusses challenges for future research. Section 14.8 concludes.

## 14.2 The Co-modelling and Co-simulation Landscape

In order to realise the potential of model-based methods in the development of many real systems, including CPSs, it is essential to embrace models expressed in semantically distinct formalisms [95]. Hardebolle and Boulanger [41] identify several approaches to heterogeneous modelling. These include: translating models between formalisms, composing modelling languages to create a new composed language, composing models themselves, composing modelling tools and providing a unifying semantics. It is instructive to compare contemporary methods and tools against this scheme.

A *hybrid system* exhibits both continuous and discrete dynamic behaviour [2]. The term is often (wrongly) used interchangeably with embedded system or CPS. Hybrid systems are often described using transition systems that lend themselves to model checking, the major challenge here being to have a meaningful notion of time progression that can align with simulation of CT elements so that there is reasonable fidelity to the physical phenomena being modelled. A hybrid system can be described by means of a hybrid automaton consisting of a finite automaton with continuous dynamics. Each discrete state includes initial conditions for time and values of the continuous state, differential equations that describe the flow of the continuous state and invariants that describe regions of the continuous state-space where the system stays at the discrete state. The modelling of hybrid systems has also been investigated using UML, for example, in the work on HybridUML [12], introducing structure diagrams similar in some respects to the Internal Block Diagrams (IBDs) subsequently introduced in SysML, and providing for multiple views over a model. Hybrid systems approaches tend to fall into the “composition

of modelling languages” category. Here the combination of continuous and discrete time in the models is typically carried out in an ad hoc manner. However, if a modelling problem fits the constraints of a specific approach, it is generally very efficient.

Modelica [38] is a non-proprietary, object-oriented, equation-based language intended to model complex physical systems. Both commercial and free Modelica simulation environments exist. The Modelica Association is a non-profit organisation with members from Europe, USA and Canada. Since 1996, its simulation experts have been working to develop an open standard and an open-source standard library. Modelica has a comprehensive collection of components in its libraries, including components that enable the simulation of delays in network components. The DE modelling primitives lie closer to the abstraction level found in programming languages than the DE formalism we present in this book.

Ptolemy<sup>1</sup> [7, 30, 79] has studied modelling, simulation and design of concurrent, real-time embedded systems, using a mixture of models of computation using an actor-oriented modelling approach. Ptolemy takes a “composition of models” approach in that it permits coherent coupling of heterogeneous models. The models are organised into hierarchical layers, where each layer has one Model of Computation (MoC). To our knowledge, Ptolemy is the approach which supports the widest range of MoCs.

Matlab/Simulink created by MathWorks is one of the most widely used tools for creating CT models. Matlab is a modelling language and interactive environment which lets the user create models more rapidly than is the case using traditional programming languages. Simulink is an environment for multi-domain simulation and model-based design for dynamic and embedded systems. It provides an interactive graphical environment and a customisable set of block libraries which let the user design, simulate, implement and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing.

TrueTime is an extension of Matlab/Simulink that enables to study detailed timing models of computer-controlled systems [21]. This enables both timing delays for both the computing node and the network communications between them in a network control systems setting.

In our work, we aim to enable CT and DE experts to collaborate while being able to use established notations and tools that are familiar to them and are regarded as natural for their work. In Hardebolle and Boulanger’s terms, we are not composing modelling languages or models, but our work is based on “joint use of modelling tools”, with a “unifying semantics”—in our case an operational semantics—that describes the interaction between DE and CT simulators participating in a co-simulation.

---

<sup>1</sup>See <http://ptolemy.berkeley.edu/publications/index.htm>.

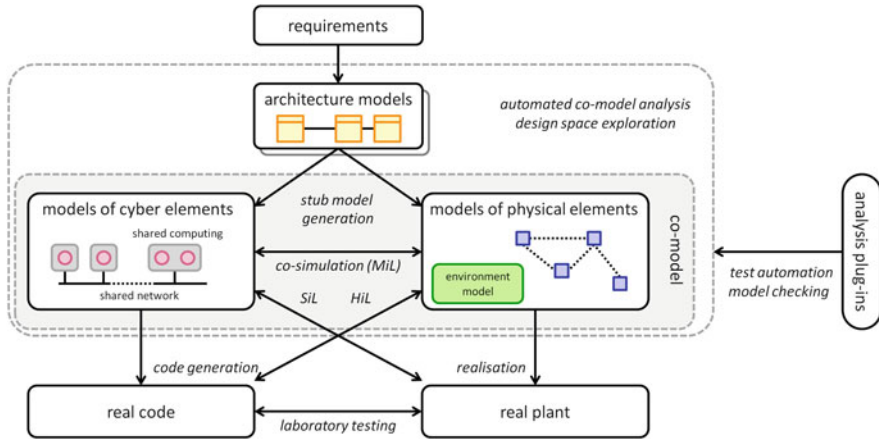


Fig. 14.1 CPS design flow from requirements to realisations

### 14.3 Co-modelling in the CPS Design Flow

One of the aims of co-modelling and co-simulation is to support the consideration of design alternatives. Where does this fit in the larger design flow for CPSs? It is worth noting that CPSs will typically integrate at least some pre-existing systems that may not have been conceived with the intention of contributing to the CPS and which may be independently owned and managed. The development of a complete CPS from scratch will therefore be a rare event. Nevertheless, it is important to consider a process of development, even though integration may form a greater part of the process than classical implementation. Figure 14.1 illustrates a possible flow for the development of CPSs. In addition to co-simulation, the figure also shows Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) simulations enabling a gradual transition of either DE or CT elements from models to their corresponding realisations. In order for this to work appropriately, it is not enough just to have *competent* co-models; one also needs to consider the *fidelity* of the different models. On the DE side, refinement theories exist to allow the relationships of abstract models to be addressed. However, on the CT side, this is not so simple because a CT model will hardly ever be a 100% match to a corresponding realisation.<sup>2</sup> The issue here is to determine when a model is sufficiently close to the corresponding realisation to be able to use simulations including the models to provide the evidence required for external stakeholders.

In implementing this kind of design flow, we envisage that SysML could be used more extensively than was suggested in Chap. 8. Requirements diagrams can be

<sup>2</sup>We might envisage partial automation of the realisation of physical elements using controlled manufacturing devices or even 3D printers.

used to link structured requirements to CPS elements and the analysis that needs to be conducted to check that the requirements have been satisfied. The composition of a CPS into its constituent elements can be described using Block Definition Diagrams (BDDs) and IBDs. At the lowest level, each of these can be represented either as DE or as CT elements, and thus it should be possible to derive the contracts that link such constituents together. This kind of information can then be used in simulations. Developers need to be able to gradually move from a modelling level to realisation. The realisations of DE elements are typically manifested in source code running on a CPU; automating this requires code generation support. For the CT elements, they are typically represented as a physical device interacting with its environment.

#### **14.4 Enabling Collections of DE and CT Models to Be Combined**

Combining collections of DE models into a single DE model, or collections of CT models into a single CT model, is conceptually straightforward if the topology of components can be statically determined. However, for many CPSs, the topology is likely to evolve dynamically, for example, in models of smart transportation, further increasing the complexity of the modelling task. In order to properly describe dynamic evolution, the BDDs and IBDs currently offered by SysML are insufficient, because they cannot describe dynamic behaviour. One solution would be to introduce a new type of “dynamic evolution” diagram to SysML; however, this would likely increase the complexity of models significantly. Therefore, adding such diagrams would only be worth the increased effort if the level of assurance required merited it. In order to support dynamic, evolving topologies at the modelling level, it is necessary to be able to express dynamic reconfiguration in the modelling language [73, 76]. The handling of time in CPSs is absolutely essential [18, 29, 64, 65]. CPSs demand that timing is a correctness criterion and not just a quality factor. Hence, time, mobility and dynamic topologies would all be essential elements of a reconciled operational semantics for co-simulation of CPSs.

#### **14.5 Open Co-simulation**

In general, CPSs involve cyber elements interacting not only with a full range of physical behaviours (electrical, mechanical, hydraulic, for example), but also with people. If we wish to model the effects of a particular control system in a public transport network, for example, it may be necessary to include agent-based models of human behaviour within a co-simulation [14, 90], or stochastic descriptions of

such behaviours. This potential for a wide range of constituent model types means that co-simulation platforms like Crescendo need to be both open and extensible.

One option for more open co-modelling than is currently available in Crescendo would be to create tool-specific Functional Mock-up Units (FMUs) that can be used in heterogeneous co-simulation via the platform-independent Functional Mock-up Interface (FMI) [13].<sup>3</sup> FMI comes in two flavours: FMI for Model Exchange (FMI-ME) and FMI for Co-simulation (FMI-Co). FMI-ME provides a mathematical description of a model. FMI-Co additionally includes a numerical solver to allow a co-simulation. FMI is a platform-independent standard that was developed in the MODELISAR project<sup>4</sup> for describing tool-specific models as FMUs. Four modes of FMI could be considered:

1. Importing FMU-MEs into tools that support a Crescendo co-simulation. This could be realised by developing a generic method for importing FMI-MEs into arbitrary simulation tools like 20-sim, ANSYS Simploer<sup>5</sup> and TRNSYS.<sup>6</sup>
2. Direct coupling of FMU-Cos to the Crescendo co-simulation. This could be realised both directly through the coupling of the XML-RPC protocol of Crescendo to the FMI and indirectly by importing the FMI-Cos into HiL tools like 20-sim 4C and RT-Tester.<sup>7</sup>
3. Converting FMU-MEs into FMU-Cos and using Method 2. This could be realised by developing a generic method to extend FMI-MEs with a numeric solver.
4. For integrating tool-specific models like TRNSYS into the Crescendo technology, export functionalities to FMU-MEs could be provided.

## 14.6 Ubiquitous and Distributed Computing

Distributed computing will play an increasingly important role in next generation CPSs. The importance of these concepts is already acknowledged in VDM-RT described in this book, as the language is execution context aware, with CPU and BUS as first-order citizens of the specification language and the keywords **duration** and **cycles** available to express execution time. This enables the description of deployments of software on a given hardware architecture, connected to an independent environment, which allows for light-weight performance analysis. Using the automated co-model analysis features of the Crescendo tools, it is even possible to make trade-offs between competing system architectures, automatically generated based on user specified templates. We consider this already a significant

---

<sup>3</sup><https://www.fmi-standard.org/>.

<sup>4</sup><http://en.wikipedia.org/wiki/MODELISAR>.

<sup>5</sup><http://www.ansys.com/>.

<sup>6</sup><http://www.trnsys.com/>.

<sup>7</sup><http://www.verified.de/en/products/rt-tester>.

step forward as current design methods typically ignore distribution and timing (which are critical for system-level performance requirements) until the test and integration phase, for example, using techniques such as worst-case execution time analysis. This typically leads to designs that are overdimensioned in order to overcome the uncertainty at design time, or mandate a specific model of computation, such as for example the time-triggered architecture [55], in order to make a priori guarantees about meeting non-functional requirements such as performance.

Even though VDM-RT provides features to express time aware and distributed software, there are some restrictions. For example, both the hardware architecture and the software deployment are static and cannot be changed at run-time, and only a single BUS can connect two CPUs. With the advent of self-healing and autonomous systems, with computing nodes that can dynamically change the processor speed and connectivity in order to balance power consumption and performance, it is clear that the capabilities that Crescendo currently offers are still limited in this respect. Furthermore, the types of communication buses and their associated behaviours are limited to just a common few, leaving the specifier to encode alternative approaches explicitly on top of these basic artefacts. An example of this is for example the Ether pattern, used to express lossy communication, as presented in Appendix C.3.1. Similarly, there are no out-of-the-box facilities to support broadcasting, for example, to mimic publish-subscribe (point to multi-point) style communication interfaces, as are paramount in wireless communication. Design patterns obviously provide flexibility at the cost of increasing model complexity and perhaps even simulation performance.

We believe that further improvements are needed to address the modelling challenges, in particular in the areas of multi-core and wireless computing, two competences that will carry future CPS implementations. Multi-core computing will be used to improve the performance of computation intensive tasks such as image processing, whereby algorithms are parallelised and executed on a large number of processing units that are interconnected within the same chip. Challenges that arise here are the integrity of the parallel code, synchronisation of the associated data flows in this massively parallel setting, possibly dynamic distribution of those artefacts over the available cores and in situ reconfiguration of the network topology while maintaining strict performance requirements and minimising power consumption. Wireless computing presents the challenge of a lossy communication channel with a constantly changing topology that depends on the current physical location of the communicating device. Wireless computing will be the back-bone of the so-called Internet-of-Things (IoT), where several devices might work together, in an ad hoc fashion, in order to perform a certain task. This even affects the more fundamental notion of what a system exactly is, as the system boundary might not be static but volatile in both space and time. This is commonly referred to as the SoS paradigm.

## 14.7 An Open and Lively Research Field

CPSs, their applications and their engineering are a highly active research area. The topic is intriguing because it brings together many ICT and non-ICT disciplines as well as systems and control engineering. Furthermore, significant public investment has been placed in research and development, notably in the USA, Germany and at the level of the European Union. The research challenges for CPSs in general are a subject of current debate [8, 19, 62, 63], and domain-specific challenges have been identified in areas including energy-sustainable CPSs [40] and in the medical domain [66]. From the point of view of co-modelling and co-simulation, and their role in design of dependable CPSs, we identify in outline some of the open areas for research:

**Modelling CPSs:** It is important to develop methodical approaches to CPS engineering [47], regardless of the specific formalisms used. Although we have presented specific formalisms for modelling DE and CT constituent elements, there are many other potential candidates, each with the limitations and advantages. Preliminary research, for example, has suggested that it is possible MATLAB/Simulink into the Crescendo framework [54]. However, there is a much wider class of formalisms relevant to CPSs that could be linked in. Examples might include agent-based models [14], cost or economic models. Deeper issues surround the semantics of integrating stochastic formalisms to describe probabilistic phenomena, for example.

**Verifying Properties of CPSs:** We have chosen to focus on simulation as a means of creating evidence about the properties of embedded systems and CPSs, but inevitably the quality of this evidence is limited by the quality of the underlying test set. Static analysis techniques, including proof and model-checking, allow symbolic analysis of models that, in principle, can deliver a high degree of confidence in the truth or falsehood of a proposition about a specific model. Such techniques can be partly or largely automated, but for each application there is a trade-off to be made between the investment demanded by these forms of verification and the evidence and risk reduction that they buy. The question of whether automated verification will ever be possible for fully fledged CPSs is an open one. There are current questions about the extent to which verification evidence composes over the structure of a CPS [98]. There have been attempts to do verification of CPS from a purely DE perspective [69] and with a focus on CT-side [89].

**Designing for Robustness of CPSs:** Error detection and recovery is an interesting topic in the context of co-modelling, since the technique enables a direct comparison of mechanisms both DE-side and CT-side, allowing a wide range of hybrid approaches to be analysed within a product development (for example, building DE-side redundancy to address CT-side errors or vice-versa). At the scale of CPSs, the challenges (and hence the potential benefits of co-modelling) magnify: you may not simply reboot all the cars in a platoon when an error is detected! In particular, the reliability of communications media will be a



bounding factor in deciding the range of mechanisms available. In our work, design patterns and guidelines for fault-tolerance in embedded systems were developed, but there are many other aspects of dependability that also need exploring in CPSs [74]. For example, there are interesting efforts to decouple system stability from cyber timing uncertainties at the design phase of CPS development [91]. We have indicated that people interact with and in CPSs, and indeed when considering “Human-in-the-loop” problems, we can envision technical CPSs collaborating with people to overcome human deficiencies.

**Controlling CPSs:** CPSs share some characteristics with SoSs, including the high degree of autonomy that may be shown by at least some of the constituent systems. This means that the constituents may have a bounded contractual relationship between one another, rather than exposing a full range of functionality to the other elements of the SoS. In considering the control of a CPS, we have to think not only about how accurate data on distributed state is gathered, but how effective control can be exercised in such an environment.

**Analysing and Ensuring Security and Safety in CPSs:** Although we have considered error detection and recovery, the design patterns and methods addressed in this book tend to assume non-malicious faults. Certainly, one can consider the augmentation of co-models of CPSs with details of vulnerabilities and the introduction of formalised attacker models. Given the independence of CPS constituents, there are certainly questions of managing trust [99] and provenance issues around the acquisition of critical data for decision-making in CPS control. Openness of platform standards and communications protocols is sometimes seen as a desirable characteristic of a CPS, enabling much of the agility that the CPS paradigm seeks to bring. However, CPSs have obvious attendant risks for both safety and security, making the development of appropriate design methodologies essential [10]. A safety roadmap for CPSs has been proposed, incorporating run-time features in order to cope with open and adaptive systems consider both security and safety with a focus on context awareness [94].

## 14.8 Conclusion

Our aim in this book has been to describe the foundations, methodology, practice and experience of collaborative modelling and co-simulation in the design of embedded systems. We began with the belief that this approach could have long-term value in promoting rapid and early innovation by providing a vehicle for real cooperation between engineering disciplines based on discrete event and continuous time formalisms. Our experience in several industry case studies is profoundly encouraging. The viability of our approach has been confirmed using independent modelling and simulation environments for VDM and bond graphs, harnessed together by the Crescendo tool linking Overture and 20-sim. Although we have a very new co-simulation engine here, we have been able to achieve positive results

with reductions in the number of early design iterations and a reduced need for early (expensive) physical prototypes.

Where next? Our work has only scratched the surface of co-modelling, with a limited initial range of supported simulation engines, but we have argued that the kind of collaborative working supported by Crescendo could be extended to allow the early-stage modelling and analysis of markedly more complex CPSs that demonstrate characteristics of both embedded systems and SoSs. The range of models could be extended significantly to permit trade-off analysis against a much wider range of design variables, including energy consumption, cost and human factors. We have concentrated to some extent on the role of co-modelling in addressing the complexity in design necessarily caused by the need to address faults and fault tolerance, often in both hardware and software.

The term “CPS” is still fairly new, and the field is still establishing itself. We would not encourage tribal debates about whether a system is or is not “really cyber-physical”. Instead, we prefer to concentrate on support for “cyber-physical thinking” during product development. This is characterised by the ability to move easily across the boundaries between discipline-specific models, exploring the full design space rather than one silo within it. Exercising cyber-physical thinking on products of modern scale and complexity entails having semantically well-founded support for the integration of a wide range of models, not just those of control software and the environment. Providing sound methods and robust tools to support such thinking has enormous potential benefit in systems development and engineering, from practical theories up to the provision of libraries of design patterns for models, system architectures and verification. Together, such techniques could help us all the more rapidly to develop embedded and cyber-physical products that earn a place in the market and also earn our trust.

# Appendix A

## 20-sim Summary

Christian Kleijn

### A.1 Introduction

20-sim is a modelling and simulation software package for mechatronic systems. With 20-sim, models can be created graphically, similar to drawing an engineering scheme. With these models, the behaviour of dynamic systems can be analysed and control systems can be designed. 20-sim models can be exported as C-code to be run on hardware for rapid prototyping and HIL simulation.

20-sim includes tools that allow an engineer to create models quickly and intuitively. Models can be created using equations, block diagrams, physical components and bond graphs. Various tools give support during the model building and simulation. Other toolboxes help to analyse models, build control systems and improve system performance. You can find more information on 20-sim on the website [www.20sim.com](http://www.20sim.com) and download the free Viewer.

Figure A.1 shows 20-sim with a model of a controlled hexapod. The mechanism is generated with the 3D Mechanics Toolbox and connected with standard actuator and sensor models from the mechanics library. The hexapod is controlled by PID controllers which are tuned in the frequency domain. Everything that is required to build and simulate this model is inside the package. No external software or compiler is needed!

---

C. Kleijn (✉)  
Controllab Products, Enschede, The Netherlands  
e-mail: [Christian.Kleijn@controllab.nl](mailto:Christian.Kleijn@controllab.nl)

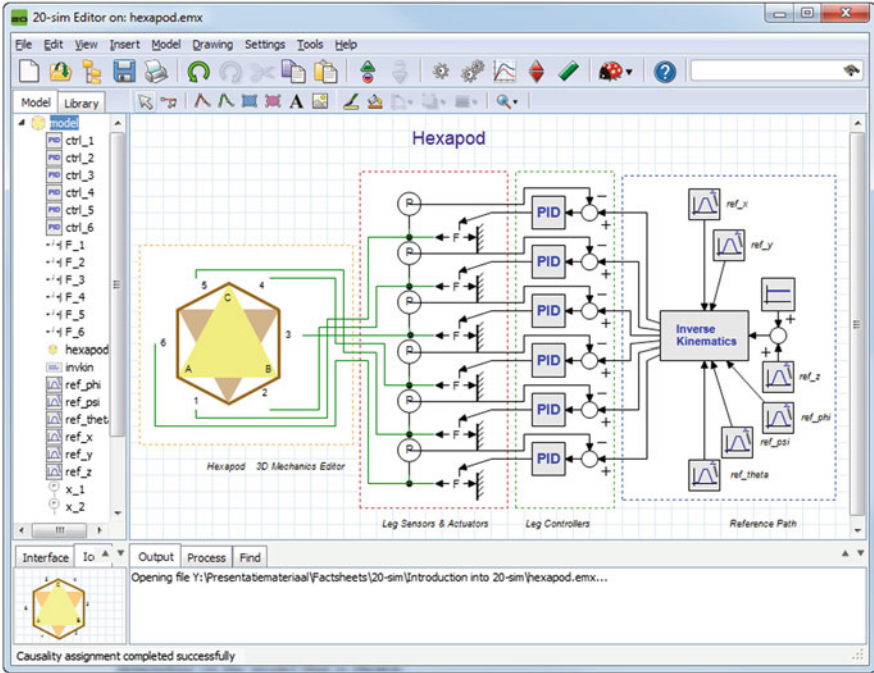


Fig. A.1 20-sim Editor with the model of a hexapod

## A.2 Overview

The 20-sim software consist of two windows which are tightly integrated. Models are created in the Editor, and simulation runs and results are shown in the Simulator. When 20-sim is started, the Editor will open. The Editor contains a model library tree from which you can drag and drop elements to the drawing canvas to construct your models. The 20-sim Editor will show CT models either in a graphical editor or in a text editor depending on the model that is shown.

The library contains elements for building bond graph models, components for building physical systems and blocks for building block diagram models. All library elements are open and can be changed by the user. The library contains the following elements:

**Bond Graph:** Elements for building bond graph models

**Iconic Diagrams:** Components for building physical systems:

**Electric:** Components for building electric networks

**Mechanical:** Components for building translational and rotational mechanical structures

**Hydraulics:** Components for building hydraulic systems

**Thermal:** Components for modelling heat transfer

**Block Diagrams:** Blocks for building block diagram models: Linear and non-linear blocks, sources and sinks, transfer functions, etc.

**Examples:** Example models showing the basic use of the library models.

### A.3 Graphical Models

Models in 20-sim are hierarchically oriented. The model on top is called the Main model. It is constructed using graphical elements which are called *Submodels*. Submodels can be connected. Depending on the submodel, the connection can be a shared variable or a physical connection. A Submodel itself can be constructed from multiple submodels, and these submodels themselves can be constructed with submodels, going many layers deep. At the bottom of the hierarchy, the models are described by sets of equations. These models are called *equation* models.

### A.4 Equation Models

Equation models are specified in a special language called SIDOPS++. SIDOPS++ has great resemblance with Maple, Matlab and other mathematical software packages.

Figure A.2 shows an example of an equation model. 20-sim equation models have a basic layout indicated by keywords:

**Parameters:** definition of values that do not change during simulation

**Variables:** definition of values that do change during simulation

**Equations:** the actual equations

Equations are relations between parameters and variables and indicated by an equal (=) sign. Various mathematical functions and operators are available for use in equations.

### A.5 Modelling Tools

20-sim comes with a number of tools to build advanced models:

**Controller Design Editor:** The Controller Design Editor helps users to design feedback systems with a linear plant, controller and pre-filter. Open and closed loop responses can be investigated using Bode and Nyquist plots.

**3D Mechanics Editor:** 3D mechanical systems are notoriously difficult to model using 1D elements. The 3D Mechanics Editor enables the user to define

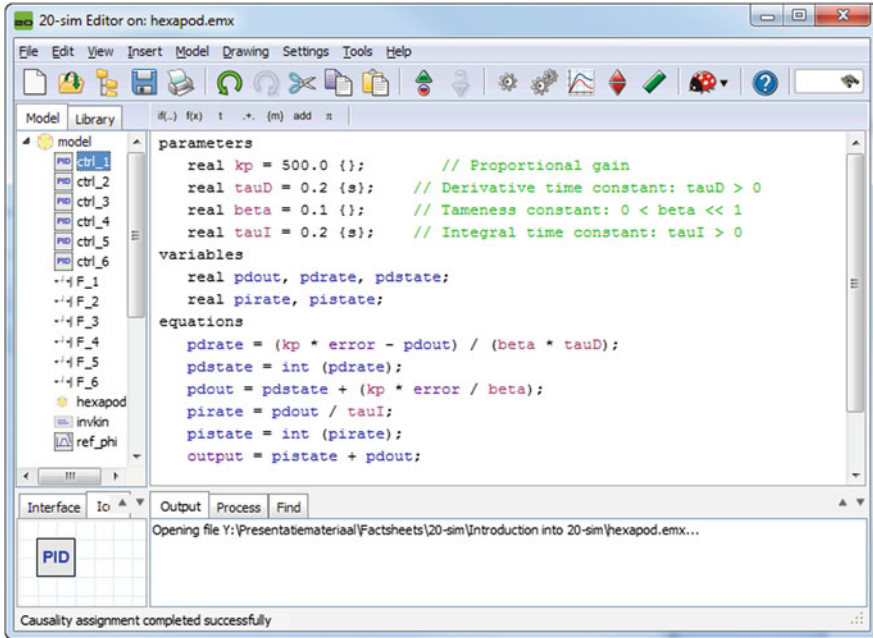


Fig. A.2 20-sim Editor with an equation model

mechanical systems by dragging and dropping bodies, joints and other objects in a 3D workspace. The corresponding set of optimised differential equations is generated automatically.

**Wizards:** Various wizards will help you to create motion profiles, define cams, build servo motors and much more.

## A.6 Simulation

When a CT model is ready, the Simulator can be opened from the Editor. Under the hood, the model is automatically compiled to create simulation code. 20-sim uses a built-in compiler for creating the simulation code, no additional tools are required. In the compiling-phase, 20-sim will check if the model is correct and optimise the equations. During optimisation, a built-in symbolic solver is used to remove derivatives, algebraic loops and other potential problems.

The Simulator is used to run simulations and analyse CT models. Before a simulation run can be started, the user has to define some settings:

**Run properties:** The start time and finish time of a run.

**Integration method:** 20-sim supports a number of advanced numerical methods for running a simulation. The numerical method can be chosen along with proper

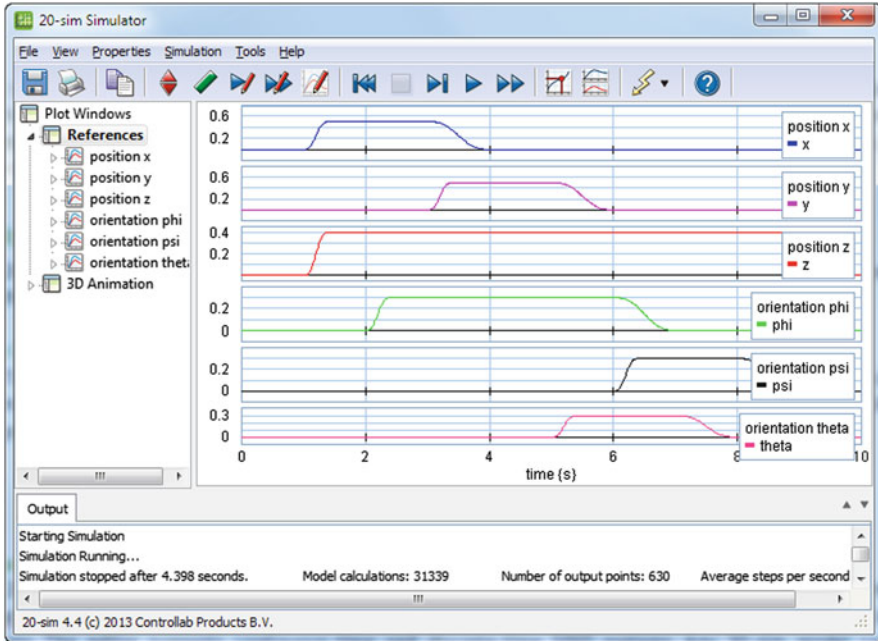


Fig. A.3 20-sim Simulator showing six simulation plots

settings. These settings include, for example, maximum integration error and step size.

**Parameter values:** Before a simulation run, the default parameters values may have to be changed.

**Plot properties:** The number and appearance of the plots have to be set and the variables to be plotted have to be chosen.

Next to simulation plots such as those shown in Fig. A.3, results can also be displayed as 3D animations in 20-sim. A special editor is available in which any variable can be connected to the position, orientation, size and colour of 3D objects. Standard 3D objects are available like cubes and spheres, but objects can also be imported from CAD packages.

## A.7 Analysis

The 20-sim package has two toolboxes which can be used to analyse models.

**Time Domain Toolbox:** The Time Domain Toolbox provides the ability to change parameter values and run multiple simulations to permit model analysis. Parameter sweeps, Optimisation and Curve Fitting will help to improve the

system performance. Sensitivity Analysis, Monte Carlo Analysis and Variation Analysis will help to check the robustness of a CT model.

**Frequency Domain Toolbox:** Models in 20-sim can be linearised to show the corresponding linear system in the Linear System Editor. The Linear System Editor is a specialised tool for the design and display of linear systems. The editor supports continuous-time and discrete-time SISO systems and can show the system response by Bode and Nyquist plots. If models cannot be linearised, Fast Fourier Transforms can be used to show the frequency behaviour of a model.

## A.8 Scripting

With scripting, tasks can be run in 20-sim automatically using specialised scripting functions. With these functions, models can be opened and run automatically, parameters can be changed, results can be exported and much more (Fig. A.4).

20-sim scripting functions can be run as m-files in Matlab or GNU Octave. GNU Octave is an open source environment that allows to run m-files similar to Matlab. Scripting functions are available to open and simulate 20-sim models, export parameters values to a 20-sim model, run simulations, export simulation plots and much more.

## A.9 Co-simulation

Based on the scripting engine, 20-sim can be run in a co-simulation. During simulation, 20-sim can receive input from external tools and send output to external tools. Co-simulation with the Crescendo tool and its predecessor DESTTECS are currently supported.

The only change that has to be made to enable a 20-sim model for co-simulation is the definition of import and export variables. These variables should be defined in equation models using the keyword “**external**”. Figure A.5 shows an example.

## A.10 Code Generation

Out of any 20-sim model, C-code can be generated for use in external systems, HIL simulators, etc. Templates allow customisation of the C-code with pre- and post-commands, file-linking, comments, etc. There are built-in templates that allow you to generate code for various targets:



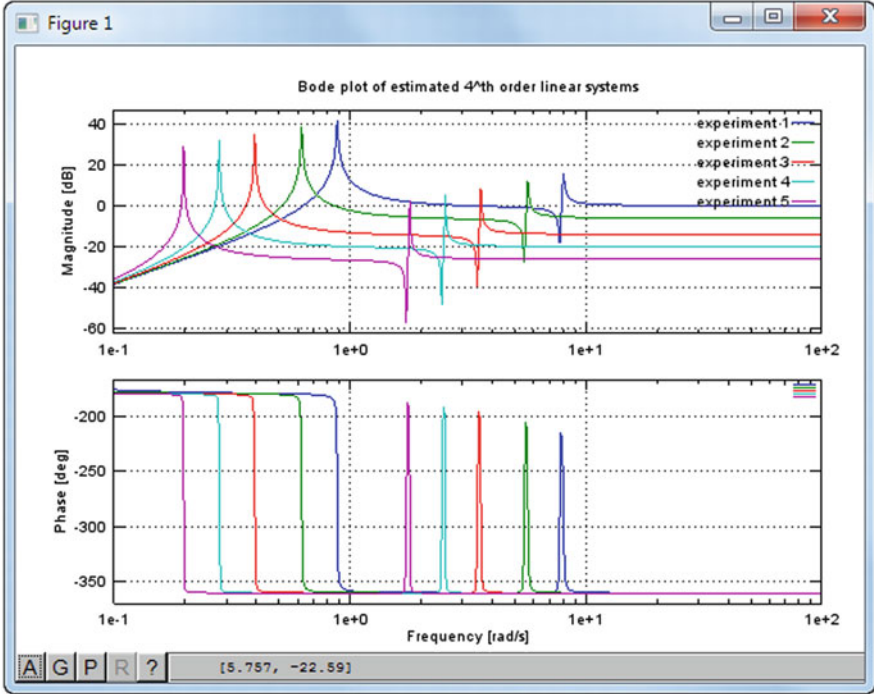
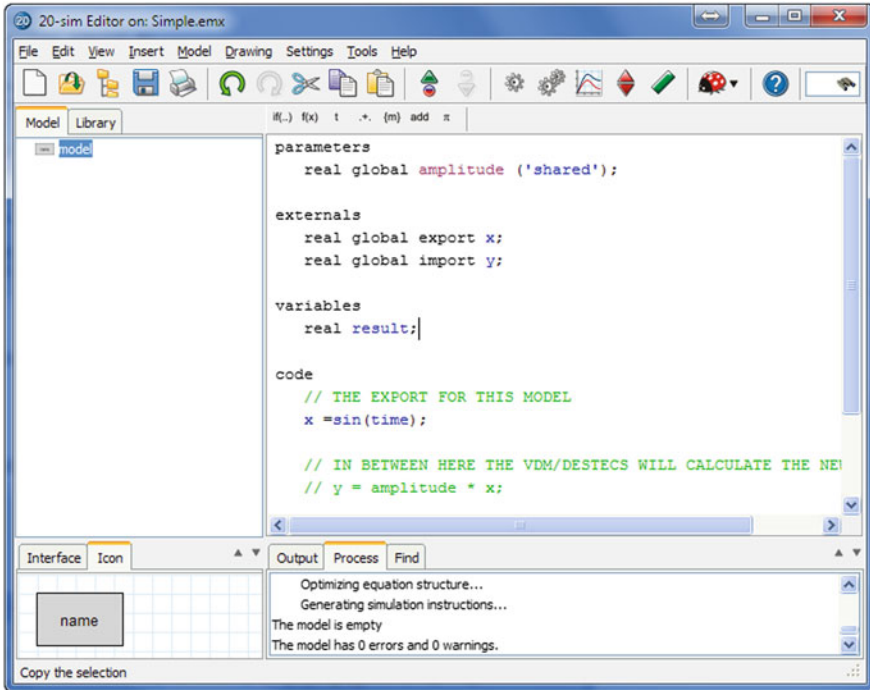


Fig. A.4 Bode plot in Octave, generated with scripting from a 20-sim model

**20-sim 4C:** The package 20-sim 4C helps to run c-code on hardware to control machines and systems. 20-sim 4C imports models (c-code) from 20-sim and runs them on hardware like embedded arm boards, PC 104 systems and much more. For more information, see [www.20sim4c.com](http://www.20sim4c.com).

**Matlab Simulink:** Generating C-code for use in MATLAB Simulink also includes a submodel block with input and output terminals. 20-sim uses the MEX-compiler to compile this code directly into an S-function. These S-functions can also be used in the Real Time Workshop in order to generate code for a specific platform, for instance, xPC Targets.

**C-Code:** 20-sim can generate standalone C-code for use in C and C++ programs. The generated C-code is supplied with several fixed step simulation algorithms to enable it to run in real time. The Euler and Runge–Kutta methods are supported by default.



**Fig. A.5** Equation model defining import and export variables for co-simulation

# Appendix B

## VDM-RT Language Summary

Peter Gorm Larsen

### B.1 Operators for Basic Types

Table B.1 provides an overview of the main operators for the basic types in VDM-RT (`bool`, `nat`, `nat1`, `int` and `real`). In these tables, the partial operators are indicated with the  $\overset{\sim}{\rightarrow}$  arrow. Note that the Boolean values `true` and `false` are different from numbers such as 1 and 0.

### B.2 Operators for Set Types

Sets are unordered finite collections of elements. Sets are written with curly braces, and the empty set is written as `{ }`. The operators on sets in VDM-RT are provided by Table B.2.

### B.3 Operators for Sequence Types

Sequences are finite ordered collections of elements. Sequences are written with curly braces, and the empty sequence is written as `[ ]`. The operators on sequences in VDM-RT are provided by Table B.3.

---

P.G. Larsen (✉)  
Aarhus University, Aarhus, Denmark  
e-mail: [pgl@eng.au.dk](mailto:pgl@eng.au.dk)

**Table B.1** Operators on basic types; “T” stands for any type

Operator	Name	Type
<b>not</b> b	Negation	<b>bool</b> $\rightarrow$ <b>bool</b>
a <b>and</b> b	Conjunction	<b>bool</b> * <b>bool</b> $\rightarrow$ <b>bool</b>
a <b>or</b> b	Disjunction	<b>bool</b> * <b>bool</b> $\rightarrow$ <b>bool</b>
a $\Rightarrow$ b	Implication	<b>bool</b> * <b>bool</b> $\rightarrow$ <b>bool</b>
a $\Leftrightarrow$ b	Biimplication	<b>bool</b> * <b>bool</b> $\rightarrow$ <b>bool</b>
a = b	Equality	T * T $\rightarrow$ <b>bool</b>
a $\langle \rangle$ b	Inequality	T * T $\rightarrow$ <b>bool</b>
-x	Unary minus	<b>real</b> $\rightarrow$ <b>real</b>
<b>abs</b> x	Absolute value	<b>real</b> $\rightarrow$ <b>real</b>
<b>floor</b> x	Floor	<b>real</b> $\rightarrow$ <b>int</b>
x + y	Addition	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
x - y	Difference	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
x * y	Product	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
x / y	Division	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
x**y	Power	<b>real</b> * <b>real</b> $\rightarrow$ <b>real</b>
x <b>div</b> y	Integer division	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
x <b>rem</b> y	Remainder	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
x <b>mod</b> y	Modulus	<b>int</b> * <b>int</b> $\rightarrow$ <b>int</b>
x < y	Less than	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
x > y	Greater than	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
x $\leq$ y	Less or equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>
x $\geq$ y	Greater or equal	<b>real</b> * <b>real</b> $\rightarrow$ <b>bool</b>

**Table B.2** Summary of VDM++ set operators

Operator	Name	Type
e <b>in set</b> s1	Membership	A * <b>set of</b> A $\rightarrow$ <b>bool</b>
e <b>not in set</b> s1	Not membership	A * <b>set of</b> A $\rightarrow$ <b>bool</b>
s1 <b>union</b> s2	Union	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>set of</b> A
s1 <b>inter</b> s2	Intersection	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>set of</b> A
s1 \ s2	Difference	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>set of</b> A
s1 <b>subset</b> s2	Subset	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>bool</b>
s1 <b>psubset</b> s2	Proper subset	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>bool</b>
s1 = s2	Equality	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>bool</b>
s1 $\langle \rangle$ s2	Inequality	<b>set of</b> A * <b>set of</b> A $\rightarrow$ <b>bool</b>
<b>card</b> s1	Cardinality	<b>set of</b> A $\rightarrow$ <b>nat</b>
<b>dunion</b> ss	Distributed union	<b>set of</b> <b>set of</b> A $\rightarrow$ <b>set of</b> A
<b>dinter</b> ss	Distributed intersection	<b>set of</b> <b>set of</b> A $\rightarrow$ <b>set of</b> A
<b>power</b> s1	Finite power set	<b>set of</b> A $\rightarrow$ <b>set of</b> <b>set of</b> A

**Table B.3** Summary of VDM++ sequence operators

Operator	Name	Type
<b>hd</b> $l$	Head	$\text{seq1 of } A \rightarrow A$
<b>tl</b> $l$	Tail	$\text{seq1 of } A \rightarrow \text{seq of } A$
<b>len</b> $l$	Length	$\text{seq of } A \rightarrow \mathbf{nat}$
<b>elems</b> $l$	Elements	$\text{seq of } A \rightarrow \text{set of } A$
<b>inds</b> $l$	Indexes	$\text{seq of } A \rightarrow \text{set of } \mathbf{nat1}$
$l1 \wedge l2$	Concatenation	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \text{seq of } A$
<b>conc</b> $l1$	Distributed concatenation	$\text{seq of seq of } A \rightarrow \text{seq of } A$
$l \mathbf{++} m$	Sequence modification	$\text{seq of } A * \text{map } \mathbf{nat1} \text{ to } A \xrightarrow{\sim} \text{seq of } A$
$l(i)$	Sequence application	$\text{seq of } A * \mathbf{nat1} \xrightarrow{\sim} A$
$l1 = l2$	Equality	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \mathbf{bool}$
$l1 <> l2$	Inequality	$(\text{seq of } A) * (\text{seq of } A) \rightarrow \mathbf{bool}$

**Table B.4** Summary of VDM++ mapping operators

Operator	Name	Type
<b>dom</b> $m$	Domain	$(\text{map } A \text{ to } B) \rightarrow \text{set of } A$
<b>rng</b> $m$	Range	$(\text{map } A \text{ to } B) \rightarrow \text{set of } B$
$m1 \mathbf{munion} m2$	Merge	$(\text{map } A \text{ to } B) * (\text{map } A \text{ to } B) \xrightarrow{\sim} \text{map } A \text{ to } B$
$m1 \mathbf{++} m2$	Override	$(\text{map } A \text{ to } B) * (\text{map } A \text{ to } B) \rightarrow \text{map } A \text{ to } B$
<b>merge</b> $ms$	Distributed merge	$\text{set of } (\text{map } A \text{ to } B) \xrightarrow{\sim} \text{map } A \text{ to } B$
$s <: m$	Domain restrict to	$(\text{set of } A) * (\text{map } A \text{ to } B) \rightarrow \text{map } A \text{ to } B$
$s < -: m$	Domain restrict by	$(\text{set of } A) * (\text{map } A \text{ to } B) \rightarrow \text{map } A \text{ to } B$
$m >: s$	Range restrict to	$(\text{map } A \text{ to } B) * (\text{set of } B) \rightarrow \text{map } A \text{ to } B$
$m > -: s$	Range restrict by	$(\text{map } A \text{ to } B) * (\text{set of } B) \rightarrow \text{map } A \text{ to } B$
$m(d)$	Map apply	$(\text{map } A \text{ to } B) * A \xrightarrow{\sim} B$
$m1 = m2$	Equality	$(\text{map } A \text{ to } B) * (\text{map } A \text{ to } B) \rightarrow \mathbf{bool}$
$m1 <> m2$	Inequality	$(\text{map } A \text{ to } B) * (\text{map } A \text{ to } B) \rightarrow \mathbf{bool}$
<b>inverse</b> $m$	Map inverse	$\text{inmap } A \text{ to } B \rightarrow \text{inmap } B \text{ to } A$

## B.4 Operators for Mapping Types

Mappings are finite unordered collections of pairs of elements with a functional relationship. Mappings are written with curly braces and with a small “| - >” arrow between the domain and range values. The empty mapping is written as  $\{ | - > \}$ . The operators on sequences in VDM-RT are provided by Table B.4.

## B.5 Record Types and Values in VDM

A record types (partly similar to `struct`'s in C) can be defined as:

```
RecordT ::
  sel1 : T1
  sel2 : T2
```

This is a *record type* with two fields. Note that this type definition uses “::” instead of the equality symbol used in other type definitions. This notation indicates that all values belonging to the type contain a *tag* holding the name of the type, `RecordT`. The presence of a tag allows us to define a constructor operator for the tagged type. The constructor is written `mk_tag`, where *tag* is the name in the tag. Values belonging to this type could be written as follows:

```
mk_RecordT(t1,t2)
```

The “mk” operator is known as a *record constructor*, and the `sel1` and `sel2` are called *selectors* that can be used to select elements from a record value.

## B.6 Small VDM-RT Examples

### B.6.1 General

Traditional if-then constructs can be used in VDM also with an **elseif** keyword.

```
if predicate then Expression elseif Expression
```

Casing between multiple alternatives can be carried out with a **cases** expression using pattern matching.

```
cases expression:
  pattern list 1 -> Expression 1,
  pattern list 2,
  pattern list 3 -> Expression 2,
  others                -> Expression 3
end
```

Given a set of values one can traverse over them using a statement like:

```
for all value in set setOfValues
do Expression
```

Inside a block statement it is possible to declare local variables like:

```
dcl variable : type := Variable creation
```

Otherwise local temporary “constants” can be defined using a **let**-expression (and here pattern matching can also be used):

```
let variable : type = Variable creation
in Expression
```

A **let-be-such-that** expression can be used to select an arbitrary value from a set satisfying some predicate, *pred*:

```
let variable in set setOfValues be st pred(variable)
in Expression
```

## B.6.2 Comprehensions (Structure to Structure)

Comprehension expressions can be very valuable for sets, sequences and mappings:

```
{element(var) | var in set setexpr & pred(var)}
{element(i) | i in set numsetexpr & pred(i)}
Typically:
{element(list(i)) | i in set inds list & pred(list(i))}
{dexpr(var) |-> rexpr(var) | var in set setexpr & pred(var)}
```

These can be used for describing implicit ways for characterising the elements contained inside the collection.

### ***B.6.3 From Structure to Arbitrary Value***

If one needs to select an arbitrary element from a set, it can be done as:

```
Select: set of nat -> nat
Select(s) ==
  let e in set s
  in
    e
pre s <> {}
```

### ***B.6.4 From Structure to Single Value***

In case one wish to extract a single value from a structure, it is most naturally done using recursion:

```
SumSet: set of nat -> nat
SumSet(s) ==
  if s = {}
  then 0
  else let e in set s
        in
          e + SumSet(s\{e})
measure CardMeasure
```

The **measure** here is used to show the reader (and the tool) how termination is ensured. Such a measure function like `CardMeasure` must take the same parameters as the function it is a measure for and yield a natural number.

### ***B.6.5 From Structure to Single Boolean***

For the special case where one needs to move from a set of values to a single boolean, quantified expressions are most natural:



```
forall p in set setOfP & pred(p)
exists p in set setOfP & pred(p)
exists1 p in set setOfP & pred(p)
```

the **exists1** quantifier is only true if there is exactly one value from `setOfP` that satisfy the predicate `pred(p)`.

## B.7 Threads and Synchronisation in VDM

In a VDM class, it is possible to define a *thread*. In the context of this book, **periodic** threads are used. These can be declared, for example, as:

```
thread
periodic(1000,10,200,0) (IncTime)
```

The four numbers used here in order represent the period (1000), jitter (10), delay (200) and offset (0) and the operation to be called in the example here is `IncTime`. Objects instantiated from a class with a *thread* part are called *active* objects. However, they still need to be actively started using a **start** statement.

When multiple threads exist, it may be necessary to synchronise them using *permission predicates*. These can, for example, be written as:

```
per Push => length < maxsize;
per Pop => length > 0
```

Such permission predicates can also make use of history counters (**#req**, **#act** and **#fin**) for describing **mutex** constraints.

## B.8 The System Class Concept in VDM-RT

In VDM-RT, a special **system** class is needed to be able to describe distributed systems. Here it is possible to declare the static structure of a distributed system. This is done using **CPUs** and **BUSses** which connect different **CPUs**. The constructor of the **system** class then enables deploying the statically declared top-level instances to the different **CPUs**.

## B.9 Example of Classes

New instances of a class (called an object) can be produced using a **new** expression. Inside such a class it is possible to refer to itself using the **self** expression.

```

class Person

types

public String = seq of char;
public Sex = Male | Female

values

protected Name : seq of char = "Peter";

instance variables

public nationality : seq of char := "Danish";
comment          : String;
yearOfBirth     : int;
sex             : Sex;
friends         : map String to Person

operations

public Person: int * Sex ==> Person
Person(pYear,pSex) ==
  (yearOfBirth := pYear;
   sex := pSex);

public GetAge : int ==> int
GetAge(year) == CalculateAge(year, yearOfBirth)
pre pre_CalculateYear(year, yearOfBirth);

functions

public CalculateAge : int * int -> int
CalculateAge (year,bornInYear) == year-bornInYear
pre year >= bornInYear
post RESULT + bornInYear = year;

Card: set of nat -> nat
Card(s) == card s;

thread
  while true do
    skip;

end Person

```

Subclasses can be defined as:

```
class Male is subclass of Person
...
end Male
```

And...

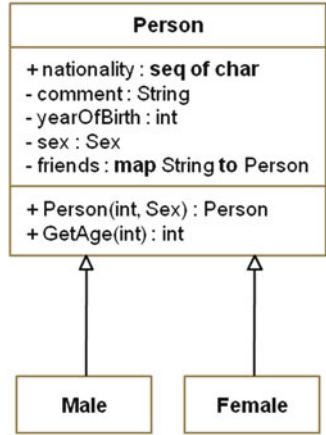
```
class Female is subclass of Person
...
end Female
```

## B.10 UML Diagrams

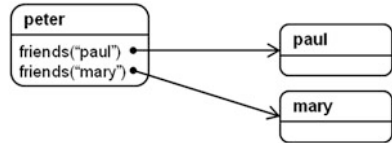
Classes can be visualised as a UML class diagram, where boxes represent classes and arrows indicate the relationships between classes. The classes in such diagrams may also show the instance variables and operations of the class. The subclass relationship between the `Person` class and the `Male` and `Female` classes is shown in Fig. B.1. The arrowhead in this case indicates a subclass relationship.

While class diagrams visualise the static relationships between classes, object diagrams show relationships between object at run time. They complement class diagrams, showing how instances of objects are related. In object diagrams, the boxes with rounded corners represent objects and arrows represent references to other objects. In the `Person` class example, the `friends` instance variable is a map of `Person` objects. Figure B.2 shows the relationship between three objects: a person called Peter, with two friends, Paul and Mary.

**Fig. B.1** UML class diagram for the `Person` class and its subclasses



**Fig. B.2** UML object diagram showing three-person objects and their relationships



# Appendix C

## Design Patterns for Use in Co-modelling

Carl Gamble, Kenneth Pierce, John Fitzgerald, Bert Bos, and Marcel Verhoef

### C.1 Introduction

A design pattern is a template that outlines a possible solution for a specified problem. 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 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 [39], which relates to object-oriented software, but which can be applied more generally.

The general form of each pattern includes a **name**, a **problem** description, a **solution** description and a description of the **consequences** of its application [39]. Adapting from Gamma et al. [39], pattern descriptions include the following sections:

**Name:** An identifier that conveys the essence of the pattern succinctly.

**Intent:** What does the pattern do? What problem does the pattern address? What is the rationale or intent?

**Motivation:** A scenario that illustrates how the pattern solves the problem and can help interpret the rest of the description.

---

C. Gamble (✉) • K. Pierce • J. Fitzgerald  
Newcastle University, Newcastle upon Tyne, UK  
e-mail: [carl.gamble@newcastle.ac.uk](mailto:carl.gamble@newcastle.ac.uk); [kenneth.pierce@newcastle.ac.uk](mailto:kenneth.pierce@newcastle.ac.uk);  
[john.fitzgerald@newcastle.ac.uk](mailto:john.fitzgerald@newcastle.ac.uk)

B. Bos  
Chess iX, Haarlem, The Netherlands  
e-mail: [bert.bos@chess-ix.com](mailto:bert.bos@chess-ix.com)

M. Verhoef  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

**Structure:** A graphical representation of the elements of the solution, for example, a UML class diagram.

**Application to DE Domain:** Model fragments and guidelines on how the solution can be realised within VDM (where applicable).

**Application to CT Domain:** Model fragments and guidelines on how the solution can be realised within 20-sim (where applicable).

**Use in Examples:** If a pattern is used in one or more of the running examples, it will be listed here (where applicable).

**Related patterns:** Descriptions of other patterns which are closely related (where applicable).

**Also known as:** Other well-known names for the pattern (where applicable).

## C.2 Controller Patterns

### C.2.1 *Minimal Controller Pattern*

#### C.2.1.1 Intent

To build a minimal DE controller model in VDM suitable for initial co-simulation.

#### C.2.1.2 Motivation

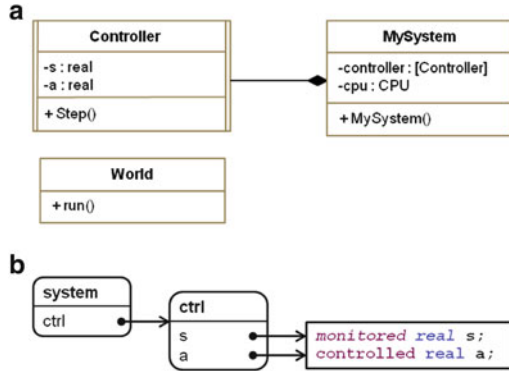
In order to test initial co-models, it is often useful to build a minimal controller with a simple structure to perform basic tests. With a small number of classes, it reduces the possibility for bugs in the DE controller, making it easier to determine if the co-model as a whole is working correctly. This pattern is suitable for initial testing of a co-model, that is, at the level of loop control. Although it does not follow best practice (see Sect. C.2.2), it is simpler and easier to debug for building initial models.

#### C.2.1.3 Structure

A class and object diagram is shown in Fig. C.1. The class diagram (Fig. C.1a) shows that the minimal VDM controller consists of three classes:

- **World.** This class is simply used as a bootstrap to initiate a co-simulation with the Run operation.
- **MySystem.** This system class is used to define the shared variables that will be updated by the co-simulation engine and the topology of the controller (which in this case is simple).

**Fig. C.1** Minimal VDM controller (class and object diagram). (a) Class diagram. (b) Object diagram



- **Controller.** This class is the actual controller, which defines a periodic thread and an operation that performs control: reads sensor values, performs calculations and writes actuator values.

The object diagram (Fig. C.1b) shows that the system holds a reference to the controller and that the controller holds references to two shared variables. Note the non-standard notation used to denote which elements of the contract the shared variables refer to.

### C.2.1.4 Application to DE Domain

The class definitions in Listings C.3–C.5 can be used to build a minimal controller for a co-model. The following notes can help in the use of these classes and in avoiding some possible errors.

- The **World** class in Listing C.3 can be used as is, with no modification.
- The **MySystem** class in Listing C.4 defines a single sensor and actuator (of type **real**). These should be replaced by variables that match the co-model contract (and with more useful names).
- The **MySystem** class may only define instance variables and a constructor (an operation called `MySystem` with return type `MySystem`).
- The constructor of **MySystem** is implicitly called *before* the `run` operation of **World**.
- The **Controller** class requires `Step` operation to be implemented with some control logic.
- The **Controller** class can access sensor and actuator values statically through the **MySystem** class, that is, `MySystem`s`, `MySystem`a`.
- The first parameter of the periodic thread definition controls the period (the other parameters can be left as zero for this minimal controller). This period is given in nanoseconds. A simple conversion from milliseconds to nanoseconds is to add the suffix `E6` to the previous value, that is, 10 ms is `10E6 ns`. The parameters of the periodic thread can be parameterised. This means that you can use a

function to convert to a period from a frequency, for example. You may also reference an instance variable as long as it has the right value when the thread is started. This means you can pass the period to the constructor of an object. See Listings C.1 and C.2.

```

values
  public FREQUENCY = 25; -- Hz

functions
  freq_to_period: real -> nat
  freq_to_period(f) == floor 1E9/f

thread periodic(freq_to_period(FREQUENCY), 0 ,0, 0) (Step);

```

**Listing C.1** Using a function as a parameter to a periodic thread (a)

```

instance variables
  private period: nat;

operations
  public Controller: nat ==> Controller
  Controller(p) == period := p;

thread periodic(period, 0 ,0, 0) (Step)

```

**Listing C.2** Using a function as a parameter to a periodic thread (b)

```

class World

operations

  -- run a simulation
  public run: () ==> ()
  run() ==
  (
    start(System`controller);
    block();
  );

  -- wait for simulation to finish
  block: () ==> ()
  block() == skip;
  sync per block => false;

end World

```

**Listing C.3** World.vdmrt



```

system MySystem

instance variables

-- sensor, actuator, controller and CPU
public static sensor: real := 0;
public static actuator: real := 0;
public static ctrl: Controller := new Controller();
private cpu: CPU := new CPU(<FP>, 1E6)

operations

public MySystem: () ==> MySystem
MySystem() ==
(
  cpu.deploy(ctrl)
)

end MySystem

```

Listing C.4 System.vdmrt

```

class Controller

public Step: () ==> ()
Step() ==
(
  -- read System'sensor
  -- perform calculation
  -- write System'actuator
)

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

end Controller

```

Listing C.5 Controller.vdmrt

## C.2.2 Object-Oriented Controller Pattern

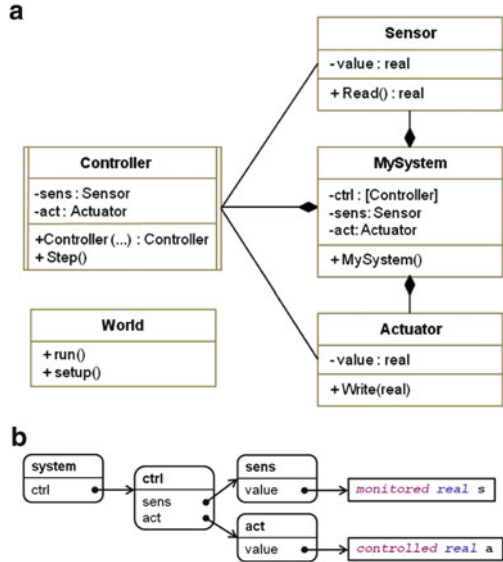
### C.2.2.1 Intent

To build a well-structured DE controller model in VDM for co-simulation.

### C.2.2.2 Motivation

The full controller model expands on the minimal controller (see Sect. C.2.1) and introduces more features and best practices. Once initial testing is complete and

**Fig. C.2** Full VDM controller (class and object diagram). **(a)** Class diagram. **(b)** Object diagram



it is necessary to build a more full VDM controller, the following pattern can help structure it. The main difference from the minimal pattern is that the shared variables are now 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.

### C.2.2.3 Structure

A class and object diagram is shown in Fig. C.2. The class diagram (Fig. C.2b) shows that the full VDM controller consists of five classes:

- **World.** This class is simply used as a bootstrap to initiate a co-simulation with the Run operation.
- **MySystem.** This class holds references to sensor and actuator objects (which in turn hold shared variables, which will be updated by the co-simulation engine). This class also handles creation of CPU and BUS objects and deployment of the controller, sensors and actuators to them.
- **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 object diagram (Fig. C.2b) shows that the system holds a reference to the controller and that the controller holds references to the sensor and actuator classes. Each of these contains a single shared variable. Note the non-standard notation used to denote which elements of the contract the shared variables refer to.

#### C.2.2.4 Application to DE Domain

The class definitions in Listings C.6–C.10 can be used to help construct a full controller for a co-model. 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). **MySystem** and **Controller** will all need to be modified to reflect these changes.
- The **MySystem** class may only define instance variables and a constructor (an operation called **MySystem** with return type **MySystem**).
- The constructor of **MySystem** is implicitly called *before* the run operation of **World**.
- The Crescendo tool requires that instance variables in the **MySystem** class are given a value, therefore the types are all given as optional (in square brackets) and they are initialised to **nil**.
- The use of **Sensor** and **Actuator** classes follows good object-oriented practice. It 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 **MySystem**), following good object-oriented practice.
- The controller is deployed to a CPU, meaning that statements within controller will take time (advance the simulation clock). The default is two cycles per statement. The sensors and actuators are not deployed and will therefore be (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.
- The first parameter of the periodic thread definition controls the period. See the tips from the minimal controller pattern above (Sect. C.2.1).

```

class World

operations

-- run a simulation
public run: () ==> ()
run() == (
  start(System`controller);
  block();
);

-- wait for simulation to finish
block: () ==> ()
block() == skip;
sync per block => false;

end World

```

Listing C.6 World.vdmrt

```

system MySystem

instance variables

-- sensor, actuator and controller
public static s: [Sensor] := nil;
public static a: [Actuator] := nil;
public static ctrl: [Controller] := nil

-- architecture
cpu: CPU := new CPU(<FP>, 1E6);

operations

public MySystem: () ==> MySystem
MySystem() == (
  -- instantiate System instance variables
  s := new Sensor();
  a := new Actuator();
  ctrl := new Controller(s, a);

  -- deployment
  cpu.deploy(ctrl);
)

end MySystem

```

Listing C.7 System.vdmrt

```

class Sensor

instance variables

-- this value will be set through the co-simulation
value: real := 0;

operations

-- read the current value of this sensor
public Read: () ==> real
Read() == return altitude;

end Sensor

```

Listing C.8 Sensor.vdmrt

```

class Actuator

instance variables

-- this value will be set through the co-simulation
private value: real := 0

operations

-- write a value to this actuator
public Write: real ==> ()
Write(v) == value := v

end Actuator

```

Listing C.9 Actuator.vdmrt

## C.2.3 Modal Controller Pattern

### C.2.3.1 Intent

To encapsulate modal behaviours using mode objects. Mode objects can be switched easily at runtime to change mode, activate fault-tolerance mechanisms or switch to degraded behaviours. This is a special case of finite state machine patterns and is similar to the strategy pattern.

### C.2.3.2 Motivation

Specifying modal controller behaviours, where the controller behaves differently in different circumstances, is a key benefit of co-modelling with a DE controller

```

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

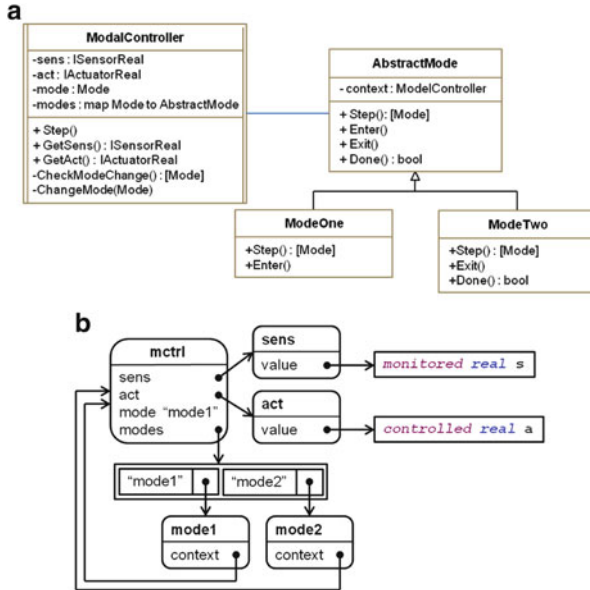
```

**Listing C.10** Controller.vdmrt

written in VDM. Modes can be described using simple conditional statements, but these become unwieldy as complexity is added and are difficult to maintain and evolve. The idea of this pattern is to encapsulate the algorithm for each controller mode into an object, much like the *strategy* pattern [39], such that these can be easily swapped at runtime. Designing each mode is then focused into a single operation in a single class. This also separates mode-switching from the modal behaviours themselves.

### C.2.3.3 Structure

A class and object diagram for this pattern is given in Fig. C.3. The important class in this pattern is the `AbstractMode` class. It holds references to sensors and actuators and has an abstract `Step` operation that should implement the control algorithm in a given mode. The `Step` operation in the `ModalController` class calls the `Step` operation of the current mode, thus delegating the responsibility of control. This modal controller `Step` operation can also determine if the mode



**Fig. C.3** Modal controller pattern (class and object diagram). (a) Class diagram. (b) Object diagram

should be switched, for example, if a sensor fails. The various mode subclasses should actually implement control actions.

### C.2.3.4 Application to DE Domain

The following listing is some (simplified and shortened) extracts from the *LineMeasurer2Sensor* example, showing how modes are initialised and how mode changing works. The mode types are defined like this:

```

types
  public Mode = token

instance variables
  protected mode: Mode;
  protected modes: map Mode to AbstractMode;
  private first: bool := true;
    
```

**Listing C.11** Mode types

Modes are initialised in the constructor. The use of a **token** type means that more modes can be easily added at run time. The controller class acts as the *context* for modes, so it passes itself to the constructor of each mode and has operations to access sensors, actuators and so on.

```

-- create modes
modes := {
  mk_token("WAIT")      |-> new WaitMode(self),
  mk_token("CALIBRATE") |-> new CalibrateMode(self),
  ...
};
-- initial mode
mode := mk_token("WAIT");

```

**Listing C.12** Mode constructor calls

The main control loop works like this. The `init` flag is used once at the beginning to call `Enter` on the initial mode. Then the controller checks if the mode needs to change (using another operation), then delegates control to the current mode. The mode may request an internal transition, in which case the mode is changed.

```

public Step: () ==> ()
Step() == (
  if first then (
    -- initial mode
    IO`printf("Initial mode: %s\n", [mode]);
    modes(mode).Enter();
    first := false
  );

  -- check if a mode change is needed
  let m = CheckModeChange() in
    if m <> nil then ChangeMode(m);

  -- delegate output to current mode
  -- change mode if it requests
  let m = modes(mode).Step() in
    if m <> nil then ChangeMode(m);
);

```

**Listing C.13** Checking for a mode change

The actual mode change is performed by an operation that calls `Exit()` on the current mode, switches the mode, then calls `Enter()` on the new mode.



```

public ChangeMode: Mode ==> ()
ChangeMode(m) == (
  -- call exit on the current mode
  modes(mode).Exit();
  -- change mode "pointer"
  IO`printf("Switching from %s to %s\n", [mode,m]);
  mode := m;
  -- call entry on new mode
  modes(mode).Enter();
)
pre m in set dom modes;

```

Listing C.14 Change mode operation

The mode change operation looks like this. Modes may use the `Done()` operation to show that their task has finished, then the controller moves to the next mode.

```

public CheckModeChange: () ==> [Mode]
CheckModeChange() == (
  -- wait for sensors to come on
  if mode = mk_token("WAIT") and modes(mode).Done()
  then return mk_token("CALIBRATE");
  ...

```

Listing C.15 Check mode change operation

### C.2.3.5 Use in Examples

This pattern is used in the *LineMeasurer2Sensor* example.

### C.2.3.6 Related Patterns

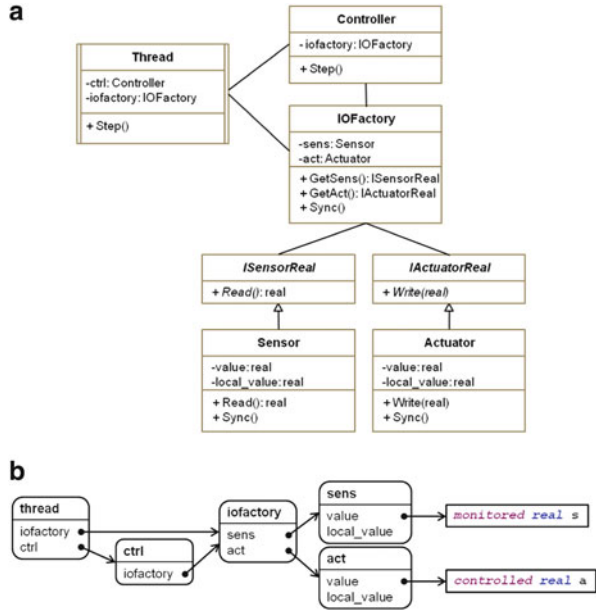
State pattern [39]; strategy pattern [39].

## C.2.4 IO Synchronisation Pattern

### C.2.4.1 Intent

To explicitly control co-simulation synchronisation in order to ensure accuracy of simulation and reduce simulation time.

**Fig. C.4** IO synchronisation pattern (class and object diagram). **(a)** Class diagram. **(b)** Object diagram



### C.2.4.2 Motivation

Due to the way co-simulation works, whenever (an instance variable linked to) a shared variable is read or written on the DE side, a co-simulation synchronisation occurs. This means that accessing shared variables more than once per control loop operation slows down co-simulation. More importantly, accessing a shared variable more than once can affect the outcome of a co-simulation because the value may change between reads due to a co-simulation synchronisation to occur. The solution in this pattern is to declare a second, local copy of each shared variable and explicitly control when these are synchronised.

### C.2.4.3 Structure

The basic structure of this pattern is to have a local copy of each shared variable and to explicitly read or write to that value to cause a synchronisation to occur. In a full object-oriented solution, each sensor and actuator object maintains its own local copy of its own shared variable and provides a `Sync` operation. This is shown by the class and object diagram in Fig. C.4.

The `IOFactory` class creates and hold the sensor and actuator objects. It provides operations to retrieve these objects, which the controller uses to access them. It has a `Sync` operations that synchronises that in turn calls `Sync` on all sensor and actuator objects. If placed in a `cycles` statement, only a single synchronisation will occur per control loop. To hide this synchronisation from the controller, the periodic

thread is defined in a `Thread` class. The `Step` operation of this class calls `Sync` on the IO factory object, then the `Step` (control loop) method of the controller.

#### C.2.4.4 Application to DE Domain

The following listing is some (simplified and shortened) extracts from the *LineMeasurer2Sensor* example, showing how synchronisation works. The extract from the `Encoder` class (a sensor) shows how a local variable is defined and synchronised:

```
instance variables
-- this value will be set through the co-simulation
protected val: real := 0;
-- local copy of the shared variable
protected local_val: real := 0;

operations
-- read the current value of this sensor
public Read: () ==> real
Read() == return local_val;

-- read shared variable to local variable
public Sync: () ==> ()
Sync() == local_val :=
    CountToDistance(if reversed then -val else val);
```

**Listing C.16** Synchronising a local variable in a sensor

The synchronisation in `IOFactory` is then done as in the following listing:

```
public Sync: () ==> ()
Sync() == (
    cycles(20) (
        -- sync actuators
        servoLeft.Sync();
        servoRight.Sync();
        -- sync sensors
        encLeft.Sync();
        encRight.Sync();
    );
);
```

**Listing C.17** Synchronising sensors and actuators

Finally, the `Thread` class initiates synchronisation before calling the `Step` operation of the controller. Note that in this case the controller contains an operation

call `IsFinished()` that will return true if the simulation should end. After this, no synchronisations will occur and the co-simulation will essentially be CT-only, finishing very quickly:

```

Step: () ==> ()
Step() == (
    if not controller.IsFinished() then (
        io.Sync();
        controller.Step()
    )
);

```

**Listing C.18** Bypassing synchronisation to finish a simulation quickly

### C.2.4.5 Use in Examples

This pattern is used in the *LineMeasurer2Sensor* example.

## C.2.5 Demux Pattern

### C.2.5.1 Intent

To divide access to a composite shared variable among (sensor or actuator) objects to maintain the object-oriented principle.

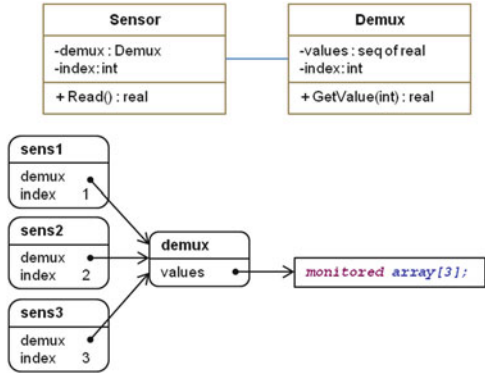
### C.2.5.2 Motivation

Arrays and matrices (of real numbers) may be defined within a contract. These are useful for describing groups of the same type of sensor or actuator. We advocate having one object per sensor and actuator, therefore a composite shared variable should be accessible by more than one object. The solution in this pattern is to introduce an intermediate “demux” object to hold the reference to the shared variable and permit access to the elements of the sequence.

### C.2.5.3 Structure

A class and object diagram for this pattern is given in Fig. C.5. A *Demux* class is used to hold the actual shared variable and provides access through an operation, taking the index as a parameter. Each sensor object holds a reference to the demux object, as well as the index it should access. These are set through the constructor.

**Fig. C.5** Demux pattern  
(class and object diagram)



### C.2.5.4 Use in Examples

This pattern is used in the *LineMeasurer2Sensor* example.

## C.2.6 Decorator Pattern (for Evolution)

### C.2.6.1 Intent

Use the decorator pattern to maintain backwards compatibility (e.g., for regression testing) during evolution by decorating simple objects with new features.

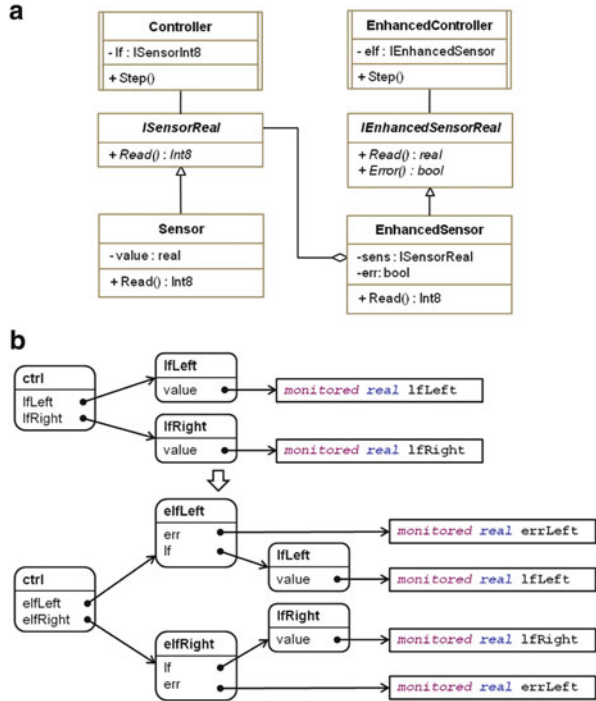
### C.2.6.2 Motivation

As co-models evolve, components such as sensors can undergo changes such as the addition of behaviours such as realistic or faulty behaviours. This can lead to changes being required in classes, such as the addition of new operations, which in turn can require changes to many other existing classes. If it is necessary to preserve existing classes, the decorator pattern [39] can be applied. This involves the creation of a new class that includes the new operations and holds a reference to an object of the original, simpler class. This means that simple objects can be passed to the older, legacy classes where necessary and the enhanced object passed to new classes.

### C.2.6.3 Structure

A class and object diagram for this pattern is given in Fig.C.6. The original structures before the evolution are the `Controller`, `ISensorReal`, and `Sensor` classes on the left-hand side. The decoration occurs through the

**Fig. C.6** Decorator pattern for evolution (class and object diagram). (a) Class diagram. (b) Object diagram



introduction of the `IEnhancedSensorReal` class that includes the operations of the `ISensorReal` and new operations introduced in the evolution (in this case, adding an error flag and associated operation). The enhanced sensors hold a reference to the simple sensor, delegating to it for basic functionality. In this structure, the simpler controller can still be instantiated for regression testing using the undecorated `Sensor` class. The object diagram shows how the two “enhanced” objects are now included and pointed to by the controller. Note how the original object remains unchanged.

### C.2.6.4 Application to DE Domain

The following code shows how the (concrete) decoration works in this pattern. The enhanced sensor holds the new shared variable and a reference to the original sensor object. Note how it delegates the old functionality to the original object in `Read`:

### C.2.6.5 Use in Examples

This pattern was used in older versions of the *LineMeasurer2Sensor*; however, the current version with the *Crescendo* tool has been refactored.

```

instance variables
  val: real := 0;

operations
  -- read the current value
  public Read: () ==> real
  Read() == return val;

```

**Listing C.19** Original sensor

```

instance variables
  err: bool := false;
  sens: ISensorReal;

operations
  -- read the current value
  public Read: () ==> real
  Read() == return sens.Read();

  -- true if the sensor is in an
  -- error mode, false otherwise
  public Error: () ==> bool
  Error() == return err;

```

**Listing C.20** Enhanced sensor

## C.3 Fault Patterns

### C.3.1 Ether Pattern

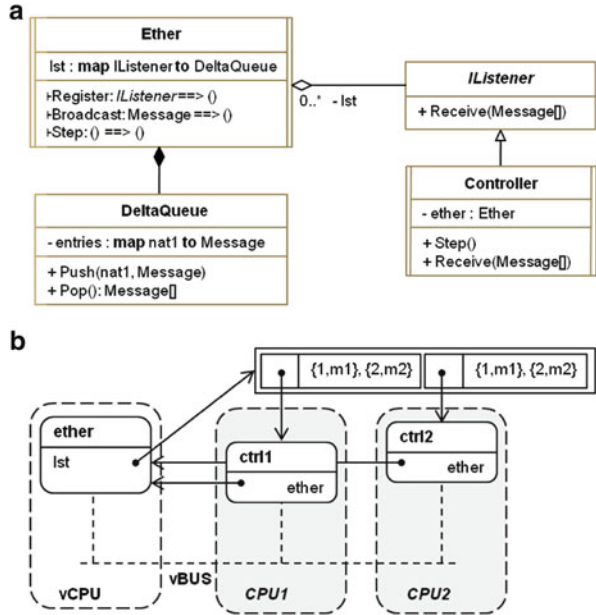
#### C.3.1.1 Intent

To model realistic communications between controllers through networks by explicitly modelling a communications medium.

#### C.3.1.2 Motivation

Distributed controllers need to communicate with each other. In VDM, the built-in concept of the CPU and BUS can be used to model this; however, these connections are always ideal: no messages are ever lost, duplicated or corrupted. In order to introduce realistic networked communications, this pattern introduces an explicit model of an *ether*, which represents some medium that data must travel through. This pattern can be applied to represent direct connections between controllers, a connection through an ethernet or wireless communications. The implementation of the ether can be tailored for specific applications, for example, different quality of service levels guaranteed by the medium.

**Fig. C.7** Ether pattern (class and object diagram). (a) Class diagram. (b) Object diagram



### C.3.1.3 Structure

A class and object diagram for this pattern is given in Fig. C.7. In the object diagram, it can be seen that the ether is deployed on the virtual CPU, with two controllers on separate (real) CPUs. These are connected by a virtual bus. The controllers can send messages to the ether, which in turn stores them in a (delta) queue. When enough simulated time has passed, the ether passes the messages to the CPUs.

The *IListener* interface represents an object that is connected to the ether. Classes that implement this interface (e.g. the controller) must give a definition for the *Receive* operation, which is called when a message arrives from the ether. The *Ether* class explicitly models a medium over which messages pass. All devices connected to the ether must call the *Register* operation. Messages can be sent using *Broadcast* (as an example) and the class handles the distribution of messages. The ether class is active: it defines its own thread to actively send messages. The *DeltaQueue* class is one way in which the ether could model travel time (and delays) for messages. As messages are received, they are added to the queue with a given delay. The ether class then periodically updates each queue to indicate the passing of time and messages that have spent the correct time in the queue are removed and passed to recipients. The *Controller* class represents the DE controller. It maintains an instance variable of the ether allowing it to send messages to the ether. The controller class must implement the *IListener* interface in order to access the ether.



### C.3.1.4 Use in Examples

The ether pattern is used in the *chesswayDESTECS* example.

## C.3.2 Noise Pattern

### C.3.2.1 Intent

To alter a component's output signal from ideal to realistic.

### C.3.2.2 Motivation

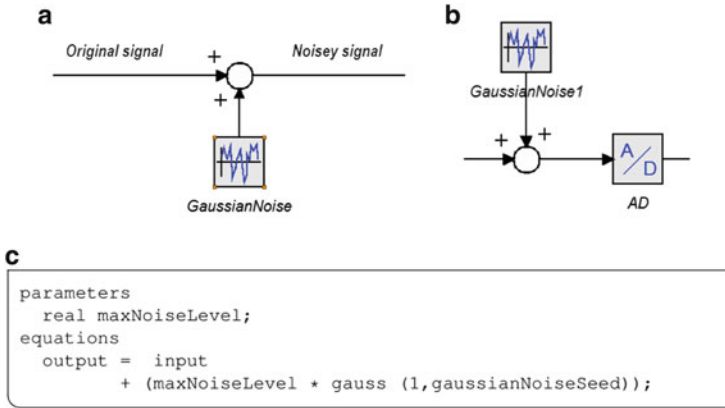
Many electrical components have their inputs in the analogue domain through either measurement of some real-world property or receipt of an analogue signal. For such a component to be used by a digital computer, there must be an element performing analogue to digital (A/D) conversion. One point at which random noise can enter the signal value is when the analogue signal is converted to a digital value. It is this noise that this pattern attempts to simulate. The noise is assumed to be random (Gaussian) with a magnitude defined by the number of bits in the output signal that could be affected.

### C.3.2.3 Structure

The general structure of the noise pattern is shown in Fig. C.8a, where the original signal output by part of the model has noise added to become a noisy signal.

### C.3.2.4 Application to CT Domain

Two implementations are presented that would allow this behaviour to be modelled in the CT domain. The first implementation introduces the noise to the signal before it enters the A/D block. In this implementation, the maximum magnitude of the Gaussian noise must be calculated externally to the model and input into the Gaussian block (see Fig. C.8b). The second implementation includes the generation and addition of the noise within the A/D block, resulting in a noisy A/D block (see Fig. C.8c). This block contains two parameters `noiseLevelInBits` and `gaussianNoiseSeed`. The former dictates the maximum magnitude the noise may take while the second allows the seed used to generate the Gaussian random number stream to be changed. While this implementation is simpler to use, care must be taken to ensure that code and parameters introducing the noise are only



**Fig. C.8** Diagram showing the general structure of adding noise to a signal in 20-sim (a); mixing a noisy signal (b); and simulating A/D noise (direct manipulation) (c)

used to influence the tolerances of a component specification and are not included in its ideal design.

### C.3.2.5 Application to DE Domain

An equivalent effect may be achieved in the DE domain by using the `rand()` operation from the MATH library. The output from the `rand()` command may be scaled and added to the original signal to produce a noisy signal.

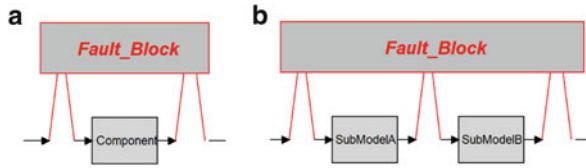
### C.3.2.6 Use in Examples

The noise pattern is used in the *LineMeasurer2Sensor* example.

## C.3.3 Fault Injector Pattern

### C.3.3.1 Intent

To allow the injection of specific fault behaviours into a nonfaulty component model.



**Fig. C.9** Fault injector pattern. (a) Single component wrapped by fault block. (b) Two components wrapped by a single fault block



**Fig. C.10** The original, ideal, encoder used in the *PaperPinch* model

### C.3.3.2 Motivation

As our methods prescribe that fault behaviour should be separate to the normal behaviour of a component, it is necessary to be able to intercept and alter the signals passing between the elements of the normal component model to simulate faulty behaviour.

### C.3.3.3 Structure

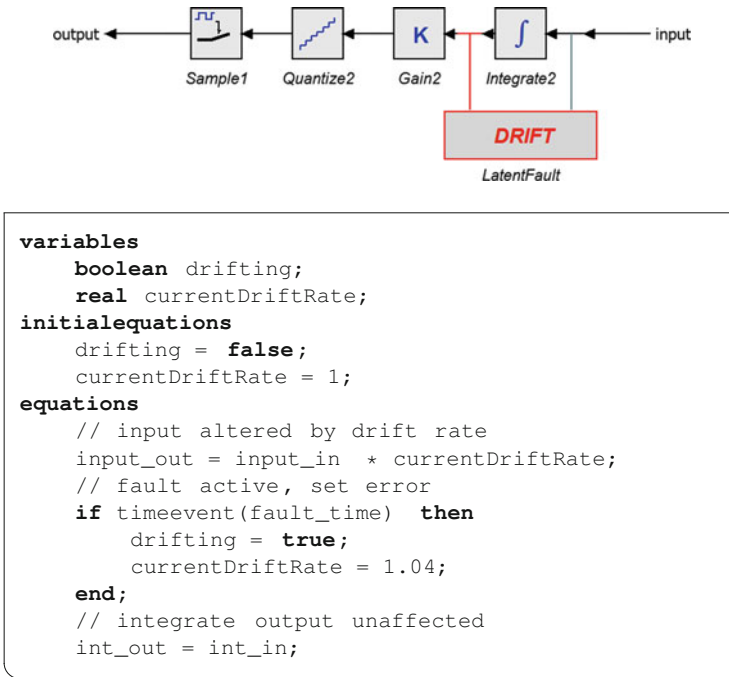
Two example block diagrams are given in Fig. C.9. The first shows an example of a component which is wrapped by a fault block (see Fig. C.9a). Here the component’s input may be pre-processed by the fault block before being passed to the component, then the output from the component may be processed before it is passed 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 (see Fig. C.9b).

### C.3.3.4 Application to CT Domain

#### Value Drifting Fault

An encoder can be modelled in 20-sim by integrating the velocity of an object and then conditioning this value by including a gain to represent the number of counts per unit of travel, and then quantising to the resolution of the A/D output (see Fig. C.10).

As drift represents an incorrect view of the total distance travelled, this is modelled here by scaling the speed value that is input to the integrator component. Missed or added counts are faults and so should be modelled externally to the ideal



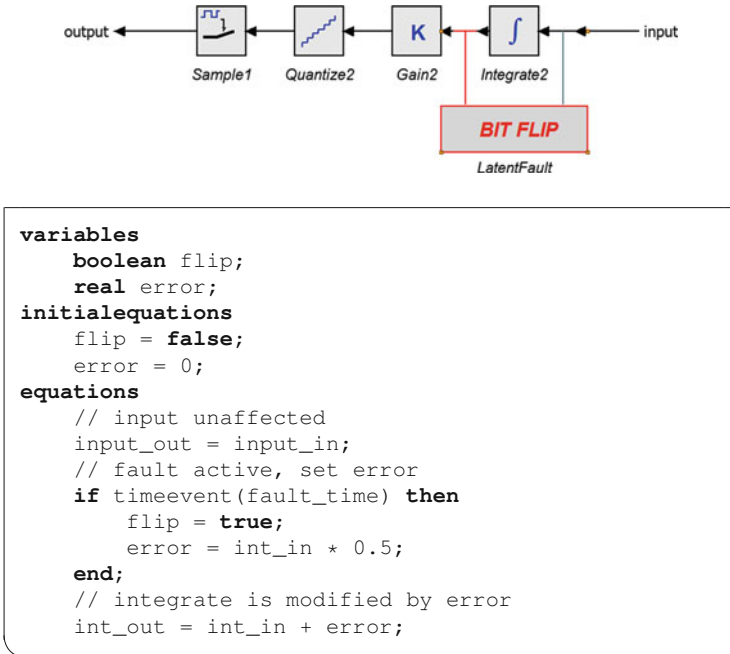
**Fig. C.11** Graphical encoder block (*top*) and code inside the drift fault block (*bottom*)

or realistic behaviour of the encoder, so the modification of the integrator input signal is performed by a visually distinct block added on to the previous model (Fig. C.11 (top)).

The drift fault block encapsulates the original integration block such that all input to the integrator passes through the drift fault block and the output from the integrator also passes through the fault block. This allows the drift fault block to apply a multiplier to the speed value before it is integrated. A multiplier of more than one model extra counts while less than one model missed counts. The code within the drift fault block is shown in Fig. C.11 (bottom).

### Bit Flip Fault

The result of a bit flip in the memory of an encoder is a sudden jump in the value of the count output from that encoder. The implementation, shown in Fig. C.12 (bottom), simply adds a fixed error value to the output of the encoder. When this fault implementation is selected, the graphical view will show the ref bit flip icon as in Fig. C.12 (top). A more realistic implementation of this fault block would consider the bit width of the encoders output, along with any scaling, and ensure that the error value equated to the value of one of the bit positions in that number.



**Fig. C.12** Graphical encoder block (*top*) and code inside the bit flip fault block (*bottom*)

### C.3.3.5 Use in Examples

The fault injector pattern is used in the *PaperPinch* and *LineMeasurer2Sensor* examples.

## C.4 Fault Tolerance Patterns

### C.4.1 Voter Pattern

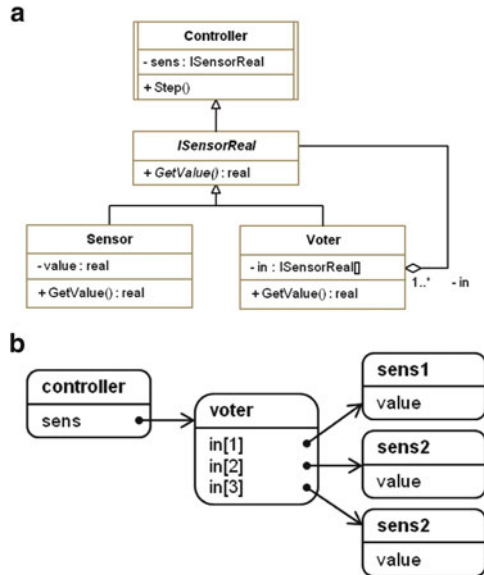
#### C.4.1.1 Intent

To produce a single sensor reading from multiple (redundant or diverse) sensor inputs.

#### C.4.1.2 Motivation

Where sensors can fail, multiple sensors can be introduced as a way to achieve dependability. This can be done through replication (using copies of the same sensor) or diversity (using different sensors). In order to gain a single value from

**Fig. C.13** Voter pattern (class and object diagram). (a) Class diagram. (b) Object diagram



various inputs, a voter can be used. A simple voter could take the mean of the incoming values or use a majority vote to ignore erroneous readings.

### C.4.1.3 Structure

A class and object diagram for this pattern is given in Fig. C.13. In this pattern, a `Voter` class is introduced that implements a sensor interface (`ISensorReal`). The voter class also holds one or more sensor objects (aggregation). By implementing the sensor interface, a voter object can be passed transparently to the controller such that the controller does not need to be altered and provides a single value from the multiple inputs. The voter pattern can be combined with the *strategy pattern* [39] by describing the voting algorithm as an interface and providing different implementations of this interface to explore alternative voting routines.

The object diagram in Fig. C.13b shows that the controller holds a reference to the voter object, which then aggregates (in this case) three other sensor objects (which represent distinct data sources). When the controller reads its sensor object, the voter decides on the value that the controller receives. This pattern is similar in structure to the filter pattern (see Pattern C.4.2).

### C.4.1.4 Application to DE Domain

The `Voter` class shown in Listing C.21 implements the `ISensorReal` interface and holds a reference to a sequence of `ISensorReal` objects (the sensors to be

voted on). In this case, the voter requires three sensors as inputs and uses a function `close` to compare values (to avoid issues with comparing real values using the equals operator). The voting occurs within the `Read` operation. First, the votes between each pair of sensors is calculated: (1,2), (2,3), and (1,3). Then, those pairs that agree are placed in the list `agree`. The length of this list determines how the vote went. This simple scheme could be extended to work with any number of sensors and made more complex to include weightings or majority votes. This pattern could also be extended with the *strategy* pattern by wrapping the voting operation in a class such that the voting algorithm can be selected by passing objects to the constructor to easily test different schemes.

#### C.4.1.5 Use in Examples

The voter pattern is used in the *PaperPinch* example.

#### C.4.1.6 Related Patterns

Filter pattern (see Sect. C.4.2); strategy pattern [39].

### C.4.2 Filter Pattern

#### C.4.2.1 Intent

To produce a sensor reading that has been processed (filtered) in some way, particularly over multiple readings.

#### C.4.2.2 Motivation

A filter modifies a sensor signal either by processing or rejecting data. A filter might maintain the ten latest sensor readings and produce a result based on the mean or median, or ignore spikes in data. Low-pass filters are often used on sensor data from accelerometers and high-pass filters on sensor data from gyroscopes.

#### C.4.2.3 Structure

A class and object diagram for this pattern is given in Fig. C.14. In this pattern, a `Filter` class is introduced that implements the sensor interface (`ISensorReal`). The filter class encapsulates a sensor object. By implementing the sensor interface, a filter object can be passed transparently to the controller such

```

class Voter is subclass of ISensorReal

instance variables
  sensors: seq of ISensorReal; -- aggregate sensors in a sequence
  default: nat1; -- default if no agreement
  inv len sensors = 3; -- supports three sensors

operations
  -- constructor for Voter
  public Voter: seq of ISensorReal * nat1 ==> Voter
  Voter(ss,d) == (
    sensors := ss;
    default := d
  )
  pre len ss = 3 and d <= 3;

  -- get voted value
  public Read: () ==> real
  Read() == (
    -- calculate votes for each pair
    dcl votes: map (nat1 * nat1) to bool :=
      {p |-> close(sensors(p.#1).Read(), sensors(p.#2).Read()) |
        p in set PAIRS};
    -- calculate paths that agree
    dcl agree: seq of (nat1 * nat1) := [];
    for all p in set PAIRS do if votes(p)
      then agree := agree ^ [p];

    -- calculate vote
    if len agree = 3 then (
      -- unanimous
      return sensors(1).Read()
    ) else if len agree = 2 then (
      -- two pairs agree
      dcl diff1: real := abs(sensors(agree(1).#1).Read() -
        sensors(agree(1).#2).Read());
      dcl diff2: real := abs(sensors(agree(2).#1).Read() -
        sensors(agree(2).#2).Read());

      if diff1 < diff2 then return sensors(agree(1).#1).Read()
    else return sensors(agree(2).#1).Read()
    ) else if len agree = 1 then (
      -- one pair agrees
      return sensors(agree(1).#1).Read()
    ) else (
      -- no agreement, use default
      return sensors(default).Read()
    )
  )

functions
  -- return true two values are close, false otherwise
  private close: real * real -> bool
  close(a,b) == abs(a-b) <= EPSILON

values
  -- constants for voting
  PAIRS = {mk_(1,2),mk_(1,3),mk_(2,3)};
  EPSILON = 0.01;

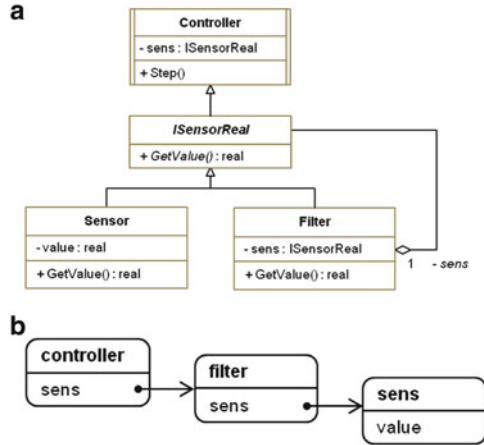
end Voter

```

Listing C.21 Voter class



**Fig. C.14** Filter pattern  
(class and object diagram).  
(a) Class diagram. (b) Object diagram



that the controller does not need to be altered and provides a filtered value of the sensor input. Multiple filters can also be stacked or combined with other sensor patterns, for example, the voter pattern (see Pattern C.4.1). The filter class could maintain multiple readings in a list in order to maintain a number of samples to implement a floating average, for example. The filter pattern could also be combined with the *strategy pattern* [39] by describing the filter algorithm as an interface and providing different implementations of this interface to explore alternative voting routines.

The object diagram in Fig. C.14b shows that the controller holds a reference to the filter object, which then encapsulates the underlying sensor object. When the controller reads its sensor object, the filter decides on the value that the controller receives. Filters could also be added to actuators where necessary (by implementing an actuator interface). This pattern is similar in structure to the voter pattern (see Pattern C.4.1).

#### C.4.2.4 Application to DE Domain

The example `Filter` class below implements an `ISensorReal` interface and holds a reference to another `ISensorReal` object (the sensor to be filtered). In this example, the filter takes a mean reading of up to ten values. Each time the filter is read, a new reading is added to the sequence of samples. Note that this implementation only samples at the speed it is read; it could however be extended to define a periodic thread to sample at a steady rate. Extensions could include storing the samples in sorted order and removing the top and bottom readings, which could smooth out anomalous spikes; or more complex filtering routines such as high-pass filters. Filter algorithms could also be encapsulated as objects by following the *strategy pattern* [39].

```

class Filter is subclass of ISensorReal

instance variables
  private sens: ISensorReal;
  private samples: seq of real

operations
  public Read: () ==> real
  Read() == (
    if len samples < 10 then samples := samples ^ [sens.Read()]
    else samples := t1 samples ^ [sens.Read()];

    return sum(samples) / len samples;
  );

end Filter

```

Listing C.22 Filter class



Fig. C.15 A low-pass filter being used to attenuate the noise from a Gaussian noise source.

#### C.4.2.5 Application to CT Domain

In 20-sim, a the filter pattern is constructed by feeding the output signal of some model into the input of a filter, as shown in Fig. C.15. A small number of band pass filters that operate under the continuous time model are provided within the 20-sim libraries.

If the signal is being processed in a discrete time part of a 20-sim model, then same general structure as above still applies; however, there is no filter provided in the libraries, so the user must construct their own. Figure C.16 shows a rate gyro having its output filtered by a high-pass filter to reduce the effects of drift, which is not uncommon with this type of sensor. The code in the bottom of the diagram represents a high-pass filter, modelled under a discrete time assumption. To function correctly, the engineer must tune the filter by adjusting the value of RC (resistor-capacitor) to suit the sample time and desired cut-off frequency.

For completeness, Fig. C.17 shows an accelerometer being filtered by a low-pass filter to reduce the effects of sudden accelerations. The rate gyro/high-pass filter and accelerometer/low-pass filter may be combined to form part of a complementary sensor system.

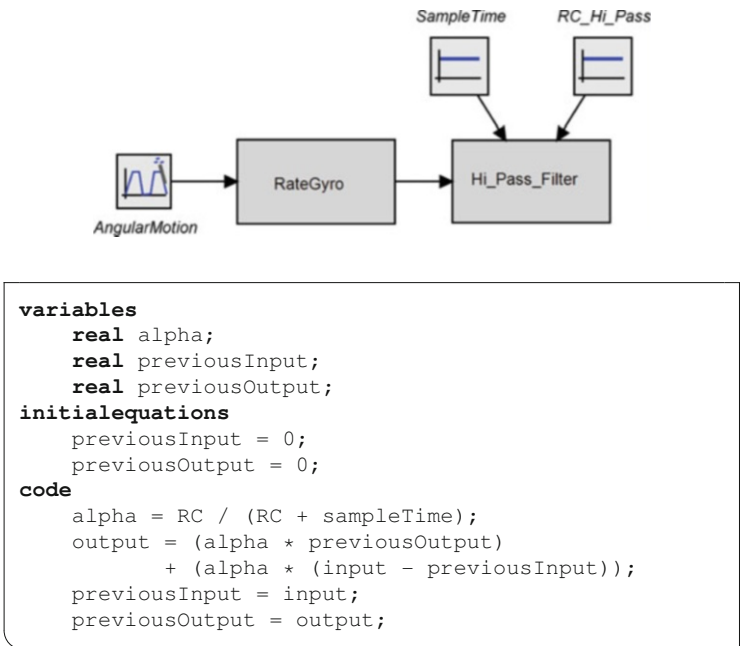


Fig. C.16 Rate gyro component being high-pass filtered (top) and associated code

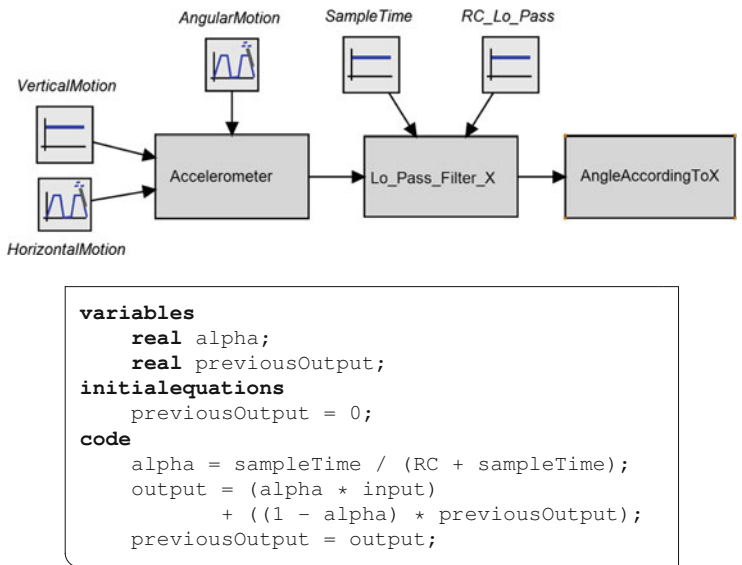


Fig. C.17 Accelerometer component being low-pass filtered (top) and associated code (bottom)

### C.4.2.6 Use in Examples

The filter pattern is used in the *LineMeasurer2Sensor* example.

### C.4.2.7 Related Patterns

Voter pattern (see Pattern C.4.1); strategy pattern [39].

## C.4.3 Kernel Pattern

### C.4.3.1 Intent

A kernel is a small, verifiable component that guarantees some property of a system, typically security or safety [85], by protecting the controller from making unsafe control actions through interception.

### C.4.3.2 Motivation

The kernel is modelled as a class that holds references to the actuators in the system. All calls to the actuators from the controller pass through the kernel, which can ignore them as necessary. In this way, the kernel guards access to the actuators, thus preventing faults of commission [6].

### C.4.3.3 Structure

A class and object diagram for this pattern is given in Fig. C.18. In this pattern, a `Kernel` class is introduced that encapsulates an actuator and implements the actuator interface (`IActuatorReal`). By implementing the interface, a kernel object can be passed transparently to the controller (such that the controller does not need to be altered) and can intercept all calls to the actuator, avoiding unsafe control access when necessary. This structure is clearer in the object diagram, which shows that the controller must access the actuator through the kernel (the variable *value* in the `pwm` object is the shared variable).

The object diagram in Fig. C.18b shows that the controller holds a reference to the kernel object, which in turn holds the actual actuator object and intercepts calls to this actuator. This pattern is similar in intent to the monitor pattern (see Pattern C.4.4).

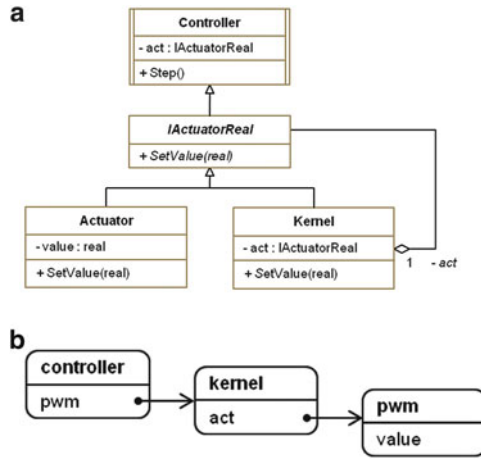


Fig. C.18 Kernel pattern (class and object diagram). (a) Class diagram. (b) Object diagram

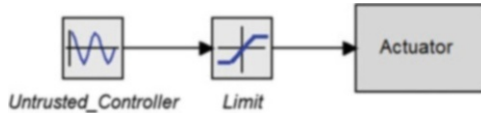


Fig. C.19 A limit function block being used to guarantee a signal will not move out of the range acceptable to an actuator

### C.4.3.4 Application to DE Domain

The example Kernel class below holds a reference to an IActuatorPWM object that it will guard access to. In the example below, the Write operation of the class, this kernel limits the absolute value of the actuator to the limit value passed to the constructor.

### C.4.3.5 Application to CT Domain

In 20-sim, a block may be used to limit the signal of a controller. This could be a *limit* function block from the library or an equation block that uses the in-built *limit* function. An example of the dedicated block is shown in Fig. C.19. A limit block should be the last entity able to modify a signal before it is passed to its intended recipient.

### C.4.3.6 Use in Examples

The kernel pattern is used in the *PaperPinch* example.

```

class Kernel is subclass of IActuatorPWM

types
  -- restricted between -1 and 1
  public PWM = real
  inv p == -1 <= p and p <= 1;

instance variables
  -- actuator
  pwm: IActuatorPWM;

  -- limit of pwm value
  limit: real;
  inv limit > 0.0 and limit < 1.0

operations
  -- constructor for Kernel
  public Kernel: IActuatorPWM * real ==> Kernel
  Kernel(p,l) == (
    pwm := p;
    limit := l
  )
  pre l >= 0 and l < 1;

  -- set actuator value
  public Write: PWM ==> ()
  Write(v) == (
    if abs(v) <= limit then (
      pwm.Write(v); -- take no action
    )
    else ( -- limit value
      if v < 0
      then pwm.Write(-limit)
      else pwm.Write(limit);
    )
  )

end Kernel

```

Listing C.23 Kernel class

### C.4.3.7 Related Patterns

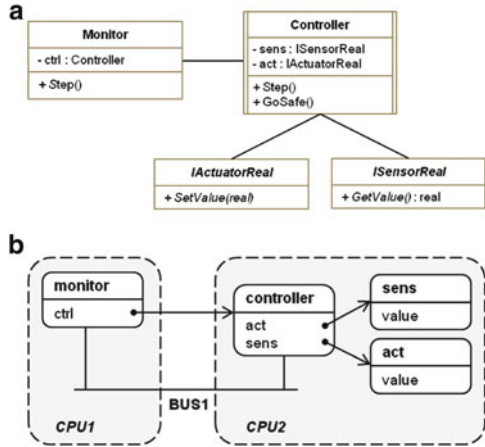
Monitor pattern (see Pattern [C.4.4](#)).

## C.4.4 Monitor Pattern

### C.4.4.1 Intent

A monitor (or watchdog) is a small, verifiable component that runs as separate process. It monitors actions of the controller (or other components) and protects

**Fig. C.20** Monitor pattern (class and object diagram). (a) Class diagram. (b) Object diagram



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

### C.4.4.2 Motivation

The monitor is modelled as an object that holds a reference to the controller and runs as a separate process on a different CPU. The monitor then checks the actions of the controller and intervenes in some way—for example, by calling an operation on controller that puts it in a safe mode. The monitor could also hold references to other objects to monitor them as required, thereby typically limiting its scope to keep it small (as with a kernel).

### C.4.4.3 Structure

A class and object diagram for this pattern is given in Fig. C.20. In this pattern, a `Monitor` class is created that holds a reference to the controller. The `Controller` class provides an operation that the monitor can call if some fault occurs (in this case, the controller will go into a safe mode).

The object diagram in Fig. C.20a shows that the monitor object holds a reference to the controller object and runs on a separate CPU. It can then put the controller into a safe mode if a fault is detected. This pattern is similar in intent to the kernel pattern (see Pattern C.4.3).

### C.4.4.4 Application to DE Domain

The example `Monitor` class below runs as a thread and simply checks if the controller is being “safe” (or simply doing something it should not). In this example,

the decision as to whether the monitor should intervene is not specified. One option is for the monitor to hold references to the actuators and intervene if the controller sets an undesired actuator value; another is for the monitor to look at the sensor readings to detect incorrect behaviour. An example `Monitor` class could be as follows. In order to allow the monitor to intervene, the `Controller` class is assumed to have a `GoSafe` operation.

```

class Monitor

instance variables
  ctrl: Controller

operations
  private Step: () ==> ()
  Step() ==
    if notsafe(ctrl) then ctrl.GoSafe()

functions
  private notsafe: Controller -> bool
  notsafe(ctrl) == ...

thread
  periodic(1E6,1,0,0)(Step); -- 1kHz

end Monitor

```

**Listing C.24** Monitor class

#### C.4.4.5 Use in Examples

The monitor pattern is used in the *chesswayDESTECS* example.

#### C.4.4.6 Related Patterns

Kernel pattern (see Pattern [C.4.3](#)).



# Appendix D

## Abstract Modelling of ChessWay Safety

Marcel Verhoef and Bert Bos

Probably the hardest part of any software design, in particular for novices, is knowing where to begin. In particular, real-time systems are notoriously hard to grasp and the essential properties are difficult to capture with techniques that primary focus on software structure, such as UML. The VDM notation provides a very rich feature set that allows to model a wide variety of application areas from basically any viewpoint, so in principle it should be right for the job. But the basic question remains the same: how do I start?

As with the mathematical model of the ChessWay physics, which was presented in Sect. 7.3.2, the answer is in managing complexity by using abstraction. The core mathematical model was identified by removing complexity and making explicit assumptions about what was left out (e.g., the surface is perfectly horizontal and energy transfer between wheel and surface is considered ideal). Confidence in these initial models is usually gained easily as they are concise and simple to understand, test or even formally verify. As Albert Einstein has put it: “*Make things as simple as possible, but not simpler*”.

---

M. Verhoef (✉)  
Chess WISE, Haarlem, The Netherlands  
e-mail: [Marcel.Verhoef@chess.nl](mailto:Marcel.Verhoef@chess.nl)

B. Bos  
Chess iX, Haarlem, The Netherlands  
e-mail: [bert.bos@chess-ix.com](mailto:bert.bos@chess-ix.com)

There are no rules to what the simplest model exactly constitutes, but when it is easy to convince other stakeholders of (the properties of) your model, this is usually an indication that you are on the right track. It usually takes a few iterations to get right. However, product engineering requires that the assumptions made *are* taken into account in the model, so we need to do more than just create an initial model. Hence, assumptions are removed iteratively by model elaboration. Basically, the model is extended to take a particular feature explicitly into account. This stepwise approach allows to check whether all initial properties still hold after such an elaboration step, which helps to build and maintain confidence in the overall design. Of course, the complexity of the model will increase over time, but the continuous improvement process typically leads to models that themselves are well structured and maintainable. Insight gained usually leads to constant refactoring, typically again using abstractions, which leads to models that even have a certain level of implicit beauty.

VDM supports this interactive way of working by means of prototyping. A very large part of the modelling language is executable, and an interpreter is available to interact with the model. This can be used in the early stages of writing models to validate assumptions, for example, by testing. Nevertheless, the tendency of software engineers is usually to start with structuring or real-time aspects, but in our experience, these are seldom the key design concerns. We propose an approach whereby the powerful features of VDM are gradually introduced in the model to address different design concerns in a stepwise approach:

1. Use basic VDM features such as values, types, instance variables, functions and operations to create an abstract executable model that captures the key design concerns. Typically, we abstract away from time altogether, we focus on the principal (temporal) behaviour of the system and its interfaces to the environment. Consistency requirements are modelled using invariants, pre- and post-conditions.
2. Use VDM object-oriented features to structure the abstract model into several components. These components can then be used to create a true software architecture and design, possibly aligned with a UML representation.
3. Use VDM real-time and distributed architecture features to explore the impact of time and deployment of software components on the overall system design.

This workflow is discussed in detail in [58, 96]. Steps 1 and 2 are covered extensively in the available references [34, 36] respectively. Note that the models resulting from step 3 are typically used for co-simulation and the majority of the models presented in this book are at that level of competence and maturity. However, for the convenience of the reader, we explore step 1 of the ChessWay case study here a bit more in detail to illustrate the point about the use and power of abstract models.

This initial specification focuses on the overall safety of the ChessWay self-balancing scooter. The approach taken here is to identify system interfaces and

properties that affect safety directly. This will help us to partition the model into elements belonging to the plant (the physics) and elements belonging to the controller early on. At a later stage, this information will allow us to define the co-simulation interface much more easily.

Two obvious candidate system interfaces that directly affect safety are the power switch and the safety key. Note that the ChessWay device is in fact always turned on, the power switch merely indicates that the driver wants to actively use the device (soft power on). Two simple enumeration types are used to denote the current status of the power switch and safety key of the ChessWay.

```
types
  PowerSwitch = <ON> | <OFF>;
  SafetyKey   = <IN> | <OUT>
```

An important property of the device is the angle of the pole. In the initial model, we focus on the property itself rather than the sensor used to acquire it. Hence, a real valued number type `PoleAngle` is used to express the current angle of the ChessWay pole, whereby zero denotes upright (vertical). An invariant is used to express the maximum angles for which this model is sound. The function `exceedAngleLimit` can be used to determine whether the ChessWay is within the allowed user envelope, specified here as  $15^\circ$  from upright.

```
types PoleAngle = real
inv pa == pa >= -90 and pa <= 90

functions
  exceedAngleLimit: PoleAngle -> bool
  exceedAngleLimit (pa) == pa < -15 or pa > 15
```

The state of the physical world (plant) output interfaces, which we refer to as *monitored variables* (denoted with a *m*-prefix here), inputs to the discrete controller, can now be defined as follows:

```
instance variables
  mPowerSwitch : PowerSwitch := <OFF>;
  mSafetyKey   : SafetyKey   := <OUT>;
  mPoleAngle   : PoleAngle   := 90
```

Note that this model initially assumes the ChessWay is lying flat on the ground, with power turned off and the safety key removed. One of the key purposes of the controller is to operate the safety override switch. The `SafetyOverride` type is introduced in order to distinguish actuated from free running wheels.

```

types
  SafetyOverride = <FREERUNNING> | <ACTUATED>

instance variables
  cSafetyOverride : SafetyOverride := <FREERUNNING>

```

The physical world (plant) input interface `cSafetyOverride`, which we refer to as a *controlled variable* (denoted with a *c*-prefix here), is initially open to imply undriven wheels. Note that, from the point of view of modelling device safety, it is sufficient to keep the initial model of the wheel and motor control at this very high level of abstraction. At the moment, we do not need to know how much power is provided to the wheels, we merely need to know whether the wheels are actively driven or not. Moreover, if we ignore steering and assume both wheels are identical, it also suffices to consider just a single wheel.

The discrete controller observes the plant state, initiates actions according to any perceived state changes and this process is continuously repeated, usually at some fixed time interval. It is very important to realise that the plant state and the observed controller state actually *differ* due to this sampling behaviour, the controller just sees a snapshot from the continuous plant history. The designer of the discrete control algorithm must therefore take great care to ensure that observation of the plant state is done consistently, preferably only once per iteration, for example, by implementing a sample-and-hold strategy: copying all relevant state variables into local variables before interpreting and using the associated values. This simple insight is typically intuitive for control engineers but foreign to software engineers.

There is no doubt that time is very relevant in control systems. But the tendency of novice modellers is to focus on the real-time aspects while complexity usually does not arise from timing, but rather from the order in which events happen. Therefore, in initial models, we prefer to focus on the temporal behaviour of the system first before we introduce timing aspects. Hence, we abstract away from time altogether here and focus solely on the discrete controller `step` operation. The discrete controller needs to maintain its own state, here identified with the *ctrl*-prefix, and this notion of history allows decision making over consecutive invocations of the `step` operation. These style of controllers are commonly referred to as *state machines*. For the wheel controller, the following states can be identified:

```

types
  OperatingMode = <IDLE> | <CHECK> | <DRIVE>

instance variables
  ctrlOperatingMode : OperatingMode := <IDLE>

```

Operating mode <IDLE> is used as the initial state, while <CHECK> is used whenever the user attempts to keep the ChessWay manually upright for several seconds and finally <DRIVE> is used when the ChessWay is actuated. But how do these controller states relate to device safety?

```

functions
  safeChessWay: PowerSwitch * SafetyKey * PoleAngle -> bool
  safeChessWay (pps, psk, ppa) ==
    pps = <ON> and psk = <IN> and not exceedAngleLimit (ppa)

```

First, we define a function `safeChessWay` that determines whether the system is safe to use. This implies the device is turned on, has the safety key inserted and the pole of the ChessWay is within the specified user envelop of 15° from upright. We can now define the `step` operation of the discrete controller.

```

operations
  step: () ==> ()
  step () ==
    ( dcl
      -- first obtain a local sample of the current plant state
      lPowerSwitch : PowerSwitch := mPowerSwitch,
      lSafetyKey   : SafetyKey   := mSafetyKey,
      lPoleAngle   : PoleAngle   := mPoleAngle;
      -- is the ChessWay still safe?
      if safeChessWay (lPowerSwitch, lSafetyKey, lPoleAngle)
      then update (lPoleAngle)
      else ( -- reset the discrete controller state
        ctrlOperatingMode := <IDLE>;
        -- reset the safety override switch (undriven wheel)
        cSafetyOverride := <FREERUNNING> ) )

```

The `step` operation first obtains a snapshot of the plant by copying the current state values into local variables, indicated by the *l*-prefix. Next, the operation checks overall safety by calling the `safeChessWay` function. The safety override switch is immediately opened and the controller state is reset to <IDLE> in case the device

is unsafe. The operation `update` is called to execute the state machine in case the device is safe.

The implicit hierarchy between the `step` and `update` operations shown here reflects our design decision to abstractly represent two semi-independent processes: the safety monitor, effectively the `step` operation and the motor controller, represented by `update`. The hierarchy reflects the fact that the safety monitor overrules the state machine behaviour of the motor controller. In the implementation, this would probably be realised as independent processes, possibly running in their own execution context, whereby the periodicity of `step` is an order of magnitude faster than `update`.

```

functions
  -- ChessWay within 5 degrees from upright?
  upright: PoleAngle -> bool
  upright (ppa) == ppa >= -5 and ppa <= 5

instance variables
  -- count how often we are in <CHECK> state
  ctrlCheckCount : nat := 0

values
  -- limit to move from <CHECK> to <DRIVE>
  CheckCountLimit = 5

operations
  update: PoleAngle ==> ()
  update (ppa) == (
    cases (ctrlOperatingMode):
      <IDLE> ->
        if upright (ppa)
        then ( ctrlCheckCount := 0;
              ctrlOperatingMode := <CHECK> ),
      <CHECK> ->
        if upright (ppa)
        then ( ctrlCheckCount := ctrlCheckCount + 1;
              if ctrlCheckCount >= CheckCountLimit
              then ( ctrlOperatingMode := <DRIVE>;
                    cSafetyOverride := <ACTUATED> ) )
        else ctrlOperatingMode := <IDLE>,
      <DRIVE> -> skip
    end)

```

The `update` operation implements the discrete controller state machine, which maintains its own history in the `ctrlOperatingMode` instance variable. In `<IDLE>` mode, the controller checks if the `ChessWay` is held upright and if so, it resets `ctrlCheckCount` and moves to the `<CHECK>` state. The `CHECK` state only

moves to the <DRIVE> state if and only if the controller has observed that the ChessWay has been kept upright for at least five consecutive iterations. In <DRIVE> mode, the wheels are actively powered. The state machine is automatically reset to <IDLE> if `safeChessWay` returns false inside the top-level `step` operation.

But how do we know that this model is fit for purpose? The tool suite provides a range of analysis techniques to assess VDM models. In particular, the capability to execute abstract models using the built-in interpreter allows for rapid prototyping and structured testing, once they are shown to be syntax and type correct. This enables early model validation by inspection at relatively low cost as it is very simple to use. Structured testing is, for example, facilitated by the VDMUnit framework and combinatorial testing [57].

However, a straightforward explicit testing approach is used here as we are still in the exploratory phase of model development. Basically, functionality is added to the specification which allows to feed usage scenarios into the state machine and observe its behaviour. First, we define a type `PlantAction` to distinguish between the different events that could be observed. The `PlantBehavior` is defined as the sequence of those observed actions, occurring at a certain iteration, indicated by a natural number. Note that an invariant is added to ensure that the sequence is sound, meaning that the events are listed in a monotone increasing order.

```
types
  PlantAction = PowerSwitch | SafetyKey | PoleAngle;

  PlantBehavior = seq of (nat * PlantAction)
  inv pb ==
    forall i, j in set inds pb &
      let mk_(pi, -) = pb(i), mk_(pj, -) = pb(j) in
        i < j => pi <= pj
```

The `scenario` can now be constructed by hand as a VDM value. First, we turn the power on and insert the safety key at iteration 1 and 2, respectively. Then we move the ChessWay gradually upright and the device should move to <ACTUATED> mode at iteration 12 as we are within the safe operating and upright zones since iteration 7. The handle bar is moved forward until it is beyond the safe operating limit at iteration 16, at which the mode should become <FREERUNNING>. The pole is moved back in the safe zone at iteration 17 and within the upright range at iteration 19 which implies that the mode is again <ACTUATED> at iteration 24. The safety key is removed at iteration 27, causing the mode once more to become <FREERUNNING>. The safety key is inserted again at iteration 30, returning to <ACTUATED> mode at iteration 35. Finally, the ChessWay is turned off at iteration 40.

```

values
  scenario : PlantBehavior = [
    mk_(1, <ON>), mk_(2, <IN>), mk_(3, 60), mk_(4, 30),
    mk_(5, 15), mk_(6, 10), mk_(7, 4), mk_(12, 2),
    mk_(13, 8), mk_(15, 15), mk_(16, 16), mk_(17, 14),
    mk_(19, 4), mk_(27, <OUT>), mk_(30, <IN>),
    -- mk_(32, 6), mk_(33, 4),
    mk_(40, <OFF>)
  ]

```

The operation `validate` simply iterates over the `scenario` and executes the actions as they are due according to the iteration counter. The `step_check` operation is called each iteration, which in turn calls the `step` operation from our model under test and simply prints a diagnostic message containing an overview of all relevant state variables.

```

operations
  step_check: nat ==> ()
  step_check (id) == (
    -- execute the state machine
    step();
    -- generate some diagnostic message
    IO'println([id, mPowerSwitch, mSafetyKey, mPoleAngle,
               ctrlOperatingMode, cSafetyOverride])
  );

  public validate: () ==> ()
  validate () == (
    -- maintain a sequence counter
    dcl cnt : nat := 0;
    -- iterate over the scenario
    for mk_(when, action) in scenario do (
      while cnt < when do (
        -- perform steps without action (state unchanged)
        step_check(cnt);
        cnt := cnt + 1
      );
      -- take the action
      if is_PowerSwitch(action) then mPowerSwitch := action;
      if is_SafetyKey(action) then mSafetyKey := action;
      if is_PoleAngle(action) then mPoleAngle := action;
    );
    -- perform a final step to process last action
    step_check(cnt)
  )

```

The model can now be executed with the `validate` operation as the main entry point, which results in the following output:



```

[0, <OFF>, <OUT>, 90, <IDLE>, <FREERUNNING>]
[1, <ON>, <OUT>, 90, <IDLE>, <FREERUNNING>]
[2, <ON>, <IN>, 90, <IDLE>, <FREERUNNING>]
[3, <ON>, <IN>, 60, <IDLE>, <FREERUNNING>]
[4, <ON>, <IN>, 30, <IDLE>, <FREERUNNING>]
[5, <ON>, <IN>, 15, <IDLE>, <FREERUNNING>]
[6, <ON>, <IN>, 10, <IDLE>, <FREERUNNING>]
[7, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[8, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[9, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[10, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[11, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[12, <ON>, <IN>, 2, <DRIVE>, <ACTUATED>]
[13, <ON>, <IN>, 8, <DRIVE>, <ACTUATED>]
[14, <ON>, <IN>, 8, <DRIVE>, <ACTUATED>]
[15, <ON>, <IN>, 15, <DRIVE>, <ACTUATED>]
[16, <ON>, <IN>, 16, <IDLE>, <FREERUNNING>]
[17, <ON>, <IN>, 14, <IDLE>, <FREERUNNING>]
[18, <ON>, <IN>, 14, <IDLE>, <FREERUNNING>]
[19, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[20, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[21, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[22, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[23, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[24, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[25, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[26, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[27, <ON>, <OUT>, 4, <IDLE>, <FREERUNNING>]
[28, <ON>, <OUT>, 4, <IDLE>, <FREERUNNING>]
[29, <ON>, <OUT>, 4, <IDLE>, <FREERUNNING>]
[30, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[31, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[32, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[33, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[34, <ON>, <IN>, 4, <CHECK>, <FREERUNNING>]
[35, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[36, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[37, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[38, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[39, <ON>, <IN>, 4, <DRIVE>, <ACTUATED>]
[40, <OFF>, <IN>, 4, <IDLE>, <FREERUNNING>]

```

Inspection of the execution output is simple to perform, and the model indeed seems to behave as expected. Of course, when complexity of the model grows, it will also become increasingly more difficult to check these results by hand. This is where VDMUnit and combinatorial testing come to the rescue. They provide the automation required to manage the complexity of testing. In addition, it may be worthwhile to build a simple user interface and connect that to the interpreter executing the model in order to present the myriad of data in ways which are easier to understand and inspect.

Function or operation	Coverage	Calls
exceedAngleLimit	100.0%	105
safeChessWay	100.0%	82
step	100.0%	82
step_check	100.0%	41
update	86.4%	62
upright	100.0%	44
validate	100.0%	1
ChessWay.vdmpp	97.5%	417

**Fig. D.1** Execution coverage statistics

Testing is a great technique to gain confidence in the model produced, but it does not provide any guarantees. For example, we should ask ourselves whether the test suite is complete. Did we exercise the entire specification? This question can be investigated by looking at the execution coverage statistics, as shown in Fig. D.1.

```

operations
  update: PoleAngle ==> ()
  update (ppa) == (
    cases (ctrlOperatingMode):
      <IDLE> ->
        if upright (ppa)
          then ( ctrlCheckCount := 0;
                ctrlOperatingMode := <CHECK> ),
      <CHECK> ->
        if upright (ppa)
          then ( ctrlCheckCount := ctrlCheckCount + 1;
                if ctrlCheckCount >= CheckCountLimit
                  then ( ctrlOperatingMode := <DRIVE>;
                        cSafetyOverride := <ACTUATED> ) )
          else ctrlOperatingMode := <IDLE>,
      <DRIVE> -> skip
    end)

```

The coverage statistics table clearly show that the core operation of the state machine, `update`, is only partly executed. The detailed overview shown above indicates that the outermost `else`-branch in the `<CHECK>` state is missed. This corresponds to the case whereby the ChessWay is kept upright, but not for five iterations continuously. This can be fixed by adding, for example, `mk_(32, 6)` and `mk_(33, 4)` to the scenario.

Test coverage is a very good indicator whether every possible execution path in the model is touched at least once by the test suite. However, it is not a completeness proof as the state of the model is typically infinite, for example, due to the use of real number variables and concurrency. More advanced techniques, such as interactive proof and model checking, can be used to go beyond what is possible with testing. But even these advanced techniques will never solve the problem of

underspecification (what isn't specified can't be machine checked). Hence, a critical attitude towards the model and the analysis results should always be maintained. The pragmatic prototyping approach advocated here is usually instrumental in maintaining a proper design dialogue because insight is gained quickly and results can be communicated back immediately with all stakeholders involved through “*what-if*”-style interaction.

We have shown in this appendix how to create initial models using VDM and how these models can be validated through testing. We have used abstraction to find the *essential* elements of the real-world problem that are needed to describe the key property of interest, in this case safety. This initial model can now be extended to include new features, such as:

- Separate state machines for left- and right wheels
- Add gyroscope, accelerometer and PWM interfaces
- Add a direction switch and implement steering
- Include failure modes of all sensors and actuators
- Operate the ChessWay on a non-ideal surface

This stepwise and iterative approach helps to fight complexity and to maintain overall model consistency. Similarly, real-time and distribution aspects can be introduced gradually in the model.

# References

1. Alexander C, Ishikawa S, Silverstein M (1977) A pattern language: towns, buildings, construction. Oxford University Press, New York
2. Alur R, Courcoubetis C, Halbwachs N, Henzinger TA, Ho PH, Nicollin X, Olivero A, Sifakis J, Yovine S (1995) The algorithmic analysis of hybrid systems. *Theor Comput Sci* 138:3–34
3. Ambrosius F (2007) Modelling and distributed controller design of the BodeRC paper-path setup. Master's thesis, Department of Electrical Engineering, Mathematics and Computer Science, University of Twente, appeared as Technical Report 003CE2007
4. van Amerongen J (2010) Dynamical systems for creative technology. Controllab Products, Enschede
5. Ashenden PJ (2001) The designer's guide to VHDL, 2nd edn. Morgan Kaufmann Publishers, San Francisco
6. Avizienis A, Laprie JC, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans Dependable Secure Comput* 1:11–33
7. Bae K, Ölveczky PC, Feng TH, Tripakis S (2009) Verifying ptolemy ii discrete-event models using real-time maude. In: Proceedings of the 11th international conference on formal engineering methods: formal methods and software engineering, ICFEM '09. Springer, Berlin, pp 717–736
8. Baheti R, Gill H (2011) Cyber-physical systems. In: Samad T, Annaswamy A (eds) The impact of control technology. IEEE Control Society, pp 161–166. Available at [www.ieeecss.org](http://www.ieeecss.org)
9. Baker RE (2005) An approach for dealing with dynamic multi-attribute decision problems. Ph.D. thesis, Department of Computer Science, University of York, UK
10. Banerjee A, Venkatasubramanian KK, Mukherjee T, Gupta SKS (2012) Ensuring safety, security, and sustainability of mission-critical cyber-physical systems. *Proc IEEE* 100(1):283–299. doi:10.1109/JPROC.2011.2165689
11. Banks J, Carson J, Nelson BL, Nicol D (2004) Discrete-event system simulation, 4th edn. Prentice Hall, Upper Saddle River
12. Berkenkötter K, Bisanz S, Hannemann U, Peleska J (2004) Executable hybriduml and its application to train control systems. In: Ehrig H, Damm W, Desel J, Grosse-Rhode M, Reif W, Schnieder E, Westkämper E (eds) SoftSpez Final Report. Lecture notes in computer science, vol 3147. Springer, Berlin, pp 145–173
13. Blochwitz T, Otter M, Akesson J, Arnold M, Clauss C, Elmquist H, Friedrich M, Junghanns A, Mauss J, Neumerkel D, Olsson H, Viel A (2012) The functional mockup interface 2.0: the standard for tool independent exchange of simulation models. In: Proceedings of the 9th international Modelica conference, Munich

14. Bonabeau E (2002) Agent-based modeling: methods and techniques for simulating human systems. *Proc Natl Acad Sci USA* 99(Suppl 3):7280–7287. doi:10.1073/pnas.082080899
15. Booch G, Jacobson I, Rumbaugh J (1999) *The unified modelling language user guide*. Addison-Wesley, Reading
16. Broenink JF (1997) Modelling, simulation and analysis with 20-Sim. *J A Spec Issue CACSD* 38(3):22–25
17. Broenink JF, Ni Y, Groothuis MA (2010) On model-driven design of robot software using co-simulation. In: Menegatti E (ed) *Proceedings of SIMPAR 2010 workshops international conference on simulation, modeling, and programming for autonomous robots*. TU Darmstadt, Darmstadt, pp 659–668
18. Broman D, Derler P, Eidson J (2013) Temporal issues in cyber-physical systems. *J Indian Inst Sci* 93(3):389–402
19. Broy M, Cengarle MV, Geisberger E (2012) Cyber-physical systems: imminent challenges. In: Calinescu R, Garlan D (eds) *Large-scale complex IT systems. Development, operation and management. Lecture notes in computer science*, vol 7539. Springer, Berlin, pp 1–28. doi:10.1007/978-3-642-34059-8
20. Bruun H, Damm F, Hansen BS (1991) An approach to the static semantics of VDM-SL. In: *VDM '91: formal software development methods*, VDM Europe. Springer, Berlin, pp 220–253
21. Cervin A, Henriksson D, Lincoln B, Eker J, Arzen K (2003) How does control timing affect performance? Analysis and simulation of timing using jitterbug and truetime. *IEEE Control Syst* 23(3):16–30. doi:10.1109/MCS.2003.1200240
22. Chiodo M, Giusto P, Jurecska A, Hsieh HC, Sangiovanni-Vincentelli A, Lavagno L (1994) Hardware-software codesign of embedded systems. *IEEE Micro* 14:26–36
23. Christiansen MP, Larsen M, Jørgensen RN (2013) Collaborative model based development of adaptive controller settings for a load-carrying vehicle with changing loads. In: *Bochtis DD, Sørensen CAG (eds) CIOSTA XXXV conference*
24. Coleman JW, Lausdahl KG, Larsen PG (2012) D3.4b—co-simulation semantics. *Tech. Rep., The DESTECS Project (CNECT-ICT-248134)*
25. Corporaal H (2006) Embedded system design. In: *Karelse F (ed) Progress White Papers 2006*. STW, Utrecht, pp 7–27
26. Coverity (2012) *Coverity Scan: 2012 Open Source Report*. *Tech. Rep., Coverity*
27. Dawes J (1991) *The VDM-SL reference guide*. Pitman, London. ISBN 0-273-03151-1
28. DESTECS09 (2009) *DESTECS (Design support and tooling for embedded control software)*. European Research Project
29. Eidson J, Lee E, Matic S, Seshia S, Zou J (2012) Distributed real-time software for cyber-physical systems. *Proc IEEE* 100(1):45–59. doi:10.1109/JPROC.2011.2161237
30. Eker J, Janneck J, Lee E, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity—the tolemy approach. *Proc IEEE* 91(1):127–144
31. European Cooperation for Space Standardization (ECSS) (2009) *ECSS Std ECSS-E-ST-40C Space engineering—software*
32. European Cooperation for Space Standardization (ECSS) (2009) *ECSS Std ECSS-Q-ST-80C Space product assurance—software product assurance*
33. Eveleens JL, Verhoef C (2010) The rise and fall of the chaos report figures. *IEEE Software*, pp 30–36
34. Fitzgerald J, Larsen PG (1998) *Modelling systems—practical tools and techniques in software development*. Cambridge University Press, Cambridge. ISBN 0-521-62348-0
35. Fitzgerald J, Larsen PG (2009) *Modelling systems—practical tools and techniques in software development*, 2nd edn. Cambridge University Press, Cambridge. ISBN 0-521-62348-0
36. Fitzgerald J, Larsen PG, Mukherjee P, Plat N, Verhoef M (2005) *Validated designs for object-oriented systems*. Springer, New York
37. Fitzgerald JS, Larsen PG, Verhoef M (2008) *Vienna development method*. In: *Wah B (ed) Wiley encyclopedia of computer science and engineering*. Wiley, Chichester

38. Fritzon P, Engelson V (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, Berlin, pp 67–90
39. Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns. Elements of reusable object-oriented software. Addison-Wesley professional computing series. Addison-Wesley, Reading
40. Gupta SK, Mukherjee T, Varsamopoulos G, Banerjee A (2011) Research directions in energy-sustainable cyber-physical systems. *Sustain Comput Inform Syst* 1(1):57–74
41. Hardebolle C, Boulanger F (2009) Exploring multi-paradigm modeling techniques. *SIMULATION Trans Soc Model Simul Int* 85(11/12):688–708
42. Heemels M, Muller G (2007) Boderc: model-based design of high-tech systems, 2nd edn. Embedded Systems Institute, Eindhoven
43. IEEE (2000) IEEE 100 the authoritative dictionary of IEEE standards terms, 7th edn. IEEE Std 100-2000. doi:10.1109/IEEESTD.2000.322230
44. IEEE (2008) 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
45. IEEE (2008) 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
46. Jackson D (2009) A direct path to dependable software. *Commun ACM* 52(4):78–88. doi:10.1145/1498765.1498787
47. Jensen J, Chang D, Lee E (2011) A model-based design methodology for cyber-physical systems. In: 2011 7th international wireless communications and mobile computing conference (IWCMC), pp 1666–1671. doi:10.1109/IWCMC.2011.5982785
48. Johnson CW (2005) The natural history of bugs: using formal methods to analyse software related failures in space missions. In: Fitzgerald J, Hayes IJ, Tarlecki A (eds) FM 2005: formal methods. Lecture notes in computer science, vol 3582. Springer, Berlin, pp 9–25
49. Johnson J (2006) My life is failure. Standish Group International, co-author of the original 1994 CHAOS report
50. Jones CB (1990) Systematic software development using VDM, 2nd edn. Prentice-Hall International, Englewood Cliffs. ISBN 0-13-880733-7
51. JPL Special Review Board (2000) Report on the loss of the Mars Polar Lander and Deep Space 2 missions. Tech. Rep. JPL D-18709. Jet Propulsion Laboratory
52. Karnopp D, Rosenberg R (1968) Analysis and simulation of multiport systems: the bond graph approach to physical system dynamic. MIT Press, Cambridge
53. Kleijn C (2009) 20-sim 4.1 reference manual, 1st edn. Controllab Products B.V., Enschede. ISBN 978-90-79499-05-2
54. Kleijn C, Visser P, Groen F (2012) D3.5—extension to Matlab/Simulink. Tech. Rep., The DESTECs Project (CNECT-ICT-248134)
55. Kopetz H, Bauer G (2003) The time-triggered architecture. *Proc IEEE* 91(1):112–126
56. Larsen PG, Battle N, Ferreira M, Fitzgerald J, Lausdahl K, Verhoef M (2010) The overture initiative—integrating tools for VDM. *SIGSOFT Softw Eng Notes* 35(1):1–6
57. Larsen PG, Lausdahl K, Battle N (2010) Combinatorial testing for VDM. In: Proceedings of the 2010 8th IEEE international conference on software engineering and formal methods, SEFM '10. IEEE Computer Society, Washington, pp 278–285. ISBN 978-0-7695-4153-2
58. Larsen PG, Wolff S, Battle N, Fitzgerald J, Pierce K (2010) Development process of distributed embedded systems using vdm. Tech. Rep. TR-2010-02, The Overture Open Source Initiative
59. Larsen PG, Lausdahl K, Battle N, Fitzgerald J, Wolff S, Sahara S (2013) VDM-10 language manual. Tech. Rep. TR-001, The Overture Initiative
60. Larsen PG, Lausdahl K, Coleman J, Wolff S, Kleijn C, Groen F (2013) Crescendo tool support: user manual. Tech. Rep. TR-001, The Crescendo Initiative

61. Lausdahl K, Coleman JW, Larsen PG (2013) Semantics of the VDM real-time dialect. ECE-TR-13, Aarhus University, Aarhus, April 2013
62. Lee E, Seshia S (2011) Introduction to embedded systems, a cyber-physical systems approach. University of Berkeley, Berkeley. ISBN 978-0-557-70857-4
63. Lee EA (2008) Cyber physical systems: design challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley
64. Lee EA (2009) Computing needs time. *Commun ACM* 52(5):70–79
65. Lee EA (2010) CPS foundations. In: Proceedings of the 47th design automation conference, DAC '10. ACM, New York, pp 737–742. doi:10.1145/1837274.1837462
66. Lee I, Sokolsky O, Chen S, Hatcliff J, Jee E, Kim B, King A, Mullen-Fortino M, Park S, Roederer A, Venkatasubramanian K (2012) Challenges and research directions in medical cyber-physical systems. *Proc IEEE* 100(1):75–90. doi:10.1109/JPROC.2011.2165270
67. Lions JL, Lübeck L, Fauquembergue JL, Kahn G, Kubbat W, Levedag S, Mazzini L, Merle D, O'Halloran C (1996) ARIANE 5—flight 501 failure—report by the inquiry board. Tech. Rep., European Space Agency
68. Liu J (1998) Continuous time and mixed-signal simulation in ptolemy ii. Tech. Rep. UCB/ERL M98/74, EECS Department, University of California, Berkeley
69. Magureanu G, Gavrilesco M, Pescaru D (2013) Validation of static properties in unified modeling language models for cyber physical systems. *J Zhejiang Univ Sci C* 14(5):332–346. doi:10.1631/jzus.C1200263
70. Maier MW (1996) Architecting principles for systems-of-systems. In: Sixth international symposium of the international council on systems engineering, INCOSE
71. Margaria T, Schätz B, Verhoef M (2006) Formal methods going mainstream: costs, benefits, experiences. *BCS-FACS FACTS* 2006(2):34–38, report on the ForTIA Industry Day at FM'05
72. Marwedel P (2010) Embedded system design—embedded systems foundations of cyber-physical systems. Springer, Berlin
73. Mazzara M, Bhattacharyya A (2010) On modelling and analysis of dynamic reconfiguration of dependable real-time systems. In: 2010 third international conference on dependability (DEPEND), pp 173–181. doi:10.1109/DEPEND.2010.33
74. Miclea L, Sanislav T (2011) About dependability in cyber-physical systems. In: Design test symposium (EWDTS), 2011 9th East-West, pp 17–21. doi:10.1109/EWDTS.2011.6116428
75. Moore GE (1965) Cramping more components onto integrated circuits. *Electronics* 38(8):114–117
76. Nielsen CB (2010) Dynamic reconfiguration of distributed systems in VDM-RT. Master's thesis, Aarhus University
77. Plotkin GD (1981) A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University
78. Plotkin GD (2004) A structural approach to operational semantics. *J Logic Algebraic Program* 60–61:17–139
79. Ptolemaeus C (ed) (2014) System design, modeling, and simulation using ptolemy II. Ptolemy.org
80. Pumfrey D (1999) The principled design of computer system safety analyses. Ph.D. thesis, Department of Computer Science, University of York
81. Rajkumar R, Lee I, Sha L, Stankovic J (2010) Cyber-physical systems: the next computing revolution. In: Design automation conference (DAC), 2010 47th ACM/IEEE, pp 731–736
82. Rational Software Corporation (1998) Rational unified process—best practices for software development teams
83. Robinson S (2004) Simulation: the practice of model development and use. Wiley, New York
84. Romanovsky A, Thomas M (eds) (2013) Industrial deployment of system engineering methods providing high dependability and productivity. Springer, Berlin. ISBN 978-3-642-33169-5
85. Rushby J (1989) Kernels for safety? In: Safe and secure computing systems, Blackwell Scientific Publications, Oxford, pp 210–220

86. Safety and Health Council of the Chemical Industries Association Ltd (1977) A guide to hazard and operability studies
87. Friedenthal S, Moore A, Steiner R (2011) A practical guide to SysML, 2nd edn. Morgan Kaufmann OMG Press, Waltham. ISBN: 978-0-12-385206-9
88. Sangiovanni-Vincentelli A (2006) Successive refinements of communication functions and architectures in system design. In: Design automation and test in Europe, hot topic session—network the next “Big Idea” in design?
89. Sanwal M, Hasan O (2013) Formal verification of cyber-physical systems: coping with continuous elements. In: Murgante B, Misra S, Carlini M, Torre C, Nguyen HQ, Taniar D, Apduhan B, Gervasi O (eds) Computational science and its applications—ICCSA 2013. Lecture notes in computer science, vol 7971. Springer, Berlin, pp 358–371. doi:10.1007/978-3-642-39637-39
90. Schirmer G, Erdogmus D, Chowdhury K, Padir T (2013) The future of human-in-the-loop cyber-physical systems. *Computer* 46(1):36–45
91. Sztipanovits J, Koutsoukos X, Karsai G, Kottenstette N, Antsaklis P, Gupta V, Goodwine B, Baras J, Wang S (2012) Toward a science of cyber-physical system integration. *Proc IEEE* 100(1):29–44. doi:10.1109/JPROC.2011.2161529
92. Taguchi G (1987) System of experimental design, vols 1 and 2. UNIPUB/Krass International Publications, New York
93. Thomas D, Moorby P (2008) The Verilog hardware description language, 5th edn. Springer, Berlin
94. Trapp M, Schneider D, Liggesmeyer P (2013) A safety roadmap to cyber-physical systems. In: Münch J, Schmid K (eds) Perspectives on the future of software engineering. Springer, Berlin, pp 81–94. doi:10.1007/978-3-642-37395-4\_6
95. Vangheluwe HL, de Lara J, Mosterman PJ (2002) An introduction to multi-paradigm modelling and simulation. In: Barros F, Giambiasi N (eds) Proceedings of the AIS’2002 conference (AI, Simulation and Planning in High Autonomy Systems), Lisboa, Portugal, pp 9–20
96. Verhoef M (2009) Modeling and validating distributed embedded real-time control systems. Ph.D. thesis, Radboud University Nijmegen
97. Verhoef M, Bos B, van Eijk P, Remijnse J, Visser E, De Paepe M, De Witte Y, Rombaut K, Van Lembergen R (2012) Industrial case studies—final report. DESTTECS Deliverable D4.3, The DESTTECS Project (CNECT-ICT-248134)
98. Wan K, Hughes D, Man KL, Krilavicius T (2010) Composition challenges and approaches for cyber physical systems. In: 2010 IEEE international conference on networked embedded systems for enterprise applications (NESEA), pp 1–7. doi:10.1109/NESEA.2010.5678065
99. Wang G, Liu Q, Wu J (2010) Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In: Proceedings of the 17th ACM conference on computer and communications security. ACM, New York, pp 735–737
100. Woodcock J, Larsen PG, Bicarregui J, Fitzgerald J (2009) Formal methods: practice and experience. *ACM Comput Surv* 41(4):1–36



# Glossary

- abstract class** (In object-oriented programming) A class where one or more methods are defined abstractly using the text **is subclass responsibility** as their body.
- actuator** A component that produces a physical output in response to a signal [43].
- aggregate** (In object-oriented programming) The act of bringing together several objects into a single whole.
- automated co-model analysis** Tool support for the selection of a single design from a set of design alternatives (including definition of scenarios, execution of co-simulations, and visualisation and analysis of co-simulation results).
- automated co-model execution** As automated co-model analysis except that it does not perform any analysis of the test results produced by the simulations
- bond** (In bond graphs) A directed point-to-point connection between power ports on submodels. Represents the sharing of both *flow* and *effort* by those ports.
- bond graph** A domain-independent idealised physical model based on the representing energy and its exchange between submodels.
- causality** (In bond graphs) Dictates which variable of a power port is the input (cause) for submodel's equations and which is the output (effect).
- class** (In object-oriented programming) The definition of the data field and methods an object of that class will contain.
- code generation** The process of implementing a system controller by automatically translating a model into a representation (in some programming language) which can then be executed on the real hardware of the system.
- co-model** A model comprising two constituent models (a DE submodel and a CT submodel) and a contract describing the communication between them.
- consistency** A co-model is consistent if the constituent models are both syntactically and semantically consistent.
- constituent model** One of the two submodels in a co-model.
- continuous-time simulation** A form of simulation where “the state of the system changes continuously through time” [83, p. 15].

- contract** A description of the communication between the constituent models of a co-model, given in terms of shared design parameters, shared variables and common events.
- controlled variable** A variable that a controller changes in order to perform control actions.
- controller** The part of the system that controls the plant.
- controller architecture** The allocation of software processes to CPUs and the configuration of those CPUs over a communications infrastructure.
- co-sim launch** The type of debug configuration used in the Crescendo tool to define and launch a single scenario.
- co-simulation baseline** The set of elements (co-model, scenario, test results, etc.) required to reproduce a specific co-simulation.
- co-simulation engine** A program that supervises a co-simulation.
- co-simulation** The simulation of a co-model.
- cost function** A function which calculates the “cost” of a design.
- debug config** (Eclipse term) The place in Eclipse where a simulation scenario is defined.
- design alternatives** Where two or more co-models represent different possible solutions to the same problem.
- design parameter** A property of a model which affects its behaviour, but which remains constant during a given simulation.
- design pattern** Is a general reusable solution to a commonly occurring problem within a given context (in this book in a co-modelling setting).
- design space exploration** The (iterative) process of constructing co-models, performing co-simulations and evaluating the results in order to select co-models for the next iteration.
- design step** A co-model which is considered to be a significant evolution of a previous co-model.
- discrete-event simulation** A form of simulation where “only the points in time at which the state of the system changes are represented” [83, p. 15].
- disturbance** A stimulus that tends to deflect the plant from desired behaviour.
- edges** (In bond graphs) See *bond*.
- effort** (In bond graphs) One of the variables exposed by a power port. Represents physical concepts such as electrical voltage, mechanical force or hydraulic pressure.
- environment** Everything that is outside of a given system.
- error** Part of the system state that may lead to a failure [6].
- event** An action that is initiated in one constituent model of a co-model, which leads to an action in the other constituent model.
- executable model** A model that can be simulated.
- failure** A system’s delivered service deviates from specification [6].
- fault injection** The act of triggering faulty behaviour during simulation.
- fault modelling** The act of extending a model to encompass faulty behaviours.
- fault** The adjudged or hypothesised cause of an error [6].

- fault behaviour** A model of a component's behaviour when a fault has been triggered and emerges as a failure to adhere to the component's specification.
- fault-like phenomena** Any behaviour that can be modelled like a fault (e.g. disturbance).
- flow** (In bond graphs) One of the variables exposed by a power port. Represents physical concepts such as electrical current, mechanical velocity, fluid flow.
- ideal behaviour** A model of a component that does not account for disturbances.
- inheritance** (In object-oriented programming) The mechanism by which a subclass contains all public and protected data fields and methods of its superclass.
- input** A signal provided to a model.
- interface** (In object-oriented programming) A class which defines the signatures of but no bodies for any of its methods. Should not be instantiated.
- junction** (In bond graphs) A point in a bond graph where the sum of flow (1-junction) or effort (0-junction) of all bonds to that point is zero.
- log** Data written to a file during a simulation.
- metadata** Information that is associated with, and gives information about, a piece of data.
- model base** The collection of artefacts gathered during a development (including various models and co-models; scenarios and test results; and documentation).
- model management** The activity of organising co-models within a model base.
- model structuring** The activity of organising elements within a model.
- model synthesis** See **code generation**.
- model** A more or less abstract representation of a system or component of interest.
- modelling** The activity of creating models.
- monitored variable** A variable that a controller observes in order to inform control actions.
- object** (In object-oriented programming) An instantiation of a class, contains data fields and methods.
- objective function** See **cost function**.
- operation** (In object-oriented programming) Defines an operation that an object may perform on some data. Operations may be *private*, *public* or *protected*.
- output** The states of a model as observed during (and after) simulation.
- non-normative behaviour** Behaviour that is judged to deviate from specification.
- pattern** (In VDM) Are like empty shells that have a meaning only once they are matched against a concrete value.
- physical concept** (In bond graphs) A class of component or phenomena that could exist or be observed in the real world, e.g. an electrical resistor or mechanical friction.
- plant** The part of the system which is to be controlled [43].
- power port** (In bond graphs) The port type connected in a bond graph. Contains two variables, *effort* and *flow*. A power port exchanges energy with its connected models.
- private** (In object-oriented programming, VDM) The method or data field may only be accessed from within the containing class.

- protected** (In object-oriented programming, VDM) The method or data field may only be accessed by its containing class or any of its subclasses.
- public** (In object-oriented programming, VDM) The method or data field may be accessed by any other class.
- ranking function** A function that assigns a value to a design based on its ability to meet requirements defined by the engineer.
- realistic behaviour** A model of a component which includes disturbances defined by the tolerances associated with that component.
- response** A change in the state of a system as a consequence of a stimulus.
- scenario** Test of a co-model.
- signal domain** Where models share a single value or array at each uni-directional port, unlike bond graphs where the ports are bi-directional.
- sensor** A component whose input is a physical phenomenon and whose output is a quantitative measure of the phenomenon.
- shared design parameter** A design parameter that appears in both constituent models of a co-model.
- shared variable** A variable that appears in and can be accessed from both constituent models of a co-model.
- simulation** Symbolic execution of a model.
- semantically consistent** The state when the constituent models of a co-model agree on the semantics of the variables, parameters and events they share. The nature of these semantics is not yet described.
- static analysis** A method for checking some property of a model without executing that model.
- state event** An event triggered by a change within a model.
- stimulus** A phenomenon that effects a change in the state of a system.
- subclass** (In object-oriented programming) A class that is defined as extending another class. The other class becomes its superclass. The subclass inherits all non-private data fields and methods.
- submodel** A distinct part of a larger model.
- superclass** (In object-oriented programming) The class from which a subclass is defined.
- system boundary** The common frontier between a system and its environment.
- System Under Test (SUT)** The part of a model that represents the system we wish to build as opposed to parts of the model which are not part of this system.
- system** An entity that interacts with other entities, including hardware, software, humans and the physical world [6].
- test result** A record of the output from a simulation of a model (see also **log**).
- time event** An expected event that occurs at a predetermined time.
- variable** Part of a model that may change during a given simulation.
- vertices** (In bond graphs) The joining points of bonds. May be manifested as either a *junction* or a submodel.

# Index

The bold entries refer to defining occurrences of indexed terms, while the normal entries refer to normal occurrences of the indexed terms.

- actuator, **16**, 47, 57, 85, 121, 147, 163, 164, 175, 228, 236, 258, 377
- aggregate, **348**, 348, 349, 377
- algebraic loop, **29**, 150, 308
- and**, **66**, 67, 80, 130, 313
- approximation, **17**, 134, 150, 171, 174–177, 244
- argument, **68**, 90, 246
- association, **88**, 91, 159
- assumption, 8, **12**, 134, 144, 240, 252, 352, 359
- async**, **95**, 247, 249, 350
- automated co-model analysis, 21, 199, **200**, 202, 377
  - folder launch configuration, **210**, 213
- behaviour
  - faulty, **186**, 192, 195, 344, 379
  - ideal, 24, **186**, 191, 379
  - realistic, **186**, 192, 380
- binding, **77**, 83, 84, 319
- bond graph, 17, **37**, 37–48, 58, 165–167, 169, 240, 241, 306, 377
- bool**, **66**, 66, 67, 70, 108, 109, 313, 315
- BUS, **93**, 94, 179, 195, 300, 319, 341
- button
  - debug, 104, **105**, 108, 210
  - pause, **106**
  - play, **106**
  - stop, **106**
- capacitor, **30**, 32
- card**, **76**, 80, 313
- causal conflict, **45**, 377
- causal constraint, **41**, 377
- causal relation diagram, **27**, 377
- causal stroke, **39**, 377
- char**, **73**, 176, 320
- ChessWay, **145**, 145–148, 151, 153, 177–181, 194–196, 223, 225, 253, 263–265, 270, 359
- class, **87**
  - abstract, **117**, 117, 118, 120, 123, 128, 377
  - diagram, **88**, 124, 125, 128, 163, 180, 190, 192, 194, 195, 229, 321, 348, 349, 354
- class**, **87**, 92, 118–120, 125, 135, 179, 181, 320, 349, 351, 355, 358
- code**, **104**
- code generation, 17, **310**, 377
- comment, **64**, 99
- co-model, **17**, 203, 219, 269, 294, 377
  - boundary, **132**, 133, 137
  - contract, 18, 19, **98**, 155, 172, 182, 378
- competent model, **16**, 37, 148, 157, 166, 297
- compliance, **29**, 30
- component, **28**, 34, 42, 46
- conc**, **315**
- concurrency, 87, **91**, 126
- configuration, **279**, 280
- constructor, **90**, 95, 176, 179, 246, 319, 349, 355
- continuous-time system, **51**
- contract, 18, 19, **98**, 378
- contract-first, 25, 155, **182**
- controlled variable, **65**, 163, 164, 274, 305, 378

- controller, 16, 28, 47, 48, **49**, 49–54, 56, 57, 72, 88, 113, 143, 172, 181, 194, 196, 214, 228, 234, 236, 258, 355, 378
  - feedback, **49**
  - feedforward, **49**
  - loop, **50**, **57**, 58, 63, 133, 171, 245, 246
  - mode, **194**, 194, 331
  - sequence, 57, **171**, 245, 249
  - supervisory, 57, 245, 250
- converter, **51**, 51, 52, 55, 57, 190, 192
- co-simulation, 17, 19, 58, **104**, 111, **131**, 233, 256, 281, 283, 299, 310, 378
- CPU, **93**, 94, 179, 195, 201, 228, 300, 319, 341
- CT-first, 25, 154, **164**, 167
- CT-only simulation, **133**, 377
- cyber-physical system, 132, **293**, 295, 297
- cycles**, **95**, 299
  
- damper, **29**, 30
- debug configuration, **104**, 200, 378
- DE-first, 25, 154, **171**, 244, 253
- DE-only simulation, 131, **134**, 136, 137, 378
- design
  - alternative, 21, 166, 199, 206, **207**, 215, 268, 294, 297, 378
  - parameter, **17**, 378
  - pattern, 116, **121**, 189, 190, 195, 323, 378
  - space, **21**, 199, 212
  - space exploration, 21, 166, **199**, 236, 378
- DESTECS, 310
- differential equation, **36**, 46, 149, 228
- dinter**, **313**
- discrete-time system, **51**
- document handling, 223, **238**, 265
- dredging excavator, 223, **225**, 264
- dunion**, **313**
- duration**, **94**, 134, 299
- duty cycle, **48**
- dynamic equilibrium, **35**
  
- editor, **98**, 308
- effort variable, **38**, 378
- elementary DC-motor, **42**
- elementary model, **28**, 46
- elems**, **78**, 315
- embedded system, **3**, 4, 6–10, 13–17, 293–296, 301–303
- environment, **16**, 160, 171, 174, 176, 378
- equations**, 227, 305, 307, 347
- equilibrium
  - dynamic, **35**
  - static, **35**
  
- error, **185**, 378
- event, 18, **56**, 98, 378
  - state, 18, **56**
  - time, 18, **56**, 380
- exists**, **319**
- experiment, 166, 203, **207**, 211, 213, 216, 233, 234
  - fractional factorial, **208**, 218
  - iterative approach, **215**
  - orthogonal matrix, **209**, 213
  - result, 107, **202**, 217, 218
  - screening, **208**
  - space filling search, **210**
- explorer, **98**
- export**, **103**, 171, 310
- externals**, **103**
  
- failure, **185**, 378
- fault, **185**, 378
  - activation, **185**
  - block, **189**
  - identification, **186**
  - injection, **185**, 283, 345, 378
  - modelling, **185**, **188**, 253, 378
  - selection, **188**
  - simulation, **185**
  - tolerance, 10, 12, **185**, 253, 347
    - coverage, **190**
- fixed effort-out causality, **40**
- fixed flow-out causality, **40**
- flow variable, **38**, 379
- fluid storage, **31**
- FMEA, **188**
- forall**, 80, **84**, 248, 319, 365
- function, **68**, 68, 318, 360, 361
  
- global**, **103**, 103, 171
- guideword, **186**, 190, 194
  - late, **195**
  - less, **187**
  - more, **187**
- gyrator, **42**, 43
  
- hard real time, **54**
- HAZOP, **186**
- hd**, **78**, 315
- Hooke's law, **29**, 30
- hydraulic resistance, **31**, 237, 306, 378
  
- iconic diagram, **34**, 306
- ideal physical model, **28**, 34, 46, 165

- implementation edit, **133**
- import**, **98, 103, 171, 310**
- inds**, **78, 80, 315**
- inductance, **30, 110, 144, 242**
- inertia, **30, 33, 42–46, 110, 144, 227, 228, 234, 242**
- inference rule, **281**
- inheritance, **117, 118, 125, 189, 379**
- in set**, **77, 313**
- instance variable, **64, 64, 68, 70–72, 74, 83, 85, 87, 88, 91, 117–120, 124–131, 135, 136, 320, 342, 349, 351, 355, 358, 360**
- int**, **63, 313, 320**
- integration method, **17, 21, 173, 175, 309**
- inter**, **77, 313**
- interface, **46, 57, 87, 103, 116, 119, 120–124, 126, 131, 132, 135, 136, 379**
- inverse**, **315**
- is subclass of**, **120, 125, 135**
- is subclass responsibility**, **117, 175**
  
- jitter, **55, 91, 126, 127, 148**
- junction, **38, 379**
  
- launch configuration, **104, 210, 378**
- len**, **78, 315**
- let expression, **69, 83, 317**
  
- mapping, **81**
  - comprehension, **317**
  - distributed merge, **315**
  - domain
    - restriction, **315**
    - subtraction, **315**
  - empty, **81, 315**
  - enumeration, **315**
  - inverse, **315**
  - maplet, **315**
  - override, **315**
  - range, **315**
    - restriction, **315**
    - subtraction, **315**
  - type, **177, 315**
  - union, **315**
- map to**, **176, 315, 320, 349**
- mass, **29, 144, 148, 149, 161**
- mass-spring-damper system, **34**
- matlab/simulink, **296**
- matrix**, **99**
  
- merge**, **315**
- model**, **16**
  - competent, **16, 37, 148, 157, 166, 297**
  - consistency, **18, 377**
  - constituent, **17, 377**
  - continuous time, **17, 28**
  - discrete event, **17, 62**
  - executable, **16, 378**
- model**, **102**
- modelica, **296**
- modulated voltage source, **47, 49**
- monitored variable, **63, 64, 163, 164**
- motor constant, **42, 110, 111, 144**
- munion**, **315**
- mutex**, **92, 247**
  
- nat**, **63, 313, 315**
- nat1**, **63, 313, 315**
- new**, **89, 119, 320**
- Newton's law, **29, 30, 161**
- nil**, **91, 118, 130, 136, 249, 250**
- not**, **66, 313**
- not in set**, **313**
  
- object diagram, **123, 124, 125, 180, 190, 195, 321, 348, 349, 354, 357**
- Ohm's law, **31**
- once**, **108, 288**
- operation, **70**
  - asynchronous, **95, 176, 247, 249, 250**
  - override, **117, 189, 197**
- or**, **66, 313**
- outline, **98**
- output**, **102, 103**
  
- parameter, **53, 54, 110, 111, 114, 202, 205, 206, 208, 210, 212, 214, 219, 251**
  - continuous, **206**
  - design, **206**
  - discrete, **207**
- parameters**, **103, 307**
- pattern
  - decorator, **122, 124**
  - ether, **195, 300, 341**
  - fault injector, **189, 344, 345**
  - filter, **194, 349**
  - kernel, **354**
  - matching, **83, 316, 365, 379**
  - monitor, **357**
  - noise, **343**
  - voter, **190, 191, 347**

- periodic**, 72, 82, 90, **91**, 126, 127, 177, 247, 319, 358
- permission predicate, 87, 91, **93**, 319
- PID, **51**, 52, 71, 118, 124, 130, 132, 134, 160, 305
- plant, **16**, 134, 171, 246, 379
- post-condition, **79**, 117
- power, **33**
  - bond, **37**
  - port, **42**, 379
- power**, **313**
- pre-condition, **70**, 79, 84, 117, 248, 249, 349
- predicate, **66**, 77, 81, 319
- preferred effort-out causality, **39**
- preferred flow out causality, **39**
- private**, **89**, 89, 118, 127, 176, 177, 179, 189, 246–248, 349, 351, 358, 379
- protected**, **118**, 118, 119, 126, 128, 130, 134, 181, 189, 320, 380
- psubset**, **313**
- ptolemy, **296**
- public**, **89**, 90, 92, 95, 118–120, 125, 130, 131, 136, 137, 246–250, 320, 380
- purpose, 16, 62, 75, **157**, 177, 206, 207, 254
- PWM, **48**, 134, 226, 239, 242, 246
  
- quantifier, **84**, 248, 318, 365
- quote value, **74**, 74, 136, 361, 362, 366
  
- ranking, 204, **216**, 219, 222, 380
  - enumeration and scoring, **217**, 220
  - weighted additive method, **217**, 220
- real**, **63**, 313
- record type, **75**, 76, 316
- recursion, 285, **318**, 318
- resistance, **31**, 110, 144, 242
- resistor, **31**, 142, 207, 379
- RESULT**, **80**, 320
- rng**, **315**
- R2-G2P, **142**, 153, 190, 194, 202, 219
- rule schemata, **279**
  
- safety, 84, **85**, 92, 146, 147, 229, 232, 235, 254, 256, 258, 302, 358, 361–363, 365, 369
- sampling, 52, **55**, 55, 56
- scenario, 19, **20**, 104, 108, 114, 148, 172, 177, 178, 202, 380
- script, **20**, 108, 172, 202
- self**, 178, **320**, 334
- semantic constraint, **275**
  
- sensor, **16**, 28, 47, 57, 120, 147, 148, 151, 163, 164, 169, 175, 177, 179, 190, 191, 194, 203, 219, 228, 258, 349, 380
- sequence, **77**
  - comprehension, **78**, 317
  - concatenation, **78**, 78, 315
    - distributed, **315**
  - elements, **78**, 315
  - empty, **77**, 85, 285, 313
  - head, **78**, 315
  - indices, **80**, 315
  - length, **78**, 315
  - tail, **78**, 315
  - value, **77**, 248
- set, **76**
  - cardinality, **76**, 313
  - comprehension, **77**, 317
  - difference, **313**
  - distributed union, **313**
  - empty, **76**, 84, 313
  - intersection, **77**, 313
    - distributed, **313**
  - membership, **77**, 313
  - power set, **313**
  - proper subset, **313**
  - subset, **313**
  - type, **76**, 79, 84
  - union, **77**, 313
    - distributed, **313**
- setpoint, 49, 51, **53**, 53, 57, 69, 70, 75, 76, 78, 79, 81–83, 246
- setpoint profile, 245, 247, **248**, 249
- setting, 20, 104, **107**, 202
- SHARD, **186**, 190, 194
- shared design parameter, 18, **98**, 102, 201, 203, 380
- shared variable, **17**, 380
- soft real time, **54**
- software development standard, **23**, 212
- spring, **29**, 30
- spring constant, **29**, 30
- start**, **92**, 178, 319
- statement, **70**
  - assignment, 71, 72, 82, 85, 90, 118, 126–128, 130, 131, 134, 137, 363, 364, 366
  - block, **71**, 72, 90, 92, 127, 128, 130, 131, 134, 246, 249, 363, 364, 366
  - cases, 136, 179, **316**, 364, 368
  - loop, **317**, 320
  - return, **71**, 71, 119, 136, 179, 232, 248
  - skip, **93**, 128, 134, 320, 364, 368
- static**, **89**, 94, 136, 179, 326, 329
- static equilibrium, **35**



- stimuli, **16**, 134, 171
- structural operational semantics, **278**
- subclass, **117**, 135, 189, 196, 321, 351, 380
- submodel, **34**, 170, 202, 228, 241, 307, 380
- subset**, **313**
- superclass, **117**, 189, 380
- sweep, 201, **203**, 210, 215
- synchronisation, 91, **92**, 117
- SysML, **157**, 298
  - activity diagram, 23, **163**
  - block definition diagram, **159**, 298
  - constraint block, **161**
  - flow port, **160**
  - internal block diagram, **160**, 298
  - parametric diagram, **161**
  - requirements diagram, **159**
  - sequence diagram, **163**
  - state machine, **163**, 362
  - use case, 23, **157**
- system, 3, 12, **15**, 380
  - boundary, 16, 116, **131**, 380
  - embedded, 3, 4, 6–10, 15, 17, 293–296, 301–303
  - under test, **185**, 380
- system**, **93**, 102, 179, 319
  
- test, **14**, 116, 119, 133, 134, 154, 156, 157, 165, 167, 168, 170–172, 174, 175, 182, 185, 189, 206–208, 226, 237, 253, 256, 258, 265–271, 299–301
  - acceptance, **155**, 182
  - coverage, **114**, 368
  - regression, **131**
  - result, **21**, 21, 380
- thread, **91**, 117, **126**, 319
  - periodic, **91**, 126, **127**, 177, 247, 358
- time**, **83**, 94, 108, 134, 177, 250
  
- t1**, **78**, 315
- TIIX Tractor, 211
- transition
  - CT, **282**
  - DE, **281**
  - relation, **279**, 279, 281
  - rule, **278**
- truetime, **296**
- type, 63
  - boolean, **66**, 66, 67, 70, 315
  - invariant, **67**, 81, 355, 360, 361, 365
  - mapping, **81**, 177
  - optional, **91**, 136
  - quote, **74**, 74, 361, 362
  - record, **75**, 116, 316
  - sequence, 73, **77**
  - set, **76**, 79, 84
  - union, **74**, 365
  
- union**, **77**, 313
  
- values**, **64**, 113, 176, 360
- variable
  - controlled, 18, 20, 48, 63, **65**, 98, 163, 164, 274, 305, 378, 380
  - monitored, 18, 20, 48, 63, **64**, 98, 101, 102, 163, 164, 171, 379, 380
- variables**, **307**
- VDM, 17, **61**, 313
- VDM link file, **101**
- verification, 264–266, **301**
- viscous friction, **29**, 30
  
- when**, **108**