PHD DISSERTATION

Methodological Guidelines for Modelling and Design of Embedded Systems

by Sune Wolff

AARHUS UNIVERSITY

Methodological Guidelines for Modelling and Design of Embedded Systems

Methodological Guidelines for Modelling and Design of Embedded Systems

PhD Thesis by

Sune Wolff

Aarhus University Department of Engineering, Denmark

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

River Publishers
Aalborg

ISBN 978-87-92982-03-3 (e-book)

Published, sold and distributed by:
River Publishers
P.O. Box 1657
Algade 42
9000 Aalborg
Denmark

Tel.: +45369953197
www.riverpublishers.com

Abstract

The development of heterogeneous embedded systems is a demanding discipline. Technical challenges arise from the need to develop complex, feature-rich products that take the constraints of the physical world into account. This thesis shows how a modelling approach to embedded systems development can address some of these challenges. Methodological guidelines supporting various levels of modelling fidelity are presented. These range from: mono-disciplinary modelling, where the embedded controller as well as its environment are modelled in a single formalism, to multi-disciplinary modelling where separate formalisms are used to describe the controller and environment. The use of the guidelines is demonstrated by means of several industrial case studies from the electronic warfare domain. To support the project management aspect of using a modelling approach to embedded systems development, the integration of formal modelling and agile methods is described. The result is a collection of lightweight methodological guidelines which can be integrated into industry strength development processes.

Resumé

Udviklingen af heterogene indlejrede systemer er en krævende disciplin. Udviklingen af komplekse, funktionalitets-rige produkter der skal tage højde for fysiske begrænsninger skaber mange tekniske udfordringer. Denne PhD afhandling viser hvordan en modelleringstilgang til udviklingen af indlejrede systemer kan løse nogle af disse udfordringer. Metodiske retningslinjer der guider brugeren i konstruktionen af modeller med forskellige detaljegrader er beskrevet. Disse retningslinjer spænder fra: modellering hvor den indlejrede controller samt dens fysiske omgivelser er modelleret i en enkelt formalisme, til tværfaglig modellering, hvor særskilte formalismer bruges til at beskrive controlleren samt de fysiske omgivelser. Anvendelsen af retningslinjerne demonstreres ved hjælp af flere industrielle cases indenfor domænet: elektronisk krigsførelse. Til at understøtte integrationen af denne modelleringstilgang til udviklingen af indlejrede systemer, beskrives en kombination af formel modellering og agil udvikling. Resultatet er en samling af metodologiske retningslinjer, som kan integreres i udviklingsprocesser der bruges i industrien.

Acknowledgements

I would like to thank my supervisor and mentor professor Peter Gorm Larsen from the Department of Engineering, University of Aarhus. It was been a pleasure and a great source of inspiration working with Peter. His professionalism, dedication and work-ethics never seizes to amaze me.

I thank Terma A/S for taking on this Industrial PhD in these financially trying times, and for providing interesting and challenging case studies. I would also like to thank my industrial supervisor Henning Staun Nielsen for his help the last three years. Many of my "termite" colleagues deserve thanks: Troels Lund Rasmussen, Mikael Kallesøe, Erling Østergård, Kristian Hennings, my former section leader Jørgen Jakobsen as well as my present sections leaders Preben Schmidt Nielsen and Martin Løkke Nielsen. Andrew Thomas Gittins deserves special thanks for his great interest in my work and for seeing business opportunities in the results.

I thank "The Danish Agency for Science, Technology and Innovation" for providing parts of the funding for the Industrial PhD.

I would also like to express my gratitude to all of my co-authors: Peter Gorm Larsen, Tammy Noergaard, John Fitzgerald, Ken Pierce, Marcel Verhoef, and Patricia Derler. I have learned a lot from working with this talented bunch of people, and they have helped improve my dissemination skills as well as my general skills as a researcher.

I owe many thanks to Peter Gorm Larsen, Joey W. Coleman Stefan Hallerstede and Marcel Verhoef for providing valuable feedback during the preparation of this PhD thesis. They have provided many detailed corrections, suggestions and questions which have greatly improved the final version of the thesis.

I would also like to thank the assessment committee: Professor Mark Lawford, Department of Computing and Software, McMaster University; Professor Anders P. Ravn, Department of Computer Science, Aalborg University; and Professor Henrik Myhre Jensen, Department of Engineering, Aarhus University.

I would like to thank the entire consortium behind the DESTECS project for many stimulating discussions, and for providing feedback on my work. Special thanks go out to Controllab Products for providing frequent 20-sim support.

For the many cups of coffee and fun lunch break with interesting talks, I thank my fellow PhD students and colleagues: Kenneth Lausdahl, Augusto Ribeiro, Joey W. Coleman, Claus Ballegård Nielsen, Anders Kaels Malmos, Rasmus W. Lauritsen, Luis D. Couto and Martin Peter Christiansen. I want to thank Mette Stig Hansen and Annie Thomsen from the administration of the Department of Engineering for guiding me through the bureaucratic jungle a life at the University can be.

I thank my parents, Hanne and Hans-Werner Wolff for raising me, and for giving me a solid set of values to live by. I also thank my brother Ebbe Wolff — the constant encouragement and support from my family has been priceless.

Last, but certainly not least, I would like to express my deepest and most sincere gratitude to my beautiful wife, Maria Schnack Wolff for her love, patience and support. I thank her for reminding me that there is more to life than work, and for giving me two strong sons: Adam and Carl — my pride and joy in life. This thesis is dedicated to the three of them.

Contents

Part I

Context

1

Introduction

This chapter introduces the context for this thesis and defines its problem domain, scope and objectives. The structure of the thesis is also described, guiding the reader through the flow of the various chapters. The extent to which the objectives of the thesis are satisfied is evaluated in Chapter 11.

1.1 Embedded Systems Introduction

Computers play an increasingly important role in the everyday life of most people. We use them for work, to stay in touch with friends and family using various social media, make online purchases, play games, check the weather forecast, and the list goes on. Over the years computers have shrunk in size ensuring that we have a vast amount of computing power in the palm of our hand — current state-of-the-art smartphones and tablet computers have the same computational power as professional workstations had a decade ago.

These general-purpose computers are only the tip of the iceberg though. We are literally surrounded by small computers solving various tasks for us. Consider a typical kitchen: the microwave, a digital egg timer, a blender, the controller managing the automatic defroster in a refrigerator, and so on. Similar systems are used on a much larger scale in our surroundings: the engine management processor in a modern car; traffic lights along the roads; satellites; aircraft guidance systems; air traffic control systems; nuclear power plants and so on. These systems are found all around us, and only few realize their contribution in making our life comfortable and safe.

These single-purpose computing systems are called *embedded systems*. They are computing systems characterised by being integral to and wholly encapsulated by the systems they control. Consider the following example from the avionics and automobile industries:

- In 2010 it was estimated that 30% of an airplane cost was from embedded systems.

- The use of embedded systems in an airplane range from engine controls over entertainment devices to navigation equipment.
- Aviation depends on embedded systems to process tickets, check-in luggage and manage logistics, and many other more critical functions.
- A multitude of advanced car applications exist: in-car cameras to judge the presence, weight, and position of occupants for best airbag deployment; rear-view mirrors with image-recognition capabilities that sense impending rear-end collisions; and cruise control systems.
- In 2010, 35% of the expenses of a new car was from embedded systems.
- State-of-the-art automobiles use concealed radar and laser sensors to analyze road conditions.
- Anti-lock Braking Systems (ABS) are one of the many safety systems in a car that uses embedded systems.

As can be seen, embedded systems can fit into various categories ranging from non-critical systems that still need to work reliably (on-board entertainment or car radio) to critical systems ensuring the safety of people.

Safety is one aspect were we depend on these computerized embedded systems. In the list above, airbag deployment and ABS systems of modern cars are good examples of this. Failures of such safety aspects can cause effects ranging from undesired to catastrophic.

Most users have accepted the need of rebooting their Personal Computer (PC) or manually having to update drivers — this is not the case for embedded systems. Consider a cash machine: if users were forced to reboot such a system halfway through a money transaction, the user would quickly loose faith in the system and stop using it. Hence, embedded systems need to consider reliability above almost everything else.

A special type of embedded systems are called *Cyber-Physical Systems* (CPSs). A CPS is a networked embedded system that integrate multiple software-rich controllers with sensors and actuators in multiple physical plants. The modelling of CPSs provides interesting challenges (see future work in Section 11.4) but is outside the scope of this thesis.

1.2 Embedded Systems Design Challenges

Viewed from both technical and commercial perspectives, the development of embedded systems –consisting of software, electronics and mechanical components that operate in a physical world– is a demanding discipline [89, 90]. Technical challenges arise from the need to develop complex, feature-rich products that take the constraints of the physical world into account. The task is made even more challenging due to the fact that these types of systems often are developed out of phase: initially the mechanical parts are designed, then the electronics and finally software is designed. Problems discovered late in the development process can only be corrected in the software without causing huge delays to the complete project. This is due to the longer iterative cycles in electronics and mechanical development. These late changes often increase the complexity of the software and the risk of introducing new errors. Hence, an otherwise well-designed software solution could be compromised. In order to avoid situations like this, early analysis and feedback at a systems level is necessary.

1.2.1 Challenges Identified by Industry

At the core of the challenges real-world development teams building embedded systems face are the trade-offs between quality, schedule, features and price. To ship a high quality product on time requires that all team members have at least a systems-level understanding of the system.

For instance, hardware engineers need to have a basic understanding of the software layers and requirements. This includes device drivers, operating systems, middleware, and the application software components. This is because target system hardware requirements depend on the underlying requirements of the software, and must ensure that the required input and output are available, the target hardware has sufficient memory of the right type, and the processor is powerful enough.

On the flip-side, programmers with a pure software background that are accustomed to developing on PCs or larger computer systems are often intimidated by the embedded hardware and tools. As a result of this, many teams end up using PCs themselves as both target and host in the place of available target hardware to do development and testing. In this case programmers do not recognize the importance of using some type of reference hardware and/or simulation platform that reflects the target.

The Aberdeen Group have made a survey of the main challenges faced when developing embedded systems [38]. Approximately 160 companies participated in the survey, and the main results can be seen in Table 1.1.

This thesis is based on an Industrial PhD which was conducted in collaboration between Aarhus University and an industrial partner Terma A/S. Terma is amongst the world leading developers of self-protection systems for military airforce and navy. Terma have defined the case studies presented in Chap-

#	Challenges	Response
1	Difficulty finding and hiring experienced systems engineers and/or lack of cross-functional knowledge	50%
2	Early identification of system level problems	45%
3	Ensuring all design requirements are met in the final system	40%
4	Difficulty predicting / modelling system product behaviour until physical prototypes exist	32%
5	Difficulty implementing an integrated product development solution for all disciplines involved in embedded systems development	28%
6	Inability to understand the impact design changes have across disciplines	18%

Table 1.1 Top six challenges of embedded software development.

ters 6 and 10. The industrial challenges in developing complex embedded systems will also be seen from the point of view of Terma.

1.3 Modelling of Embedded Systems

Modelling is a technique that is often used during the development of embedded systems to address some of the challenges identified in the Aberdeen survey. A more or less abstract representation of the system is created called the *model* and is analysed through several test runs called *simulations*. It is important that the model includes an appropriate level of detail of the system properties. The following definitions are used throughout the thesis:

Abstraction: The removal of detail that is not necessary for that level of decision making.

Accuracy: Removal of detail is a reduction of accuracy in that particular domain.

Fidelity: A measure of accuracy of the model when compared to the real world realisation.

Competent model: We regard a model as being *competent* for a given analysis if it is of a sufficient level of fidelity to permit that analysis.

Normally, the engineers involved in the development of an embedded system make use of domain-specific tools in order to create models of the individual parts of the system. This enables the individual engineering disciplines to optimize specific parts of the system. By creating abstract high-level executable models of the entire embedded system, and then simulating them, systems engineers are able to reason about system-level properties at design time. This ensures early feedback on system properties, as well as easing communication across different disciplines, since the impact of a given design decision is made visible to the entire team. The following terms define the difference of these modelling approaches:

Mono-disciplinary modelling: The entire system is modelled using a single formalism. A high level of abstraction is applied to other discipline-specific parts of the system, or those parts are abstracted away all together. In this thesis, the mono-disciplinary modelling is done using the *discrete-event* (DE) formalism VDM.

Multi-disciplinary modelling: Different formalisms are used for modelling separate discipline-specific parts of the system. In this thesis the software controller of the embedded system is modelled using a DE formalism while the physical dynamics are modelled using a *continuous-time* (CT) formalism. In this thesis, the multi-disciplinary modelling is mainly done in the DESTECS tool, where the two domain-specific models are analysed using co-simulation linking the simulation engines of the two domain-specific modelling tools.

Modelling is applied at various maturity levels focusing on a single discipline or spanning several engineering disciplines; Figure 1.1 gives an overview of four levels of maturity. Just like similar capability maturity models [93], it should be seen as a progressive standard where an organisation must pass their current level in order to proceed to the next. This ensures that the organisation undergoes a slow, steady progress through the levels of modelling maturity. Rios et al. [169] have defined a similar maturity model for model-driven development that focusses more on code generation.

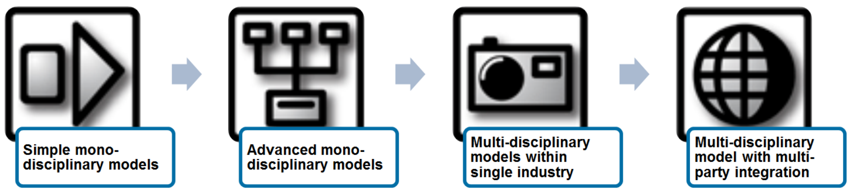

Simple mono-disciplinary models
Advanced mono-disciplinary models
Multi-disciplinary models within single industry
Multi-disciplinary model with multi-party integration

Figure 1.1 Modelling maturity model.

Each of the four levels of modelling maturity supports several levels of fidelity. The simple mono-disciplinary models could focus solely on functionality of the system, while the more advanced mono-disciplinary models add an architectural description. The multi-disciplinary models add fidelity to the model by describing the physical dynamics of the system, while the multi-party integration adds complex considerations to the modelling process. This last level of modelling maturity is outside the scope of this thesis, however.

1.3.1 Research Projects

In the research world, modelling of embedded systems has been considered for more than a decade. Many research projects have attempted to tackle the challenges for developing such multi-disciplinary systems in a predictable fashion. Below the research initiatives that are most relevant for this thesis are listed. A more complete overview of related projects can be found in Appendix C.

DESTECS [43] developed methods and tools that combine CT plant models with DE controller models through co-simulation to allow multi-disciplinary modelling, including modelling of faults and fault tolerance mechanisms.

Modelica [79] is a non-proprietary, object-oriented, equation-based language to conveniently model complex physical systems.

MODELISAR [76] has developed an open co-simulation interface for coupling models called the *Functional Mock-up Interface* (FMI).

Ptolemy [44] has studied modelling, simulation and design of concurrent, real-time embedded system, using a heterogeneous mixture of models of computation using an actor-oriented modelling approach.

Some of these projects have an academic approach to system modelling, while others have more focus on industrial applicability. DESTECS and MODELISAR are good examples of projects that are inspired by the industry needs described in Section 1.2.1.

1.4 Scope of the Thesis

The area of research presented in this thesis is modelling and validation of embedded systems. However, creating models specifying concurrency, distribution, real-time, and that combine several formalisms is itself a complex endeavor. The focus of this thesis is providing guidelines that support the model designer in this challenging task. The thesis remains in the scope of modelling in the early phases of embedded system development where the concept of operations for the embedded system must be defined and different design alternatives must be analysed. A model-based implementation of the final system is outside the scope of the thesis. The thesis is placed in the context of the V-model as depicted in Figure 1.2 below — the main focus of the thesis is on modelling in the "Concept of operations" phase, and to a lesser extent in the "Requirement", "Architecture" and "Design" phases.

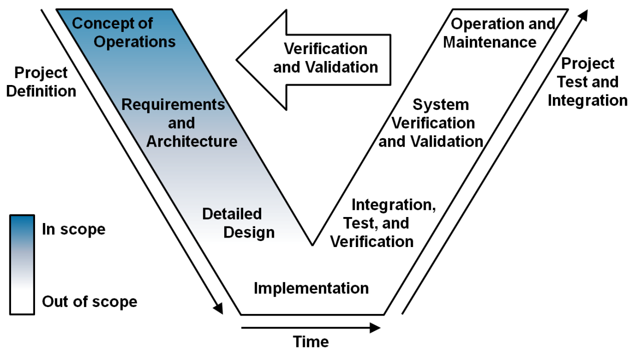

Concept of Operations
Verification and Validation
Operation and Maintenance
Project Definition
Project Test and Integration
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
In scope
Out of scope
Implementation
Time

Figure 1.2 Project scope identified in the context of the standard V-Model defined by INCOSE.

The work presented in the thesis fits into existing international industrial de facto standards like IEEE/IEC 12207 [95] and Rational Unified Process (RUP) [53]. The main focus is on the *operational concept* of IEEE 12207 and the *vision* of RUP, where a potentially huge design space must be explored in order to define the most feasible concept for the embedded system. In the *Requirements definition* and *analysis* phases models can be created to explicitly state the user requirements elicited to help identifying ambiguous and/or incomplete requirements. Such models can fit into the different artifacts produced in these phases in a natural way [184]. The models can also help in identifying critical corner-cases in which the final implementation of the system needs to function correctly. Some of the effort put into validation and verification of the models can potentially be saved in the testing of the finalised system since model test cases can be reused.

The IEC 61508 Functional Safety standard [94] defines an overall safety lifecycle that forces safety to be addressed independently of functional issues. The methodological guidelines presented in this thesis mainly support activities in phase three *"hazard and risk analysis"* of the safety lifecycle, and to a lesser extent in phase one *"concept"*, two *"Scope definition"* and four *"safety requirements"*. Since IEC 61508 addresses safety for both the *equipment under control* (the physical plant) and the *control system*, a multi-disciplinary modelling approach should be used. An alternative safety standard is the ANSI/ISA-S84.01[16]. This standard only addresses safety instrumented systems, and not the equipment under control. Hence, a mono-disciplinary modelling approach is a better fit.

1.5 Objectives of the Thesis

The objective of this thesis is to define and evaluate methodological guidelines supporting modelling and design of embedded systems. To support different levels of modelling maturity, separate guidelines must be developed for mono-disciplinary modelling (where the entire system is modelled using a single formalism) and multi-disciplinary modelling (where different formalisms are used for modelling the software controller and the physical properties of the system).

Creating models of complex embedded systems to be used for exploration of different design alternatives is inherently difficult, and it is hard to estimate the duration of the individual activities involved. Describing a process including activities with such a degree of uncertainty would not make much sense — processes are more useful for describing well known tasks of a more repetitive nature. This is the reason why this thesis focuses on methods and guidelines which should be seen more like a collection of tools — when encountering a modelling challenge methodological guidelines are in place to support the user in overcoming this challenge.

1.6 Evaluation Criteria

The developed guidelines will be evaluated with regard to the properties described below. The evaluation criteria are defined based on the challenges of embedded software development identified by the Aberdeen Group in Section 1.2.1. Challenge #3 is not addressed since the final system implementation is outside the scope of this thesis.

Prediction of system behaviour: The guidelines must help the model designer in predicting real system behaviour based on model simulation. This is important when exploring different system design alternatives using a modelling approach. This criteria is based on challenge #4 in the Aberdeen Group survey.

Cross-disciplinary collaboration: The guidelines must strengthen cross-disciplinary collaboration by ensuring common means of communication, and by allowing the individual disciplines to work with the domain-specific tools they are accustomed to. Collaboration across both technical and non-technical boundaries must be enhanced. This criteria is based on challenges #1 and #5 in the Aberdeen Group survey.

Understanding system-level impact of design decisions: Design decisions must be made visible at the system-level. This will help in the analysis of system-level impact of local design decisions, and will help the individual disciplines to understand the effect of domain-specific design decisions. This will help in optimising the system globally opposed to local optimisation of only parts of the system. This criteria is based on challenges #2 and #6 in the Aberdeen Group survey.

Industry-ready: It is important that the methodological guidelines fit into existing development methods used by the industrial partner Terma A/S. The guidelines shall augment existing development practices and standards, and be applied with the least disruption possible. This criteria is not based on any *one* of the challenges identified by the Aberdeen Group survey, but is of major importance to ensure that the industrial partner will benefit from the guidelines.

1.7 Thesis Structure

The thesis structure accommodates the thesis objectives of providing methodological guidelines that support various levels of modelling fidelity. Part I introduces the thesis and provides information on the application domain. If the reader is primarily interested in mono-disciplinary modelling guidelines he should proceed to Part II and from there go straight to the evaluation, discussion and conclusion in Part IV. Readers interested in multi-disciplinary modelling should go to Part III and from there to Part IV. Of course the thesis can also be read all the way through which will provide information on both mono- and multi-disciplinary modelling. Each chapter provides concluding remarks on that specific chapter, whereas Chapter 11 concludes the thesis in its entirety. This reading guide is shown graphically in Figure 1.3.

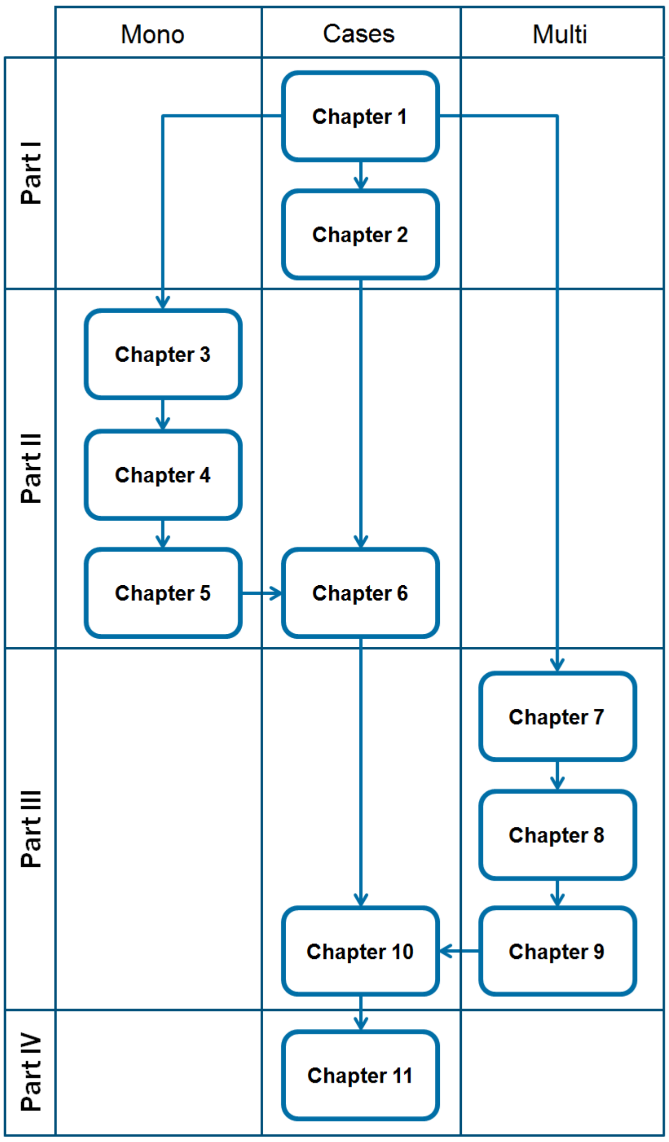


Figure 1.3 Structure and reading guide for the thesis.

Part I of this thesis introduces the problem domain and defines the objectives of the thesis (this chapter). The electronic warfare domain is introduced in Chapter 2, giving the reader the necessary background knowledge needed in order to understand the case studies presented throughout the thesis. Part I is based on the following publication[1]:

[199] Sune Wolff, Peter Gorm Larsen, and Tammy Noergaard. *Development Process for Multi-Disciplinary Embedded Control Systems*. In proceedings of the 7th EUROSIM Congress on Modelling and Simulation. September 2010.

Part II of this thesis describes the use of mono-disciplinary modelling (using the DE formalism VDM) in the development of embedded systems. A structured, stepwise approach to developing distributed real-time models using the formal language VDM-RT is described in Chapter 4. In Chapter 5 a way of combining formal and agile methods is discussed, and a concrete example is given of how formal specifications can be used in the agile project management framework Scrum [174]. The use of formal methods in an industrial setting is discussed in Chapter 6. The discussion is based on a case study within the electronic warfare domain. A combination of the methodological guidelines presented in Chapters 4 and 5 is applied in the case study. The guidelines are deliberately kept separate from the case study exemplifying their use. This was done to create a better reference document, easing future use of the guidelines without having them cluttered with the case study description. In addition, this creates a clearer separation of the guideline presentation to the evaluation. Part II is based on the following publications:

[119] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. *Methods for the Development of Distributed Real-Time Embedded Systems using VDM*. International Journal of Software and Informatics, 3(2-3), October 2009.

[120] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. *Are formal methods ready for agility? a reality check*. In Stefan Gruner and Bernhard Rumpe, editors, 2nd International Workshop on Formal Methods and Agile Methods, pages 13–25. Lecture Notes in Informatics, September 2010.

[197] Sune Wolff. *Scrum Goes Formal: Agile Methods for Safety-Critical Systems*. In ICSE 2012: Proceedings of the 34th International Conference on Software Engineering, pages 23–29, June 2012. Workshop on For-

[1] Bibliographic reference added for reading convenience

mal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA 2012.

[198] Sune Wolff. *Using Formal Methods for Self-defense System for Fighter Aircraft — An Industrial Experience Report*. International Journal of Empirical Software Engineering, 2012. Submitted — under review.

Part III of this thesis describes the use of multi-disciplinary modelling (combining DE and CT models) in the development of embedded systems. An overview of available tools capable of collaborative modelling and co-simulation is given in Chapter 7. The Ptolemy and DESTECS tools are subjects of a comparative study in Chapter 8. Both tools are used for modelling an aircraft fuel system and several comparison criteria are used to benchmark the two tools against one another. In Chapter 9 methodological guidelines for developing collaborative models are presented and evaluated in Chapter 10. As part of this evaluation the central case study from Chapter 6 is extended with a CT description of the environment in which the system operates, following the methodological guidelines. As in Part II, the guidelines are kept separate from the case study applying them, to create a methodological reference document easing future use. Part III is based on the following publications:

[74] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff. *Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems*. In D. Méry and S. Merz (editors), IFM 2010, Integrated Formal Methods, volume 6396 of Lecture Notes in Computer Science, pages 12–26. Springer-Verlag, October 2010.

[200] Sune Wolff, Ken Pierce and Patricia Derler. *Multi-domain Modelling in DESTECS and Ptolemy — a Tool Comparison*. In SIMULATION: Transactions of The Society for Modeling and Simulation International (journal). Submitted — under review.

[196] Sune Wolff. *Methodology for collaborative modelling*. In International Journal of Software and Systems Modeling. Submitted — under review.

In the final part of this thesis (Part IV), the research contribution of the PhD thesis is summarised. The extent to which the objectives of the thesis have been met is discussed, and future work is identified.

Appendix A provides a glossary of commonly used abbreviations and terms used in this thesis. In Appendix B, a complete list of all the operators of the formal method VDM are listed along with some examples of the use of the formal specification language. Related projects are briefly introduced in Appendix C.

2

Application Domain: Electronic Warfare

Throughout this thesis several case studies are presented, all associated with the electronic warfare domain. They showcase the application of the guidelines and how they support the modelling of embedded systems. This chapter provides necessary domain knowledge for the case studies described in Chapter 6 and 10 and to a lesser extent to Chapter 8.

2.1 Introduction

Military forces depend on the electromagnetic spectrum for many applications including: detection; identification; and targeting. Combatants try to dominate the electromagnetic spectrum in order to find, fix, track, target, engage, and assess the adversary, while denying that adversary the same ability. Electronic Warfare (EW) is traditionally divided into the following three categories:

Electronic Support Measures (ESM) : Used to gain knowledge of the enemy using sensors operating in the electromagnetic spectrum. Examples of this are infrared (IR) sensors mounted on missiles to track the position and movement of an enemy, or IR sensors mounted on airborne systems to acquire information of incoming missiles.

Electronic Countermeasures (ECM) : Used to suppress the enemy use of ESM. One way to do this is to deploy decoys emitting energy in the correct frequency band of the electromagnetic spectrum in order to hide the airborne system from incoming threats. Another example is active ECM like jammers that blind or destroy enemy ESM.

Electronic Protective Measures (EPM) : Used to lower the enemy use of ECM. As an example, resistance to jamming can be mentioned. Is sometimes called *Electronic counter-countermeasures* (ECCM).

These three categories –as well as the electromagnetic spectrum– are described in further details in the following sections of this chapter. The description below is based on the engagement scenario between a fighter aircraft and a missile. The main inspiration for this chapter was found in [173], [166] and [21] — for a more detailed description of the electronic warfare domain, please consult these texts.

Section 2.2 gives a short introduction to the electromagnetic spectrum. In Sections 2.3 an overview of different categories of threats is given, while Sections 2.4 introduces various generations of missile guidance technologies. Sensor technologies used to monitor the electromagnetic spectrum are introduced in Section 2.5, and Section 2.6 describes various technologies used to reduce the effectiveness of these sensors. Section 2.7 describes techniques used to protect the sensors, and finally Section 2.8 gives a brief introduction to movement and rotation in 3D space.

2.2 The Electromagnetic Spectrum

For a large part of history, light was the only known part of the electromagnetic spectrum. Renowned scientists like Michael Faraday and James Maxwell started the expansion of this definition to encompass the range of all possible frequencies of electromagnetic radiation. An overview of the electromagnetic spectrum is given in Figure 2.1.

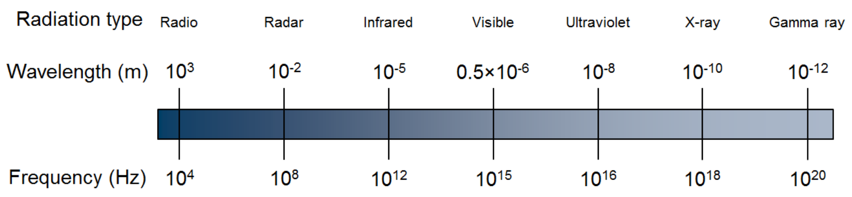


Figure 2.1 A diagram of the electromagnetic spectrum, showing radiation type across the range of frequencies and wavelengths.

EW is mainly concerned with the radar and IR bands of the electromagnetic spectrum. Radar is short for *RAdio Detection And Ranging*. A radar system continuously transmits waves or short bursts of electromagnetic energy in the radio frequency (RF) band, that bounces off objects such as ships or aircraft. Part of the transmitted signal is reflected back to the radar system making it possible to determine the direction of and range to the objects. A special type of radar system called a Doppler radar can determine the velocity of an object using the difference in the frequencies between the emitted radar signal and the returned signal, caused by the movement of the object relative to the radar source.

Asymmetrical objects (like aircrafts) return different amounts of energy when hit from various angles. This phenomena is call the *radar-cross-section* (RCS) of a given object and is used by radar systems to determine the type of object detected. An example of the RCS of an aircraft is given in Figure 2.2.

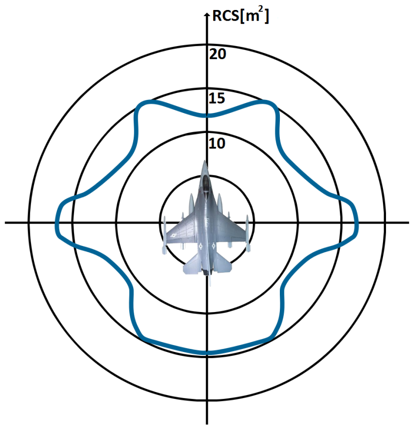


Figure 2.2 Example of radar-cross-section of an aircraft.

Radar systems operate in different modes: initially the radar is *scanning* for possible enemy aircrafts; once an aircraft is intercepted, the radar can *track* it in order to monitor the trajectory of the aircraft; if a missile is fired, the radar is *locked* onto the aircraft in order to help guiding the missile towards its target. The energy transmitted from the radar differs in these modes, which makes it possible for sensors to determine the current mode of the radar system.

Radiation of energy from an aircraft within the infrared band of the electromagnetic spectrum is mainly due to the heat generated by the engine. In addition, air friction generates heat on the leading edges of the aircraft that leads to energy emission in the IR band. Examples of the IR signatures of a fighter aircraft, helicopter and commercial aircraft can be seen in Figure 2.3. This energy is monitored by IR sensors and is used by IR guided missiles (*heat-seeking missiles* in layman's terms) to guide the missile towards its target.

2.3 Threats

An aircraft can encounter different types of threats when flying missions in enemy territory — either from a ground-based missile launcher (*Surface-to-Air Missile* (SAM)) or from an enemy aircraft (*Air-to-Air Missile* (AAM)). A special type of SAM is the *Man Portable Air Defense System* (MANPADS) — a small portable missile system that can be operated by a single person.

Older types of missiles needed to actually impact with a target in order to detonate. Most modern missiles however, have proximity sensors that detonate the missile once it is within a certain range of the aircraft — this is called the *deadly envelope* of the missile.

2.4 Missile Guidance

Modern missiles use a guidance system to steer towards the target. The guidance system uses electromagnetic radiation in either the RF or IR bands of the electromagnetic spectrum to determine the position of the target.

RF guided missiles are either equipped with a radar system itself or make use of one or more ground-based radar systems to locate the target. This is an *active* type of guidance where the missile transmits energy in the RF band to determine the position of the target.

IR guided missiles uses a sensor to monitor the IR radiation from the surroundings that is used to determine where the target is. Since no energy is emitted from the ESM of an IR guided missiles this type of guidance is called *passive*.

IR guided missiles have evolved a lot through the years. A quick overview of four generations of IR guidance systems is given below (for a more in depth explanation, see [21]):

First generation seekers: Using an un-cooled detector technology, operating in the near-IR waveband, see high temperature targets like the engine and exhaust plume of an aircraft. This limits the use of this guidance technology to rear-engagement. A rotating disk in front of the IR sensor is used to determine the size of the target as well as determine if the missile is flying towards the middle of the target. This generation of

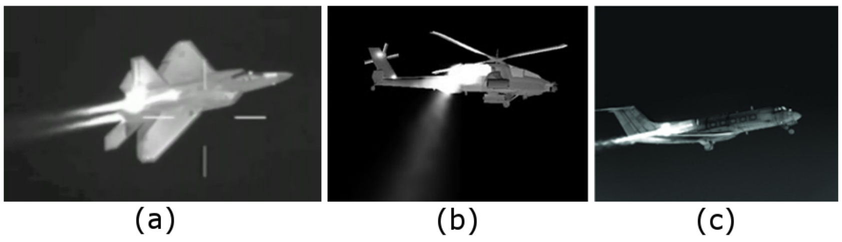


Figure 2.3 Infrared signatures of (a) F22 fighter aircraft, (b) helicopter and (c) commercial aircraft.

IR guidance suffers from insensitivity when the target is directly in the center of the field-of-view of the sensor.

Second generation seekers: By rotating the IR sensor, the insensitivity of the first generation IR guidance is removed. Instead of trying to maintain the target directly in the middle of the field-of-view of the sensor (as the first generation does) the target must rotate around the center field-of-view in a perfect circle if the missile is on target. Late second generation missiles started using dual band IR sensors enabling them to distinguish the hot engine of an aircraft from the warm leading edges resulting in a much more nuanced target picture.

Third generation seekers: Creates a simulated image of the target by rotating prisms in front of the IR sensor in certain patterns in order to move the target over the IR detector (this technology is often called *Pseudo Imaging Systems*). This generation of guidance uses a much narrower field-of-view (while maintaining the same accuracy as earlier generations) which increases the robustness against countermeasures.

Fourth generation seekers: Multi-element detectors are used to create a true 2-dimensional IR image of the target. This technology is robust against countermeasures since the missile has an actual image of the target, and is less likely to be lead away from this. The more pixels in the IR sensor, the more accuracy — but also the more computational power is needed to process the imagery.

Once the missile target has been determined (using one of the technologies described above) various steering techniques are used to control the missile trajectory. *Line-of-sight* trajectory continuously maintain the target in the center of missile field-of-view. This is often called *pure pursuit* since the missile continuously steers directly towards the target.

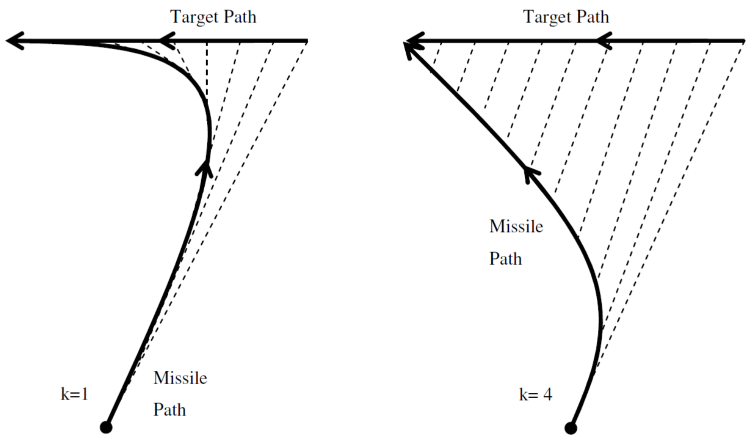


Figure 2.4 Pure pursuit (left) and proportional navigation (right) explained. Source [21].

Proportional navigation uses a proportional gain on the current direction error in order to aim at where the target is moving towards instead of the targets current position. These two approaches are illustrated in Figure 2.4.

2.5 Electronic Support Measures

ESM is the general term used when describing equipment used to monitor and sense energy being emitted in the electromagnetic spectrum. ESM covers the RF and IR guidance sensors used by missiles, but this section focuses on the ESM installed on-board aircraft used by pilots to gain knowledge of the current threat situation.

2.5.1 Radar Warning Receiver

Radar tracking is an active type of missile guidance, where the aircraft is hit by the electromagnetic energy transmitted by the radar system. This enables sensor systems (called *Radar Warning Receiver* (RWR)) installed on-board the aircraft to notify the pilot of incoming radar guided threats. As described in Section 2.2 radar systems operate in different modes if they are scanning for a target, if a target is being tracked and if the systems locks onto the target in order to guide a missile towards it. By detecting the characteristics of the radar signal, the RWR can determine the current mode and warn the pilot.

2.5.2 Missile Warner System

IR guidance is a passive guidance system, which in itself does not emit electromagnetic energy that can be monitored by ESM. A *Missile Warner System* (MWS) is a passive system operating in either the IR or RF domain. A passive IR-based MWS continuously monitors the IR images of the surroundings of the aircraft. If a hot spot is detected (the exhaust plume of a missile), which over a short time span is moving towards the aircraft, the MWS issues a missile warning to the pilot.

Active MWS also exist, operating in the RF frequency band. Radar waves are continuously being transmitted in order to determine the range and velocity of incoming missiles. RF-based MWS systems can have issues detecting missiles with a small RCS (see Figure 2.2). Another major drawback by having an active MWS is that missiles can use the transmitted energy to help track the aircraft.

2.6 Electronic Countermeasures

In an engagement scenario between an aircraft and a missile, both systems make use of ESM: the missile uses ESM to determine the position of the aircraft, and the aircraft uses ESM to detect missiles to determine incoming direction, range, missile type and so on. ECM are used to counter the enemy's use of ESM — in this section we are mainly concerned with the ECM used by the aircraft in order to reduce the effectiveness of the missile ESM.

2.6.1 Jammer

Radar systems utilise the reflected energy from a transmitted radar signal to determine position and velocity of the aircraft. To counter radar guided missiles an aircraft can use a radar *jammer* — the simplest of which basically emits a noise signal within the same frequency band as the one used by the enemy radar system. More sophisticated jammers can transmit a signal that tricks the radar system into calculating the wrong position, range and/or velocity of the aircraft.

For the jammer to be effective, the emitted signal needs to be with a higher power than the signal the enemy radar would otherwise receive. This ensures that the original signal is interpreted as noise, and the jammer signal is used instead. There is a drawback to this though: the jammer signal can attract enemy radar systems. Thus, it is important to turn off jammers when not in use, so that the jammer is not acting as a beacon for enemy radar systems.

2.6.2 Chaff

An alternative type of ECM effective against RF guided threats are dispensable decoys called *chaff*. Chaff consists of thin pieces of aluminum foil or metalised glass fibres, which creates a cloud when dispensed from the aircraft. This cloud has a RCS similar to that of the aircraft appearing as an alternative target. By combining the dispensing of chaff with aircraft evasive maneuvers, the enemy radar system will start tracking the chaff cloud instead of the aircraft. This scenario is illustrated in Figure 2.5.

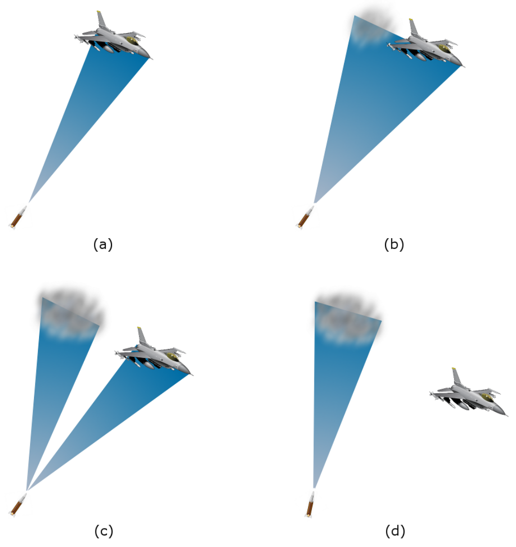


Figure 2.5 Chaff dispensing scenario. In (a) the missile is tracking the aircraft. A chaff cloud is dispensed in (b) at which point the missile see one long target. The chaff cloud separates from the aircraft in (c) at which point the missile has two targets to choose from. In (d) the missile hos chosen the chaff cloud as the new target.

RF guided missiles use different techniques to determine the main target amongst multiple possible targets. One such technique is a *range gate* where the missile only evaluates targets within the given gate — if the aircraft can escape this gate before the chaff cloud disappears, the missile looses its track on the aircraft.

2.6.3 Flare

IR guided missiles rely only on the electromagnetic radiation in the infrared waveband. Dispensable decoys called *flares* can be used by the pilot to escape incoming missiles. Flares are made from flammable material that creates an IR signature when dispensed. The engagement scenario is similar to the one described in Figure 2.5.

As described in Section 2.4 early generation IR guided missiles only monitor a single frequency band in the IR domain to detect the engine and exhaust plume of the aircraft. To counter this type of IR guidance, flares which emit energy in a single frequency band are used — the flares need to emit more energy in that frequency band in order to be more attractive to the missile.

Later generation IR guidance systems are able to distinguish the hot engine from the warm leading edges of the aircraft by monitoring the IR domain in two separate frequency bands. To counter these more modern types of IR-based guidance systems, pilots can deploy dual-frequency flares that emit energy in two separate frequency bands to better resemble the IR signature of the aircraft.

Once the flares are dispensed from the aircraft the velocity quickly degrades due to drag generated by the air resistance. More modern IR guidance systems are *forward-biased* — they prefer a target that keeps moving forward and does not slow down all of a sudden. In order to counter these more intelligent systems, *self-propelled* or *kinematic* flares can be used. This type of flares follows the movement of the aircraft for a longer time, and only slowly increase the distance to the aircraft making it much harder for the missile to distinguish the flare from the aircraft.

2.7 Electronic Protective Measures

Whereas ESM are used to monitor the electromagnetic spectrum and ECM are used to reduce the enemy's electromagnetic knowledge, EPM are used to protect one's own ESM equipment from enemy ECM interference. One example of EPM already mentioned in Section 2.6.3 is the forward-biasing of modern missile guidance systems. This is a protective mechanism used in order to avoid interference from flares dispensed by the target. EPM is also often called *counter-countermeasures* since they try to counter the enemy use of ECM.

Radar systems can also make use of EPM in order to reduce the effect of jammers. One such technique is *frequency agility* where the radar system swiftly shifts the frequency used. This makes it much harder to RWR systems to determine the correct frequency to be used by the jammer.

2.8 Movement and Rotation in 3D Space

Most of the background information described in this chapter applies to not only fighter aircrafts, but also to helicopter, transport aircrafts and even battleships — even though ships are outside the scope of this thesis. A full introduction to flight dynamics is also outside the scope of this thesis, but a short introduction to some of the terms used when describing the position and rotation of an aircraft in a six degrees of freedom (6DoF) model is needed though.

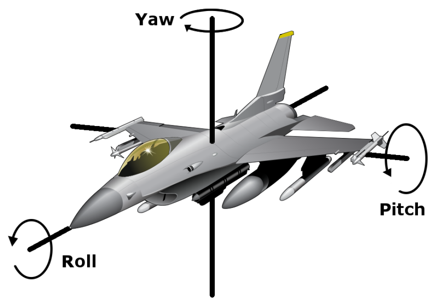


Figure 2.6 The three rotational axis of an aircraft: roll, pitch and yaw.

The position of an aircraft can be described using Global Positioning System (GPS) coordinates, or (as is the case in this thesis) as a simple three dimensional vector: forward-backward on the x-axis; right-left on the y-axis; and up-down on the z-axis. The angles of rotation in three dimensions about the vehicle's center of mass, known as roll, pitch and yaw are shown in Figure 2.6. In this thesis, the three-dimensional rotation is described using quaternions, which are four-dimensional structures, representing an axis vector and an angle of rotation around this vector.

Part II

Mono-Disciplinary Modelling

3

Formal Methods for Embedded Systems Development

This chapter introduces a mono-disciplinary modelling approach for embedded systems development through the use of formal methods. The formal method VDM is described, as are selected industrial applications of formal methods. Related work on incremental formal methods and the combination of agile and formal methods is also presented. This chapter provides the formal foundation for Chapter 4, which describes guidelines for incremental development of VDM models, and Chapter 5, which discusses the combination of agile and formal methods.

3.1 Introduction

As has been described in Chapter 1, embedded systems are omnipresent. As functionality is added, the software embedded in them grows to incorporate more powerful algorithms. For example: a hybrid car with its advanced propulsion, safety and navigation features runs on as much as 200 million lines of embedded code. A method for managing the complexity of such systems is needed, and the use of system models throughout the design process has been proposed as a possible solution.

System models are often used to capture ideas and requirements. A system model can be an executable specification that allows collaboration across engineering disciplines and modelling domains. The *design space* of an embedded system encompasses all possible solutions to a given problem. *Design Space Exploration* (DSE) is the task of exploring these alternative designs in order to find the optimal solution. A system model can be simulated to explore design trade-offs quickly, decreasing the size of the viable design space.

Modelling provides the freedom to innovate by making it easy to try out new ideas and exposing design problems early. Moving design tasks from the lab and field to a simulated environment on desktop computers, enables exploration of the design before any real hardware prototypes exist. This has the potential to speed up the development of embedded systems, and to help discovering design issues as early as possible.

One way to create such system models is using formal methods which are a collection of mathematically-based techniques used in the development of computer systems. Formal methods consists of a formally defined language (a formal specification language) and various techniques for system description and analysis. Using a formal specification language, a system can be described precisely with regards to functionality, concurrency, completeness, correctness, etc. This means that the properties of a system can be analysed without actually having to execute the system [87]. Many formal specification languages have an executable subset that can be used to specify executable models of the system. The model developer can then exercise the system model in order to investigate runtime properties of the system [81].

This second part of this thesis describes the use of mono-disciplinary modelling as a tool for developing embedded systems. The focus is on the discrete-event (DE) domain where both the controller as well as the physical entities of the embedded system are modelled using a discrete formal specification language. Section 3.2 gives an introduction to the use of formal methods for developing embedded systems. The DE modelling language used throughout this thesis is the formal method *Vienna Development Method* (VDM) which is described in Section 3.3. A complete overview of the VDM operators can be found in Appendix B. This chapter is concluded in Section 3.4 with a literature overview of the use of incremental formal methods and combinations of agile and formal methods.

3.2 Formal Methods in Industry

A survey of 62 industrial applications of formal methods carried out by Woodcock et al. [201] gives a good overview of application experiences as well as future challenges in ensuring wider usage of formal methods in industry [1]. It appears that a lightweight approach [98] (where formal verification techniques are applied to only the most critical parts or properties of a system) dominates the industrial use of formal methods. Lightweight analysis techniques involving targeted use of formalisms in only the most critical subsystems or system properties, or automated validation techniques seem to be preferred over time-consuming formal proof. The survey shows that 85% of the projects had staff with prior experience in the use of formal methods. The

[1] In the most recent update, the survey now includes feedback from 98 projects.

value of targeted assistance from experienced practitioners is emphasised by several projects in the survey. The main challenges identified were making tools more robust and usable, integrating the use of formal methods into industrial development processes and overcoming skill barriers.

One of the industrial partners in the DEPLOY project [170] reports some impressive benefits by using automatic checking of formal specifications: *"...one person-month of effort was replaced by 17 minutes of computation."* A preference for lightweight formal methods is also expressed where the formal elements should be as non-intrusive as possible. In one case, the industrial partner SAP achieved this by building adapters to the formal notations used, integrating these into well-established development environments. This ensured that a wider range of developers could make use of the formal notation technique from within the development environments they preferred.

Two industrial applications of formal methods in the automated transport sector (Line 14 of the Paris subway [26] and the shuttle at Roissy airport [19]) have been compared by Abrial [4], and common experiences and lessons learned are derived. Both of the projects used the formal method B [3], where a software specification is transformed into the executable code through a series of formal refinement steps. Since the basis of the formal modelling is the requirements document, the importance of the quality of this document is stressed. By using formal methods to define the requirements of the software system, the risk of ambiguous or erroneous requirements is reduced. The missing integration of formal methods into existing development processes is found to be a major obstacle though — especially the proof activities are hard to incorporate.

Several reports on industrial usage of model checking at Rockwell Collins have been conducted (e.g. [138] and [139]). Several interesting lessons were learned including: *"...it is easier for the developers to master the verification tools than it is for experts in formal verification to master the complex product domain."* — this clearly stresses the importance and challenge of good communication between developer and customer in order to ensure knowledge transfer of domain specific details. A high degree of automation was used, ensuring automatic generation of verification models from the engineering models used on a daily basis. This eliminates the tedious process of manually keeping two types of models in sync and supports quicker re-verification of models.

Androick et al. [15] have reported a rare success in large-scale formal verification — a formal, machine checked, code-level proof of the full functional correctness of the seL4 microkernel is provided. 8700 lines of C code and

600 lines of assembler were analysed in order to provide assurance for safety, security, and correct functionality of the system. An approximate effort of 14 person-years was spent on the project — similar projects are expected to take 4 to 6 person-years of effort using conventional development techniques which would not produce a formally verified kernel. It is expected that a subsequent project could be done in 8 person-years because of the experience gained as well as the possibility to reuse many proofs.

Another successful application of formal verification is reported by Wassyng and Lawford who describe the development of the shutdown system for the nuclear generating station at Darlington, Ontario [193]. Through a thirteen year project, formal methods were applied to all aspects of system development: requirements, design, implementation and verification. A lot of effort was put into making the formal methods used more easily understandable by domain experts. They conclude that: *"using a notation and presentation method that industrial participants will accept, is about as important as the technology in the formal methods themselves"*.

3.3 Formal Modelling in VDM

The formal method VDM is used throughout this thesis — both for mono-disciplinary modelling (because the author had prior experience using VDM) but also for multi-disciplinary modelling (since VDM is used to describe the DE models in the DESTECS toolchain as described in Section 7.2). This section gives an introduction to VDM covering the notation, tool support and industrial usage.

VDM is a collection of formal techniques to specify and develop software. It consists of a formally defined language, as well as strategies for abstract descriptions of software systems. In the rest of the thesis, the term VDM is used to cover both the formal language, as well as the techniques accompanying it.

VDM originates from IBM's Laboratories in Vienna in the 1970s. The very first language supported by the VDM method was called Vienna Definition Language (VDL), which evolved into the meta-language Meta-IV [27, 32] and later into the specification language VDM-SL [160] which has been ISO standardised [97]. Over the years, extensions have been defined to model object-orientation (VDM++ [72]) and distributed real-time systems (VDM-RT [145, 185]). Two alternative tools exists; the commercial tool VDM-Tools [75] and the open-source initiative Overture [116]. These are introduced in more details in Section 3.3.3.

3.3.1 VDM Notations

In the following, an overview of the VDM notation is provided. This introduction to VDM is deliberately kept short. We only introduce the key operators and concepts that are needed for understanding the capabilities of the formal method, the case studies and the guidelines. For more details, Appendix B includes a complete overview of all the VDM operators.

Data in VDM models is described using simple abstract data types such as natural numbers, booleans and characters, as well as product and union types and collection types such as sets, sequences and mappings. VDM has both a mathematical and an ASCII based syntax, where the latter is used in this thesis since it is much easier to understand. The system state is described using **`state`** in VDM-SL and **`instance variables`** in VDM++ and VDM-RT, the value of which can be restricted by invariants. To modify the state of the system, operations can be defined either explicitly by imperative statements, or implicitly by pre- and post-conditions. Functions that cannot access the state are defined in a similar fashion. Even though a large subset of implicit functions and operations can actually be executed (see [80]), this thesis uses explicitly defined functions and operations that can be interpreted by the VDMJ interpreter [127] of the Overture Tool [154].

In VDM++ and VDM-RT models, classes may be active or passive. The instances of an active class have their own thread of control; instances of passive classes are always manipulated from the thread of control of objects belonging to active classes. A thread executes a sequence of statements. Each thread is created whenever the object that houses it is created but the thread needs to be started explicitly using a **`start`** statement. Each thread terminates when the statements in the body of the thread completes execution. It is also possible to specify threads that do not terminate, and this feature is valuable in modelling reactive systems. Operations may be specified as asynchronous, allowing the caller to resume its own thread after the call is initiated. A new thread is created, automatically started and scheduled to execute the body of the asynchronous operation.

Operation execution may be constrained by *permission predicates* [113], which are Boolean expressions over instance variables. If a permission predicate evaluates to **`false`** on an operation call, the call is blocked until the predicate becomes **`true`** again. Such predicates can make use of *history counters* that, for each object, count the number of requests, activations and completions per operation. A shorthand **`mutex`** permission predicate allows the user to specify mutual exclusion between the executions of specified operations. This is used to manage concurrent access to data by the objects encapsulating the shared variables.

The VDM-RT extensions to VDM support the description and analysis of real-time and distributed systems. VDM-RT includes primitives for modelling deployment over a distributed hardware architecture and support for asynchronous communication. Within a special **`system`** class, the modeller can specify computation resources (CPUs) connected in a communication topology by buses. Two predefined classes, `CPU` and `BUS` allow scheduling (first-come-first-serve or priority-based) and performance characteristics of CPUs and buses to be readily expressed.

The semantics of VDM has been extended with a notion of time so that any thread running on a computation resource or any message in transit on a communication resource can cause time to elapse. Each construct in the modelling language has a default time associated with it, that can be changed by the user if desired.

In VDM-RT, threads can be made periodic using the built-in time concept:

```
thread

-- (period[ns], jitter[ns], delay[ns], offset[ns])
periodic(400E6,0,0,0) (PeriodicOp)
```

The timing characteristics of a periodic thread are given by a 4-tuple (p, j, d, o): p describes the period; j is the jitter, d is the minimum time between invocations of `PeriodicOp` and o is the initial offset (see Figure 3.1).

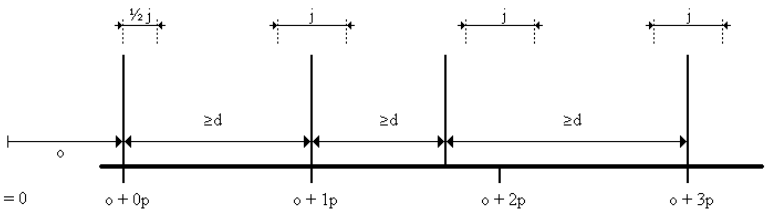


Figure 3.1 Period (p), jitter (j), delay (d) and offset (o).

Special (**`duration`** and **`cycles`**) statements may be used in operation bodies to specify time delays that are independent of or dependent upon processor capacity. The time delay incurred by a message transfer over a bus can be made dependent on the size of the message being transferred and on the bandwidth of the bus.

3.3.2 Semantics and Validation Techniques

The core VDM specification language has formally defined denotational semantics [97, 125], a proof theory [30, 114, 69] and rules for formal refinement [103].

The operational semantics have been defined [122] as part of the effort to develop an interpreter for the executable subset of VDM. These semantic definitions are themselves given in VDM and the language extensions to support object-orientation, concurrency, real-time and distribution have all been specified in VDM in the same way. This also includes constructs supporting loose specification [121] and operational semantics for features supporting real-time and distributed systems [185, 92, 184].

The availability of formal semantics makes it possible to conduct a wide range of analyses on VDM models. The term *Integrity Checking* is used to refer to the generation and discharging of proof obligations [29, 167]. Proof obligations are logical conjectures that are to be proven if a model is to be regarded as well-formed and consistent. In broad terms, this means that there exists at least one semantically correct model. Automated proof support is being developed for discharging proof obligations [7, 8, 188].

In the last two decades, what has guided the development of VDM has been the take-up of the technology by industry (see Section 3.3.4 below). The perceived high barrier to the initial use of proof and model-checking technology has led to a relatively greater emphasis being placed on validation through testing. As a result, validation has mostly been through testing of executable models, and there is considerable methodological and tool support for this approach. Model validation in VDMTools and Overture (see Section 3.3.3 below) is mainly supported by an interpreter allowing the execution of models written in the large executable subset of the language. Scenarios defined by the user are essentially test cases consisting of scripts invoking the model's functionality. In the case of VDM-RT models, the interpreter executes the script over the model and returns observable results as well as an *execution trace* containing, for each event, a time stamp and an indication of the part of the model in which it appeared.

3.3.3 VDM Tool Support

VDM is supported by an industry-strength tool set VDMTools owned and developed by CSK Systems [75] (now called SCSK) and derived from the former IFAD VDM-SL Toolbox [67]. The tools offer syntax checking, type checking and proof obligation generation capabilities, code generators, pretty printer, a CORBA-based Application Programming Interface (API) and links to external tools for Unified Modelling Language (UML) modelling to support round-trip engineering. It is worth noting that the development of VDM-Tools has itself been done using VDM [115].

The Overture project is developing an open-source Eclipse extension that is designed to provide a common interface to all available tools for VDM, covering all dialects [143] and to include proof support [188, 167]. The *RT-logger* plug-in has been developed to allow execution traces of VDM-RT models to be displayed graphically so that the user can readily inspect behaviour after the execution of a scenario, and thereby gain insight into the ordering and timing of message exchange, thread activation and operation invocation.

The *Proof Obligation Generator* [29] built into the Overture tool can detect missing pre/post-conditions or invariants before making scenario-based tests for the model. The tool automatically generates all the proof obligations that must be discharged in order to guarantee the internal consistency of a model. Examples of this includes: checks for potential run-time errors caused by misapplication of partial operators and termination of recursion.

To run structured scenario-based tests, a unit testing framework called VDMUnit can be used [72]. This framework is inspired by JUnit [49] and is used to set up hierarchical test suites for unit and integration tests. This enables easy regression testing, ensuring that all prior test cases still pass after model updates. By using the special operations `AssertTrue` and `Assert-False` that are built into VDMUnit the model state is tested against the expected state.

In addition to the manually specified test scenarios, the *combinatorial testing* tool [123] built into Overture can be used to automatically create a large collection of test cases. *Trace definitions* are defined as regular expressions describing a combination of inputs to a possible sequence of operation calls. All possible permutations are generated by the tool and executed automatically. The intent is to allow the definition of a large number of tests in a compact manner.

3.3.4 Industrial Use of VDM

VDM's rich collection of modelling abstractions has emerged because industrial application of the method has emphasised modelling over analysis [70]. Although a well worked-out proof theory for VDM exists [30], more effort has gone into developing tools that are truly "industry strength". Although research on proof support has been ongoing for some years [105, 61, 189], the tools have never fully incorporated proof, mainly due to a combination of lack of demand from industry users, and a lack of robustness and ease of use on the part of the available tools.

The majority of the industrial applications of VDM can be characterised as uses of *lightweight* formal methods in the sense of Jackson and Wing [98]. Early experience of this was gained in the 1990s when the ConForm project compared the industrial development of a small message processing system using structured methods with a parallel development of the same system using structured methods augmented by VDM, supported by VDMTools [118]. A deliberate requirements change was introduced during the development in order to assess the cost of model maintenance, and records were kept of the queries raised against requirements by the two development teams. The results suggested that the use of a formal modelling language naturally made requirements more volatile in the sense that the detection of ambiguity and incompleteness leads to substantial revision of requirements.

The use of a formal method to validate and improve requirements has been a common theme in VDM's industrial applications [63, 175, 163]. For example, two recent applications in Japanese industry, the TradeOne system and the FeliCa contactless chip firmware [111, 112], have used formal models primarily to get requirements right. In some situations, such as the FeliCa case, the formal model itself is *not* primarily a form of documentation — it is merely a tool for improving less formally expressed requirements and design documents that are passed on to other developers. In these applications, the trade-off between effort and reward of proof-based analysis has come down against proof, but in favour of a carefully scoped use of the formalism.

Several VDM industry applications have made successful use of high volume testing rather than symbolic analysis as a means of validating models. So for example the FeliCa chip was tested with 7000 black box tests and around 100 million random tests all with both the executable VDM specification as well as with the final implementation in C. The FeliCa chip is now deployed in more than 225 million mobile phones in Japan, with no recalls to date.

3.4 Incremental Formal Methods and Agile Development

It is of great importance that the modelling guidelines described in this thesis fit into existing development practices used by the industrial partner (see evaluation criteria in Section 1.6). Incremental (and sometimes even agile) system development is the de facto standard for system development, so the modelling techniques used must fit into an iterative mindset. The techniques used in the industrial applications of VDM mentioned above, have had a continued focus on the needs of the customer and the process of applying VDM has been one for removing more and more uncertainty rather than focusing on obtaining a perfect system by verification. This section gives an overview of related work in the field of incremental formal methods; agile software development; and the combination of the two.

Work in SCTL/MUST [191] addresses the iterative production of early-stage models of real-time systems. Validation by testing is supported and the model production process gives feedback to the requirements scenarios. The Credo project [58] focuses on modelling and analysis of evolutionary structures for distributed services and also includes formal models, in a combination of Creol [102] and Reo [17].

Suhaib et al. [180] have proposed a method derived from that of eXtreme Programming, in which "user stories" are expressed as Linear Temporal Logic (LTL) formulae representing properties that are model-checked. On each iteration, new user stories are addressed. The ordering of properties is significant for the practical tractability of the analysis on each iteration. In the context of research on real-time UML [65], a combination of UML and System Design Languages (SDL) with a rigorous semantic foundation has been presented [59]. Burmester et al. [46] describe support for an iterative development process for real-time system models in extended UML by means of compositional model checking, and Uchitel et al. [181] address the incremental development of message sequence charts, again model-checking the models developed in each iteration.

The relationship between formal and agile methods has been explored for many years. More than 20 years ago, Richard Kemmerer investigated how to integrate formal methods into conventional development processes [108]. More recently the issues were brought into focus by Black et al. in their article for IEEE Computer in 2009 [33]. Some researchers have sought to develop hybrid methods that benefit from both rigour and agility. For example, Ostroff et al. [153] seek to harness Test-Driven Development (TDD), a well-known agile technique, with a more formal method of Design by Contract. Niu and Easterbrook [147] argue for the use of machine-assisted model checking to address the problem of partial knowledge during iterations of an agile process. López-Nores et al. [136] observe that evolution steps in an agile development typically involve the acquisition of new information about the system to be constructed. This new information may represent a refinement, or may introduce an inconsistency to be resolved through a retrenchment. Solms and Loubser [176] describe a service-oriented analysis and design methodology in which requirements are specified as formal service contracts, facilitating auto-generation of algorithm-independent tests and documentation. Model structure is formally defined and can be verified through a model validation suite. The combination of agile and formal elements into an incremental development process has been done by del Bianco et al. [60].

Some work seeks to develop more agile formal methods. For example, Eleftherakis and Cowling [66] propose XFun, a lightweight formal method based on the use of X-machines and directed at the development of component-based reactive systems. Suhaib et al. [180] propose an iterative approach to the construction of formal reference models. On each iteration, a model is extended by user stories and subjective to regression verification. Liu et al. have described the integration of formal methods into industry development standards with the SOFL language and methodology [133]. Instead of a pure mathematical notation, they use *condition data flow diagrams* as a graphical notation of the high level architecture of the system. Through several semi-formal refinement steps the final implementation is developed. More recently Liu has applied the SOFL method using a more agile approach [132] where an incremental implementation is used.

4

Guidelines for Stepwise Development of VDM-RT Models

This chapter provides guidelines for mono-disciplinary modelling of distributed real-time embedded systems in VDM-RT. Both the embedded controller and the environment with which it interacts are modelled in the discrete-event formalism VDM. Chapter 6 describes a case study in the electronic warfare domain where these guidelines are applied.

4.1 Introduction

Formal models have a potentially valuable role to play in managing the complexity of embedded systems design. Rapid feedback from the machine-assisted analysis of such models has the potential to detect defects much earlier in the development process. However, models that incorporate the description of functionality alongside timing behaviour and distribution across shared computing resources are themselves potentially complex. Moving too rapidly to such a complex model can increase modelling and design costs in the long run.

While there are many formal notations for the representation of distributed and real-time systems, guides to practical methods for model construction and analysis are essential if these approaches are to be deployed successfully in industrial settings. Such methods should be incremental, allowing the staged introduction of detail, and make use of tool-supported analysis of the models produced at each stage.

This chapter proposes a pragmatic and tool-supported method for the stepwise development of models of distributed embedded systems. Guidelines –summarising the key aspects of the method– are supplied to ease future referencing. At each step in our method, we develop a model that considers an additional aspect of the design problem, such as distribution or concurrency. Formal notations provide explicit support for these aspects, and tools provide an analytic capability based on static analysis and systematic testing of models at each stage. Our approach uses and extends VDM and its tools (VDMTools and Overture) described in more details in Section 3.3.

Section 4.2 presents our approach to the development of formal models of distributed real-time systems. Sections 4.3 to 4.6 describe the different tiers of models constructed at each phase in the development. Section 4.7 describes validation techniques and tools. Finally, Section 4.8 summarises the work done and identifies further work.

4.2 An Incremental Approach to Model Construction

As indicated above the goal of our work is a method for developing formal models of distributed real-time systems that is incremental and allows tool-supported validation of the models produced at each stage. Using such models for design space exploration helps in defining the correct software and/or system design which meet the timing requirements specified.

We propose a stepwise approach [55] which exploits the capabilities of each of the VDM modelling language extensions described in Section 3.3. Our approach aims to assist the management of complexity by enabling the developer to consider a different facet of the modelled system at each stage. The steps themselves are as follows:

1. System Boundary Definition.
2. Sequential Design Modelling.
3. Concurrent Design Modelling.
4. Distributed Real-time Design Modelling.

In the design of models of embedded systems, a key decision is the drawing of the boundary between the environment, which consists of elements that can not be controlled by the designer, and the controller that is to be developed. In particular, this allows the developer to state assumptions about environment behaviour and the guarantees that describe the correct operation of the controller in a valid environment. The first stage of our method involves making the system boundary, assumptions and guarantees explicit. Such an explicit abstract description can be given informally or using both formal and informal elements side by side. It is important to note that the description of the environment is only a rough approximation described in a DE formalism used for injecting stimuli into the system model and to observe the responses from the system. To add additional details to the environment model, a domain-specific notation must be used, though this is not described in this chapter. Support for modelling and analysis of more realistic environment models is the subject of Part III of this thesis. Using a CT formalism, a high fidelity model of the environment is created. The DE controller is then co-simulated with the CT environment model.

Based on this abstract description of the system boundary, an object-oriented architecture is introduced, creating a sequential model with structure expressed using the features of VDM++. Consideration of the synchronisation of concurrent activities is deferred in order to focus on functional aspects. In the next stage, this sequential model is extended to encompass concurrency and thread synchronisation (still using the VDM++ language, which includes concurrency modelling features). Subsequently, the concurrent design model may be extended with real-time information and a distributed embedded architecture using the VDM-RT extensions. At this stage it may prove necessary to revisit the concurrent design model, since design decisions made at that stage may prove to be infeasible when real-time information is added to the model (for instance, the model may not be able to meet its deadlines). From the concurrent and distributed real-time VDM design model an implementation may subsequently be developed. Testing of the final implementation and the various design-oriented models may be able to exploit the more abstract models as a test oracle. Note that the approach suggested here enables continuous validation of the models if these are written in executable subsets of the different VDM dialects.

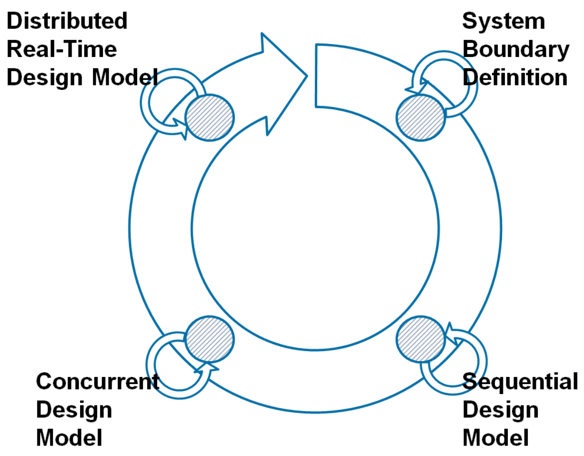

Distributed Real-Time Design Model
System Boundary Definition
Concurrent Design Model
Sequential Design Model

Figure 4.1 Overview of the models produced.

Figure 4.1 gives an overview of the relationships between the products in our proposed method. The four phases of model development are indicated by the large arrow — prior models feed into subsequent models. The circular arrows show iteration that might follow the detection of modelling errors in the validation of the model produced at each stage. Note that this is not intended as a process model, but rather a rational structure for the relationships between the models produced. Internal iterations, and even iterations between models, are likely to occur in practice. Details are introduced in a staged manner, where the executions at each level might, informally, be seen as providing a finer level of granularity than its predecessor, ensuring a gradual increase in model fidelity.

In the following sections, we describe each stage in our approach, identifying the features particular to each type of model and the typical structures of models produced. The original publication of this process [119] includes a running example from the home automation domain explaining the process steps. Chapter 6 of this thesis introduces another case study that makes use of the guidelines described here.

In [126] the approach is described in more detail, where a concrete example of requirements capture is given. A case study is also included, where a complete countermeasures system is modelled using the guidelines.

The approach to developing VDM-RT models described in this chapter has also been tested in an academic setting. For several years students following the VDM courses of Aarhus University have been taught the process, and applied it in mandatory projects. Feedback from the students has been used for improving the process.

4.3 System Boundary Definition

As indicated in Section 4.2, defining the boundary between the environment and the engineered system is an important step on the way to specifying the assumptions under which the system operates and the guarantees that it offers when those assumptions are satisfied. Given the explicit statement of these contracts, it should be possible to formulate sound engineering arguments for the overall system having desirable properties.

Guideline mono_1: Explicitly specify the assumptions under which the system operates and the guarantees that it offers when those assumptions are satisfied.

4.3.1 System Requirements and Guarantees

There are many approaches to systematic requirements capture, and we do not propose a particular method here. However, past studies [118] and recent industry experience [111, 117] suggest that the development of a formal model can have a strong positive effect on the identification of defects in requirements. We have provided rough guidelines on the extraction of "first pass" models from natural language requirements [71, 72] but other more structured approaches exist, targeted at embedded systems. For example, the approach of Hayes, Jackson and Jones [104] uses the expression of desired system-level behaviour coupled with modelling the environment as a means of deducing the specifications of control systems. Such an approach provides ways of reaching initial models that can be expressed using the formal notations that we consider here.

The assumptions made about the environment are key to the formulation of engineering arguments. In order to derive precise statements of environmental assumptions, the model developer must enter into a dialogue with domain experts. The kind of environment is likely to determine the formalism in which these assumptions are most naturally expressed. For example, if the assumptions have to do with a moving mechanical device, it would most likely best be described in a formalism including differential equations; if the assumptions are logical in nature it may be more natural to express it in VDM.

The purpose of an initial model of the engineered system is to provide a basis for the description of its functionality. At first the focus is on functionality rather than the static structure of a design or the dynamic architecture of processors on which such a design might be implemented — those features are introduced in the later models. At this stage in the modelling process, functional and timing requirements are captured. For the systems to which we have applied our approach so far (e.g. [55, 135]), an appropriate initial model is usually structured around one top-level function taking as input a sequence of events acting as stimuli on the controller from its environment. The output is the trace of events generated by the system.

Guideline mono_2: Create an initial model with one top-level function taking as input a sequence of events acting as stimuli on the controller from its environment and generating the trace of events as output.

4.4 Sequential Design Modelling

The purpose of the sequential design model is to provide a description of the static structure of the system under development in terms of classes, without making any commitment to a specific architecture for the dynamic system. For the techniques of class structure derivation, the reader is referred to a standard text such as [64].

Activities in the basic model that are functionally independent are typically encapsulated in separate classes in the sequential design. For an embedded system, each kind of sensor and actuator will typically get its own class. We find that it is advisable to add a `World` class that sets up and manages the interaction between objects representing the environment and those representing the system under development. In this section we consider the representation of such a class structure in VDM as well as aspects that are specific to embedded systems.

4.4.1 Class Descriptions in VDM

Class diagrams in UML provide a convenient notation for the expression of object-oriented class structures. Tool support exists in both VDMTools and Overture, allowing class diagrams developed in a UML tool to be translated into VDM class definitions with the same inheritance and association structure. Consistency is maintained between the two views: a modification to either the VDM or the UML is reflected in the other model, enabling round-trip engineering between a UML class diagram view of a system and a VDM textual view.

Where possible, invariants on types and instance variables should be identified and specified. Pre-conditions should be specified for all operations and functions that are non-total, and post-conditions should be specified wherever appropriate, especially if they provide a clear, distinct, representation of functionality that is not simply a restatement of the expression in the function or operation body.

Guideline mono_3: Specify invariants on types and instance variables and pre- and post-conditions for operations and functions.

4.4.2 Typical Design Structure

Figure 4.2 illustrates a typical class structure for reactive embedded systems of the kind covered by our method. We advocate the structural separation of the environment and the technical system, shown here by the classes `Environment` and `SystemName` respectively.

> **Guideline mono_4:** Create a structural separation of the environment (`Environment`) and the engineering system (`SystemName`).

A `World` class sets up instances of both environment and system and manages the interaction between them, and so manages the execution of test scenarios during validation.

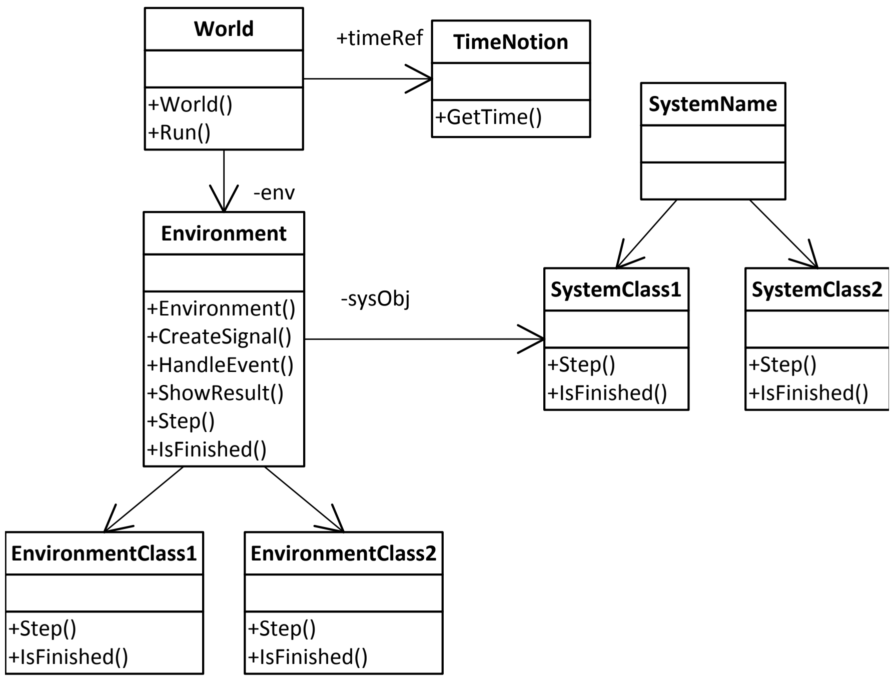


Figure 4.2 Generic class diagram for sequential design of an embedded system. The `World` class instantiates the model, and holds static references to both system, environment and time. The `Environment` class generates input to the system (`CreateSignal`), and receives the system response (`HandleEvent`). When all input has been processed, the Environment shows the results of the simulation (`ShowResult`).

The `Environment` should contain operations to provide a sequence of input for the embedded system (`CreateSignal`) and operations to receive feedback from the system (`HandleEvent`).

Guideline mono_5: In the `Environment` class, add operations for providing input (`CreateSignal`) to the system and receiving the response (`HandleEvent`).

For all the classes that need to conduct active tasks this functionality is normally organised into `Step` operations that describe the functionality to be performed each time the instances of the class are scheduled to be executed. An operation (`IsFinished`) should be included in all environment and system classes that play an active role in a scenario to indicate when processing is finished locally.

Guideline mono_6: All active classes should have `Step` operations performing the periodic functionality of the class, and `IsFinished` operations to indicate that the process is finished.

For the `Environment` class it is also necessary to provide an operation (`ShowResult`) that can show the result of running a given scenario.

To avoid passing object references around the system, we recommend the use of public static instances contained in the `World` class and the overall system class `SystemName`. This ensures that all object references are accessible throughout the model. This also includes a global discrete notion of time: in the sequential model a basic `TimeNotion` class is appropriate, but in the concurrent model a stronger notion of time and synchronisation is required (see Section 4.5 below).

Guideline mono_7: Use public static instances contained in the `World` class and the overall system class `SystemName` to avoid passing object references around the system.

Normally, we expect the flow of control in the sequential model to be steered by the `Environment`. This is because most reactive systems are designed to respond to stimuli by sending commands to the actuators to control the environment in the desired fashion. However, there are exceptions to this where the system is intended to react when a stimulus from the environment is absent; for example, a cardiac pacemaker might be expected to produce a stimulus when a natural pulse is expected but not detected. In such cases we recommend modelling the omission of the expected stimuli by using special stimuli from the environment. Alternatively, the system could periodically check the environment for stimuli, and only react when a stimuli is absent. At the sequential level this is typically done using a loop until both the environment and the system are finished (using different `IsFinished` operations). Inside the loop, `Step` operations are used for progressing time and passing control.

4.5 Concurrent Design Modelling

The objective in developing the concurrent VDM++ design model is to take the first steps towards a particular dynamic architecture, deferring for the moment the complexity introduced by real-time behaviour and distribution concerns.

The first steps in developing a concurrent model are the identification of computations that can be performed independently of one another and their separation into independent threads. Often, this separation is forced by hardware constraints and/or pre-defined architectural requirements. While it is worthwhile to identify as many independent threads as possible, this should be balanced against the complexity of understanding and validation. Syntactically such threads are expressed as VDM++ statements written after a **`thread`** keyword.

Guideline mono_8: Identify processes that can be performed independently, turning these into separate threads. The bodies of the `Step` operations are typically turned into threads.

As threads are identified, it is necessary to specify the communication between them in terms of both communication events and any necessary object sharing. Classes that define objects that contain shared data are responsible for ensuring synchronised access. Explicit synchronisation points may also be required to orchestrate the ordering of events among threads or to ensure data freshness. Permission predicates and mutual exclusion constraints (see Section 3.3.1) can be defined for the operations in a class in a special **`sync`** part of the class definition.

Guideline mono_9: Specify synchronisation between threads using permission predicates and **`mutex`** constraints.

The signatures for the `IsFinished` operations are changed so that no value is returned. Instead, the Boolean expressions typically become permission predicates. This way the threads requesting each operation are blocked until the corresponding instance is indeed finished.

It is recommended that the simple `Timer` class is replaced by a combination of the standard `TimeStamp` class and the `ClockTick` class. `TimeStamp` is a subclass of the `WaitNotify` class used in Java [84] in order to easily synchronise the steps taken when the flow of control is distributed to multiple threads. `ClockTick` is used to ensure lock-stepping between different concurrent threads. The use of these classes is illustrated in the case study in the original paper [119].

Guideline mono_10: Replace the simple `Timer` class with a *Wait/Notify* approach to easily synchronise the steps taken when the flow of control is distributed to multiple threads.

4.6 Distributed Real-Time Design Modelling

The purpose of the distributed real-time design model is to add information on real timing constraints, and to describe allocation of system functionality to multiple processors using the predefined `BUS` and `CPU` classes. Following a timing analysis, it might be concluded that a proposed system architecture is infeasible, necessitating revision of the system architecture, redeployment of functionality to `CPU`s, or modifications of the capabilities of the `CPU`s.

The explicit notion of time modelled at the concurrent level using the `TimeStamp` class is removed. Now time is implicit and a keyword **`time`** is used to refer to the current global time, which is a finite accuracy discrete clock represented as a natural number.

The `SystemName` class is extended with additional instance variables for the CPUs and buses in the system architecture. A class that is to define such a deployment is identified by using the **`system`** rather than the **`class`** keyword. The class' constructor defines the actual allocation of static instance variables to `CPU`s. The constructor can be used to define the priority of operations on `CPU`s that operate a priority-based scheduling scheme.

Guideline mono_11: Extend the `SystemName` class (using the **`system`** rather than the **`class`** keyword) with additional instance variables for the CPUs and buses in the system architecture.

In most cases a common virtual `CPU` is used for all objects that are not a part of a declared system class describing a deployment architecture. However, the `Environment` class may also be defined as a **`system`** in order to enforce restrictions such as limiting the concurrent execution of environment functions.

Some of the threads (typically those that previously had `Step` functionality) are made periodic, as described in Section 3.3.1.

Guideline mono_12: `Step` operations should typically be converted into periodic threads.

Deployment to processors allows the model designer to take advantage of potential asynchrony where a calling operation need not wait for the called operation to complete. Note that the user does not need to know where every instance is deployed; the interpreter looks up the relevant deployment information (which is concentrated in the **`system`** class and not spread about the model) before passing calling parameters over the `BUS` to the relevant CPU.

Guideline mono_13: Use asynchronous operations for distributed communication so the calling operation need not wait for the called operation to complete.

Where the designer has *a priori* knowledge of real-time characteristics of design elements, for example where off-the-shelf components are being reused, it is possible to give precise estimates of fixed execution times via **`duration`** statements. For example, when modelling a closed loop system, a duration statement might be used to record the delay between sending a command to an actuator and seeing its effect at a sensor. The modelling language also supports **`cycles`** statements, which are used to give precise estimates of execution time relative to a processor (in the form of the number of expected clock cycles).

Guideline mono_14: Specify estimates of fixed execution times (using the **`duration`** statement) or execution time relative to the computational power of a processor (using the **`cycles`** statement).

The task switching overhead for the target real-time kernel should be ascertained and then used to configure the task switching overhead in Overture during execution of the model. Whenever there is a task switch between two tasks on a `CPU`, the time is stepped forward by the task switching overhead time.

4.7 Validation Technology

Validation takes place on each of the models produced through the process that we have outlined. In this context we use the term *validation conjecture* for a logical expression describing a property which is expected to hold. A systematic testing approach [72] is used to validate the models derived during the staged development process. “Validation” in this context refers to the activity of gaining confidence that the formal models developed are consistent with the requirements [135]. The initial system model, which is typically expressed in VDM-SL, can be validated by proof or by testing. Overture automatically generates proof obligations from the model and these may be discharged using theories of the basic VDM logic and types [30, 69].

Recent extensions to Overture [124] allow the designer to create a Java applet or JFrame that can act as a Graphical User Interface (GUI) of the VDM model. This allows an external program to manage the GUI, reflecting the current state of the model back to the expert user, who can then invoke operations to change the state, allowing the execution of scenarios [9].

4.7.1 Test-Based Validation

Unit and integration tests should be made for the Sequential model, using VDMUnit or the combinatorial testing feature built into Overture (see Section 3.3.3). The test-suite must ensure that every part of the model is exercised and that the model reacts according to the specified requirements.

Guideline mono_15: Perform unit- and integration tests of the model to gain confidence in the functionality of each component as well as the integrated system model (for example using the VDMUnit framework).

These sequential test scenarios can be reused for the Concurrent and Distributed Real-Time models with minor modifications. Instead of having several invocations of the `Step` operations in the different classes, all threads run concurrently handling a sequence of input to the model. This allows for incremental development of test scenarios as well as the models themselves, which further increases the reuse from one modelling phase to the next.

Guideline mono_16: Reuse test scenarios from the sequential model in the test of the concurrent and real-time model.

The Concurrent and Distributed Real-Time models should be executed using the same scheduling policies that are likely to be available in the target processors. There is not yet a formal static deadlock analysis for the concurrency part of VDM, so confidence in deadlock freedom is limited by the comprehensiveness of the scenarios used as test cases. A general guarantee of deadlock freedom cannot be constructed because of the expressiveness of the permission predicates available in the language.

Riberio et al. [168] have proposed an approach to specifying temporal constraints of a VDM-RT model and checking these at run-time. A prototype tool has also been implemented as an extension to the Overture tool. Timing invariants are formulated as predicates over events to specify the timing properties of the system that must hold. Examples of these are: deadline property which is defined as a time by which en event must have happened, and separation property which describe a minimum separation time between two events happening. Using these timing invariants, it is possible to test schedulability and other timing constraints of the system. Using the RT-logger (see Section 3.3.3) violation of these timing invariants can be visualised.

4.8 Summary

We have described guidelines for the construction of formal models of potentially complex distributed real-time control systems. The models support validation of system properties, including timing behaviour on a specific embedded and distributed system architecture. Our approach is based on the staged introduction of detail into a series of extended VDM models derived from an initial abstract specification based on initial requirements. Robust tools support validation, predominantly using scenario-based testing at each stage.

Considering our methodology, we have not so far addressed the process of deriving a specification of a system that is embedded in a real-world environment. Systematic approaches such as that of Hayes, Jackson and Jones [104] are worth further investigation here, in that they can also lead to clearer initial separation of the controller under development and the model of the environment. This separation is covered in further detail in Part III of this thesis.

We have so far only provided preliminary guidance on the outline structure of the VDM models in our chain. Prior work shows that there is potentially great value in developing model patterns appropriate to embedded applications [195]. Such patterns have the potential to aid model construction, and also to support validation by encouraging re-use of test scenarios and, where appropriate, proofs. Since we use a staged approach, we expect patterns to encompass several models from our chain [96].

Our approach has been pragmatic because it is meant to be an aid to system architects when they try to take informed decisions between alternative candidate system architectures. Thus, we have not formalised the incremental addition of detail that occurs in each of our model development steps with formal refinement. In particular, we would like to drive useful proof obligations out of each of these steps. An examination of this issue must treat atomicity in the abstract and sequential models (for example in handling the maintenance of invariants).

The validation of execution traces does not replace formal verification. Nevertheless, when development resources, especially time, are short, rapid checking of validation conjectures offers a means of assessing key properties of the formal model. Once conjectures have been defined and have been used to validate the execution of the model (which also serves as a validation of the conjectures themselves), they can be used (with event transformation) to validate log files generated by the final implementation of a system. The validation framework discussed here might be extended to support the evaluation of fault tolerance strategies at the architectural level.

5

Formal Methods Meet Agile Development

This chapter assesses the readiness of formal methods for integration with agile development techniques. A concrete example of the use of formal methods in the agile project management process Scrum is given. Chapter 6 describes a case study where the combination of formal and agile principles were applied.

5.1 Introduction

In spite of their successful application in a variety of industry sectors, formal methods have been perceived as expensive, niche technology requiring highly capable engineers [172]. The development of stronger and more automated formal analysis techniques in the last decade has led to renewed interest in the extent to which formal techniques can contribute to evolving software development practices.

The principles of agile software development emerged as a reaction to the perceived failure of more conventional methodologies to cope with the realities of software development in a volatile and competitive market. In contrast with some established development approaches, like the methodologically suspect waterfall model, agile methods were characterised as incremental, cooperative, straightforward to learn and modify, and adaptive to changes in the requirements or environment [2]. Four value statements [10] summarise the principles of the approach. Proponents of agile techniques value:

- *Individuals and interactions* over processes and tools;
- *Working software* over comprehensive documentation;
- *Customer collaboration* over contract negotiation; and
- *Responding to change* over following a plan.

This chapter discusses if agile and formal methods can work to mutual benefit, or if the underlying principle of rigorous model-based analysis are incompatible with the rapid production of code and the favouring of code over documentation. A review of the four value statements of the agile manifesto is given in Section 5.2. In each case, we ask whether formal methods as they are now are really able to help achieve the value goal, and what research might be needed to bridge the gaps between the two approaches. The following sections give a concrete example of the combination of formal and agile methods: Section 5.3 provides an overview of the agile management method Scrum. The addition of formal modelling techniques to Scrum is described in Section 5.4. Finally, Section 5.5 gives a summary of the work on combining formal and agile methods.

5.2 The Agile Manifesto Meets Formal Methods

The four value statements of the agile manifesto are supported by 12 principles [11]. In this section we consider each value statement and the principles that relate to it. For each value statement, we review the extent to which formal methods support it today, discuss some deficiencies and suggest future research to remedy these.

5.2.1 Individuals and Interactions over Processes and Tools

This value statement emphasises the technical competence and collaboration skills of the people working on the project and how they work together. If this is not considered carefully enough, the best tools and processes are of little use[1]. Two of the 12 principles supporting the agile manifesto are relevant:

- Build projects around motivated individuals. Give them the environment and support their need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

The most important resources available to a project are the people working on it and not the tools or methods they use. But once the right people have been chosen, neither the tools nor the processes should be disregarded.

A criticism of the work on the ConForm project (see Section 3.3.4), as of other demonstrations of the effectiveness of formal modelling, is that clever "pioneering" people were employed to do the formal methods work and they were bound to do a better job than those applying traditional techniques. Our industry colleagues have frequently refuted this claim, arguing that, while

[1] The aphorism *"A fool with a tool is still just a fool"* is sometimes attributed to Grady Booch.

highly skilled engineers will perform tasks well given the most elementary of tools and processes, the world is not full of excellent engineers. Most of us benefit from having tools and processes that embody the wisdom gained by previous projects and more capable colleagues.

To live up to this value statement, it is required that the team members are technically competent in using efficient tools to develop working software for the customer in short periods of time. We wonder if all our fellow formalists appreciate the levels of competence among software engineers and hence the work required to deliver methods that can be used in the majority of products. Numerous research projects have demonstrated the potential of new formal methods and tools, such as Rodin [6]. However, the process of bringing them to a state where they are deployable even by research and development engineers requires a major effort, as seen in Deploy [170].

The second principle above refers to "soft" skills and particularly to collaboration and communication. Given engineers who have a willingness and ability to communicate easily among themselves, the tools supporting formal modelling have to make it easy to share models and verification information where appropriate. But how many formal methods tools integrate well with existing development environments, allow models to be updated and easily transmitted, or even exchanged between tools? Wassyng and Lawford have presented high-level requirements for tools to support formal modelling of safety-critical systems [194]. They emphasize the importance of a comprehensive tool suite, where the individual tools are designed to complement and interface with each other. We agree, and further feel that collaborative modelling and analysis is not given enough attention in formal methods research.

There are noteworthy formal methods tools and specification languages that integrate well with existing development environments: The verification and testing environment Java PathFinder [192] integrates model checking, program analysis and testing of Java bytecode; and the specification languages JML [129] for Java programs and Spec# [23] for C#, where source files are annotated with pre- and post-conditions and object invariants that are checked at run-time using various verification tools.

Most formal methods tools originated in academic research and few have been matured for industrial use. As a result, the focus has been on the functionality offered by the tools at the expense of the accessibility or user friendliness. If formal methods are to move a step closer to agility, the tool support needs to become easier to pick-up and start using, so attention can be put back on the people actually doing the formal models instead of the limitations of the tools. Increased automation is required, and research effort being put into the interaction between modelling and verification tools and the engineer.

5.2.2 Working Software over Comprehensive Documentation

The second value statement of the agile manifesto asserts that, whilst good documentation is a valuable guide to the purpose and structure of a software system, it will never be as important as running software that meets its requirements. The value statement is supported by the following principles:

- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Working software is the primary measure of progress.
- Simplicity – the art of maximizing the amount of work not done — is essential.
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

We suggest that adherence to these principles is probably easier in a project that is kept in-house, rather than a major distributed software development with extensive subcontracting. For large projects with many person-years of work involved, documentation is indispensable and is often a crucial part of the contract between the developer and customer.

For the formal methods practitioner, these must be some of the more difficult agile principles to accept. Much of the work on model-oriented formal methods emphasises the quality and clarity of models and the production of proven refinement steps on the way to well engineered code, where proofs document the correctness of the design steps. An agile process adhering to the principle above is more likely to be based on rapid development cycles in which the quality of the formal models produced takes second place to the rapid generation of code. In turn, this may mean that models will tend to be more concrete and implementation-specific than necessary. There is a risk that the incremental development of system models will end up with models, and of course proofs, that are much too hard to understand or verify to serve a useful purpose.

Automation is once again a key factor: where code is automatically generated, the model may become the product of interest. Further, the benefits of the model must be seen to justify its production costs, for example by allowing automatic generation of test cases, test oracles or run-time assertions. An agile process that wants to gain the benefits of formal modelling techniques has to be disciplined if the formal model is to remain synchronised to the software produced. It is worth noting that nothing in this approach precludes the use of formal methods of code analysis, for example to assist in identifying potential memory management faults. Here again, the high degree of automation can make it an attractive technology.

In general one can say that this value statement is most applicable with executable models where one then needs to be careful about implementation bias [87, 81, 14]. From a purist's perspective this is not a recommended approach. As a consequence, formal refinements from non-executable models cannot be considered agile since potentially many layers of models may be necessary before one would be able to present anything to the customers that they can understand [144, 18, 5]. Examples of less opaque formal methods would be: the SOFL language and methodology [133, 132], tabular expressions [100], or VDM which has a syntax that will be familiar to most software engineers.

5.2.3 Customer Collaboration over Contract Negotiation

The third value statement of the agile manifesto, while recognising the value of a contract as a statement of rights and responsibilities, argues that successful developers work closely with customers in a process of mutual education. Two of the agile principles would appear to relate to this value statement:

- Business people and developers must work together daily throughout the project.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Close customer collaboration has been a feature of several successful industrial formal methods projects [118, 183, 175, 112]. However, in these projects formal methods have only been used as a high-level executable model of the system and no advanced formal methods techniques such as verification have been applied. Examples of successful application of formal methods using verification also exist (see the work of Wassyng and Lawford [193] as well as Andronick et al. [15] described briefly in Section 3.2), though they are seen more rarely. In order to successfully exploit collaboration between business people and formal methods specialists interpersonal skills for this kind of multi-disciplinary teamwork is essential.

A major weakness of many formal methods tools is the inability to attach a quick prototype GUI to a model giving domain experts the opportunity to interact directly with the model and undertake early validation without requiring familiarity with the modelling notation. Actually, this general idea was implemented over 20 years ago in the EPROS prototyping environment [88]. As mentioned in Section 4.7, recent extensions to Overture [146] allow the designer to create a Java applet or JFrame that can act as a graphical interface of the VDM model. Many formal methods modelling tools could benefit from similar extensions if they aim to support some of these agility principles.

5.2.4 Responding to Change over Following a Plan

The fourth value statement of the agile manifesto acknowledges that change is a reality of software development. Project plans should be flexible enough to assimilate changes in requirements as well as in the technical and business environment. The following two agile principles are relevant here:

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Formal methods have no inherent difficulty in coping with requirements change. Indeed formal models may form a better basis for predicting the consequences of radical changes than attempting an analysis of complex code. However, when formal models are adjusted, the associated validation and verification results will need to be redone if full value is to be gained from the model. Thus, the speed of repeat analysis, and the extent of automated tool support are paramount.

As can be seen, applying the agile principles directly is not something that fits every type of project, but this does not mean that some agile practices cannot be applied to large projects. On the contrary, our experience is that agile development is most often used as an element in a hybrid configuration. For example, initial requirements might be analysed in a model-based, documented way, while day-to-day development might employ an agile method like Scrum.

We are not convinced that formal methods and formalists in general "embrace change". Black et al. state [33] that *"formal methods cannot guarantee against requirements-creep"*. While the concept of requirements creep has negative connotations associated with confused and drifting functionality, being ready to address changing requirements is a necessary part of the agile mindset. Coping adequately with requirements change in a formal setting requires models that are well structured and mechanisms for checking validity that are rapid. Further, to be able to respond to changes in a quick and seamless manner, formal methods practitioners need to accept that models can be less than perfect, especially in the early stages of an iterative development.

5.2.5 Two Remaining Principles of Technical Excellence

Two of the 12 principles do not fit the value statements quite so easily as the others listed above. Both of them deal with aspects of the technical quality of the product:

- Continuous attention to technical excellence and good design enhances agility.
- The best architectures, requirements, and designs emerge from self-organizing teams.

Formal techniques certainly support this focus on quality. Black et al. [33] also mention the potential synergy between agile and formal methods, opening up the possibility of agile methods being applied to more safety critical domains — something which is currently, to a large extent, off limits to pure agile methods. We conjecture that the second value statement of the original agile manifesto (working software over documentation) has provided a pretext for hack-fixing and ad-hoc programming to call itself an agile process. This has hurt the reputation of agile development, and we would suggest that the addition of a fifth principle favouring quality of the end product over ad-hoc solutions could prevent some of the abuses of agility.

Formal methods can be more agile than is normally perceived, if lightweight analysis approaches and tool automation is used over more time consuming formal proof activities. Agile methods must be disciplined, following well defined process steps, if they want to gain benefits of formal specification techniques.

The rest of this chapter provides a concrete example of how formal methods can be used as part of the agile project management process Scrum. Scrum was chosen since it is well structured and well established in industry. The goal is to describe a concrete example of how formal methods can be added to a widely used agile method. Our hope is that this will help break down the barriers between formal and conventional software development by introducing the use of lightweight formal analysis techniques in an agile setting.

5.3 Agile Development — Scrum

Scrum has been used to develop complex software systems since the early 1990s and is one of the most widely used agile methods in industry [47]. Like all agile methods Scrum focusses on short feedback loops in system development, user requirements and the development process itself. By involving the user continuously in the iterative process it is ensured that the final product fulfils the user needs. This requires great adaptive capabilities of both process and development team in order to support change in user needs.

5.3.1 The Relevant Roles of the Scrum Process

In a Scrum setup, project participants take on one of three roles:

Product owner: The person who represents the customer's interest, and determines the goal of each iteration called a sprint. One of the most important tasks of the product owner is to prioritise the different tasks ensuring that the most important functionality is implemented first.

ScrumMaster: It is the task of the ScrumMaster to facilitate the Scrum process and protect the agile principles. He also needs to protect the team by removing any impediments encountered, ensuring that they are not distracted from the task at hand.

Team: A Scrum team consists of five to nine people working together to create a functional system that satisfies the user needs. Usually, the team consists of people with a broad set of competences in order to ensure that the team is self-organised and contained.

5.3.2 Relevant Activities and Artifacts of the Scrum Process

Scrum consists of several activities which describe the workflow and artifacts produced during the Scrum development:

Product vision: This is the initial idea phase, where the vision of the product is defined.

Product backlog: A prioritised list of functionality or artifacts that are needed in the final product. It is good practice to describe the items in the product backlog as user stories which add value for the user — it is then the job of the team to develop a product that fulfils these needs and supports the stories.

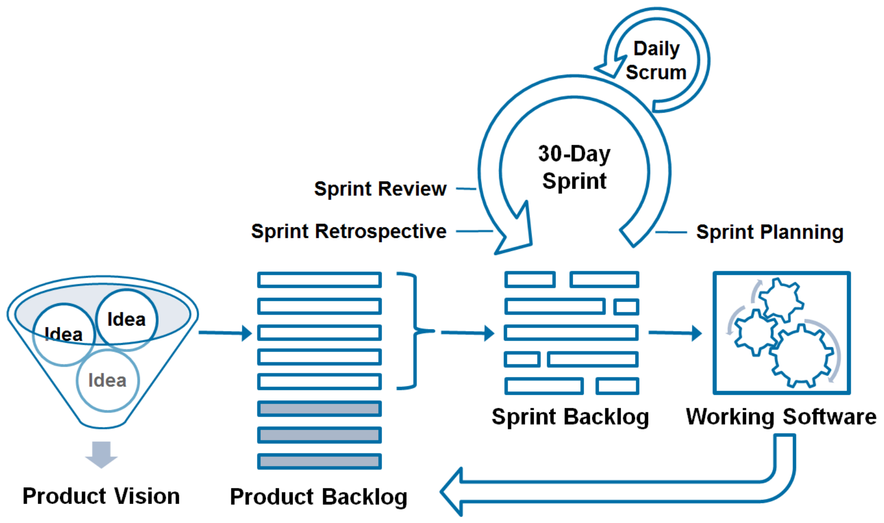


Figure 5.1 Overview of the Scrum process with a 30-day sprint duration.

Sprint: This is a development iteration which is time-boxed to last between 2 and 6 weeks — a sprint length of one month is commonly used.

Sprint planning: The first day of every sprint is dedicated to plan the goal of the sprint.

Sprint backlog: Tasks derived from the user stories from the product backlog which the team has committed to implement in the sprint planning meeting. Two types of tasks are defined: *implementation* and *investigation* tasks. Implementation tasks are completely defined with little or no uncertainty. Investigation tasks often involves the use of new technology, and are used to investigate the feasibility of said technology and to better estimate the implementation tasks which will be derived from the investigation.

Daily Scrum meeting: A daily meeting called a “daily scrum” with a short duration of approximately 15 minutes is held every morning. Each team member describes what was accomplished since last meeting, what will be done until the next meeting and describes any impediments discovered or encountered.

Sprint review: At the end of each sprint, a review meeting is held, where the team demonstrates what has been accomplished to the product owner.

Sprint retrospective: Following the sprint review an additional meeting is held, where focus is on people, processes and tools and not on the items produced in the last sprint.

5.4 Formal Methods in a Scrum Setting

The process proposed here is inspired by the process supporting the modelling guidelines presented in Chapter 4 and then adjusted to a more agile iterative process like Scrum.

Formal methods are used for modelling and validation of investigation tasks, before they are implemented in future sprints. The goal of this approach is to mitigate risk by investigating critical system properties or the use of new technologies using formal modelling. Once these investigation tasks have been modelled, new implementation tasks are derived from the results of the analysis. These new tasks will be implemented by conventional software engineers in subsequent sprints.

It is important that the formal specification helps the programmers understanding the requirements and design of the system, and that the formalists can communicate their findings to the software engineers. This communication process can be eased by using a graphical representation of the formal specification as it is done in SOFL, using a syntax like VDM which is readable by most programmers, or by using an executable specification using simulation results to aid in the communication.

5.4.1 Relevant Changes to Roles

The roles of the ScrumMaster and product owner are identical to the conventional Scrum process. Only the role of the Team is changed:

Team: The team has several tasks: to create a high level executable formal specification of the system; to validate key concepts of the system design; to investigate the use of new technology and finally the team must implement the functionality needed to solve the tasks defined in the sprint. This means that the team needs to include engineers with expertise in formal methods as well as conventional software developers in order to solve both types of tasks.

5.4.2 Relevant Changes to Activities and Artifacts

The main changes in the method described here compared to conventional Scrum are in the individual sprints. Hence, the product vision, product backlog and daily Scrum meeting are unchanged.

Sprint: It is important for the formal team members to have a good understanding of what the conventional software team members are working on, and vice versa, to support better knowledge transfer within the group. This ensures that more engineers get a good understanding of formal modelling techniques and opens up the option of using a formal engineer to solve some of the conventional software tasks (or the other way around) in order to balance the tasks within the team during a sprint.

Sprint planning: The sprint planning is similar to the conventional Scrum version, with the product owner and team deciding which stories from the product backlog go into the sprint. It is important to balance the number of investigation and implementation tasks to match the team composition of formalists and software developers.

Sprint backlog: The sprint backlog is divided into two parts: one for the formal specification investigation tasks and one for the conventional software implementation tasks.

Sprint review: After the sprint is finished, the team will present and demonstrate their work to the product owner. Apart from customer and product owner feedback, the focus on the sprint review should be on deriving future conventional software implementation tasks based on the formal modelling done on the investigation tasks. It is important to decide if enough knowledge has been gained from the formal modelling to fully clarify the investigation tasks so these can be turned into implementation tasks for future sprints.

Sprint retrospective: Following the sprint review, the team will have a sprint retrospective meeting discussing what went well and what could be done better process-wise in each of the parts of the sprint. It is important to assess both general interpersonal issues but also issues tied to either the formal or software tasks individually. Since knowledge transfer is a key aspect of this project setup, it is important to discuss the degree to which this is working as well.

5.4.3 Benefits of Combining Formal and Agile Methods

One of the key skills for software engineers is abstraction [110] — the ability to specify in detail the system elements needed in order to verify a certain property, while abstracting away from all unnecessary details. This is a complex challenge often mastered by engineers with formal specification experience. Conventional software developers can have a tendency to focus on low-level implementation details of the system and not having the more abstract overview of the entire system. Working alongside formalists (who utilise more abstract thinking) will in time improve the abstraction skills of the programmers which could potentially lead to better system quality.

One of the main benefits of adding the use of formal methods is to give valuable input to the implementation phase. Test cases used to evaluate the formal specification can be reused to test the final implementation. In the agile development method TDD developers are required to write automated unit tests defining the code requirements before the code is implemented. In VDM it is possible to define implicit functions by means of a signature and pre- and post-conditions. By using VDM and TDD as the daily development methods, the formal modelling tasks can generate unit tests which can be reused in the conventional implementation tasks, and hence saving time. This combination of VDM and TDD could be ideal for the method described in this chapter, but more work is required before any conclusions are drawn.

5.5 Summary

The improved analytic power of formal methods tools and greater understanding of the role of rigorous modelling in development processes are gradually improving software practice. However, the claim that formal methods can be part of agile processes should not be made lightly. In this chapter, the value statements and supporting principles of the agile manifesto are examined and areas in which formal methods and tools are hard-pressed to live up to the claim that they can work with agile techniques are identified.

Formal methods should not be thought of as development *processes*, but are better seen as collections of techniques that can be deployed as appropriate. For the agile developer, it is not possible to get benefits from using a formalism unless the formal notation or technique is focused and easy to learn and apply. Luckily, formal modelling and analysis does not *have* to be burdensome. For example, forms of static analysis and automatic verification can be used to ensure that key properties are preserved from one iteration to the next. For this to be efficient, and to fit into the agile mindset, analysis must have automated tool support. Formalists should stop worrying about development processes if they want to support agility. Instead, they should start adjusting their "only perfect is good enough" mindset, and try a more lightweight approach to formal modelling (already discussed in Section 3.2) if their goal is to become more agile.

A concrete approach to using formal methods in the popular agile project management method Scrum has been presented. A single team, consisting of formalists as well as conventional software engineers, creates a formal model in addition to the software implementation in each sprint. By having engineers trained in the use of formal methods working alongside conventional software engineers in a single team, we hope to remove some of the mysticism that surrounds formal methods. During the daily Scrum meetings, knowledge is transfered implicitly, ensuring that the use of formal development methods is better understood by the software engineers. In time, the software engineers of the team can help out on the investigation tasks getting an even better understanding of formal validation techniques.

We hope that the method presented here will provide valuable input to companies looking into adding the use of formal modelling techniques to their agile software development processes. Introducing the use of formal methods in an industrially widely used process like Scrum makes formal methods more accessible and hence increase the industrial penetration.

6

Mono-Disciplinary Modelling Case Study

This chapter describes a case study that applies a combination of the mono-disciplinary modelling approaches described in Chapters 4 and 5. The experiences gained are summarised and compared to the general feedback from industrial use of formal methods described in Chapter 3. Chapter 10 expands the case study presented here by adding a high fidelity model of the physical dynamics.

6.1 Introduction

In a recent project at Terma A/S several components of a self-defense system for fighter aircraft needed to be updated to meet new customer needs. The communication between two subsystems required an update in order to support more complex message types. In order to avoid re-certification of the military-certified handshake protocol used, it was proposed to add information to the existing message structure to enhance the capabilities of the protocol. To gain a higher degree of confidence in the proposed design it was decided to make use of formal analysis techniques to ensure that the communication upgrades did not compromise the core functionality of either of the two subsystems. The guidelines and modelling approaches described in prior chapters were used.

The design of the case study using formal methods is presented in Section 6.2 describing the self-defense system that was analysed in the case study which was called the ECAP case study (short for *Electronic Combat Adaptive Processor*). The purpose of this case study is also defined here. Section 6.3 shows examples of the formal specification created as well as the tests used to evaluate the model. The results of these tests are described in Section 6.4, and lessons learned during the ECAP case study are discussed in Section 6.5. Finally, a summary of the work is given in Section 6.6.

6.2 Case Study Design

An important first step of any case study is to design it thoroughly. The approach described by Runeson et al. [171] was followed. A clear description of the ECAP case study itself is made and a clear purpose of the case study is defined.

6.2.1 Case Study Description

This section gives a functional overview of the self-defense system and the protocol which is the main focus of this study. There are details of the system which cannot be presented here due to military classification restrictions, but sufficient information is given to introduce the reader to this complex system.

When fighter pilots are flying missions in hostile territory there is a risk of encountering enemy anti-aircraft systems. To respond to these threats, the pilot can deploy different types of countermeasures, as described in Section 2.6. Since anti-aircraft systems are becoming increasingly sophisticated, and on-board self-defense systems are also becoming more sophisticated, the fighter pilot is in need of assistance in choosing the optimal countermeasure strategy.

A system called (ECAP) has been developed by Terma A/S to assist the pilot in choosing the optimal response to incoming threats. The system is a programmable unit that provides threat-adaptive processing, threat evaluation and countermeasure strategy to counter incoming threats. From a multitude of sensor inputs (aircraft position, orientation, speed and altitude, and threat type and incoming angle to name a few) the system chooses an effective combination of countermeasures against the incoming threat. The sensors attached to ECAP can detect different types of threats, and will report data of incoming threats to ECAP. The threat response calculated by ECAP, which can consist of one or more countermeasure components, is sent to a *Digital Sequencer Switch* (DSS) subsystem which administers the deployment of the correct types of dispense payloads with the correct timing.

The subsystem of interest for this case study is a special *Advanced Sensor* (AS). This sensor not only detects incoming threats, but also calculates the optimal countermeasures against incoming threats much like the ECAP functionality. AS is running in parallel with the rest of the system, and relies on ECAP to accept and execute generated threat responses. Hence, ECAP –acting as the master– needs to check the proposed response from the AS slave for conflicts with the current system state, and accept or reject the response based on this information. A robust handshake protocol has been specified, defining the communication between ECAP and AS. A threat re-

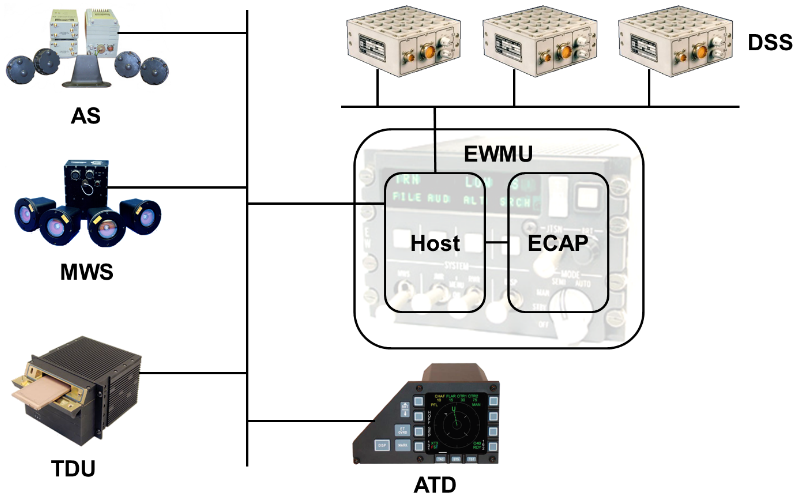


Figure 6.1 Self-defense system overview: Advanced Sensor (AS), Missile Warner System (MWS), Tactical Data Unit (TDU), Advanced Threat Display (ATD), Digital Sequencer Switch (DSS), Electronic Warfare Management Unit (EWMU) and Electronic Combat Adaptive Processor (ECAP). The Host connects all the components of the system.

sponse consists of several components, that are either dispensing routines of countermeasure decoys, a command to a subsystem or audio feedback to the pilot. In the previous version of the system, ECAP treated all components separately which resulted in the need for several pilot-consent actions in order to execute a response. Not only did this put unnecessary strain on the pilot, but it also resulted in delays between the different countermeasure components which could decrease the effectiveness of the compound threat response.

The main focus of this case study was to evaluate an update to the way ECAP interprets messages from the AS system. The protocol itself has undergone military certification, hence no changes to the protocol were possible as this would involve re-certification of the protocol which is both a costly and time-consuming task. In addition to the individual components of the threat response, it was proposed that AS should also generate a compound threat response message which is the concatenation of the sequence of components. What distinguishes a component from the compound threat response is the position in the complete AS message — only the last sub-message is the compound threat response whereas all preceding sub-messages are components.

An example is shown in Figure 6.2 where the first component is a dispense routine, the second component is another dispensing routine, the third component is a jammer program and the fourth component is a command to a subsystem (audio feedback to the pilot for example). The new part of the AS message is the compound message which is the concatenation of the four prior components.

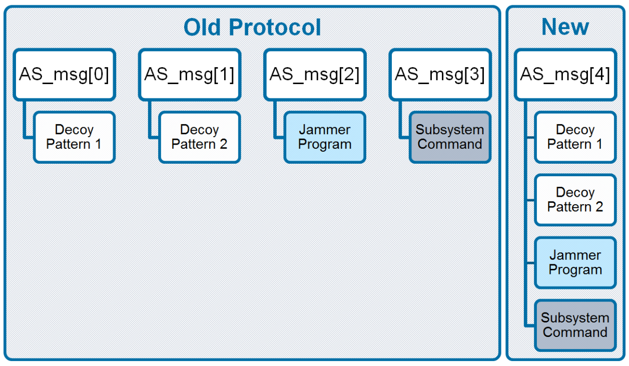


Figure 6.2 Countermeasure components and compound threat response from AS.

ECAP will still test for system conflicts for each of the message components, but will only ever execute a compound threat response. This ensures that only a single pilot-consent is ever needed, which in turn also solves the issue of unwanted delays between countermeasure programs. At any time before the accept of the final complete threat response, ECAP can cancel the requests from AS if another threat surpasses the AS request, or if a higher priority threat is discovered by another subsystem. The use of components ensures that AS knows exactly which part of the compound message is rejected by ECAP, and it will then try to generate another effective compound threat response not using that particular component.

The functionality of –and communication between– ECAP and AS is complex because of both *a)* potential conflicts between system state and proposed countermeasures from AS; and *b)* different priority levels of incoming threats reported by different subsystems. In order to gain confidence in the proposed update using the compound messages in addition to the individual countermeasure components, many corner cases needed to be analysed in order to ensure that no details were ignored. Traditionally, this analysis would have been done by hand — a time-consuming and error-prone process. As an alternative, an executable model of the updated ECAP system with the AS subsystem was developed and analysed.

The extension of ECAP described above was only a small part of a larger update to the self-defense system used by the customer, including modification to other parts of the on-board self-defense system. True to the intent to use lightweight formal methods, only the extension to ECAP was included in the case study described in this chapter. Traditional software development methods (mainly Scrum, based on a thoroughly negotiated backlog) were used to manage the upgrades of the self-defense system. The project had a short timespan with only four months of development time with a workforce of roughly 30 engineers. The case study presented in this chapter was conducted by the author in parallel with the main product development. This setup made it possible to try out a limited version of the process described in Chapter 5 where agile software development was combined with the use of VDM.

Parts of the stepwise approach to VDM model development described in Chapter 4 were used to define the tasks of the individual sprints. The system boundaries were defined and a sequential VDM++ model was created and analysed. By applying an additional phase of the stepwise process, a concurrent VDM++ model was also created to test if the added fidelity made the model more competent. This was not the case, though, so the description of the concurrent model is not included in this chapter. Hence, only the application of Guideline mono_3 through mono_7 for the modelling and Guideline mono_15 and mono_16 for tests of the sequential VDM++ model are described in this chapter.

Figure 6.3 gives an overview of the backlog and duration of the three sprints. The author was not fully integrated into the Scrum teams of the software developers, so their tasks are not included in the figure.

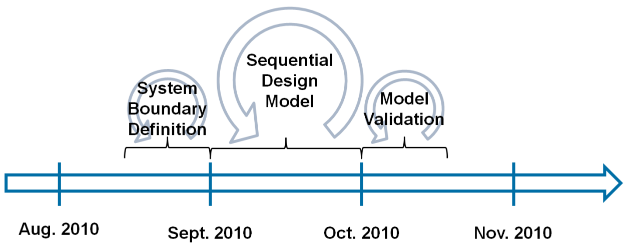


Figure 6.3 Overview of the stepwise model construction.

6.2.2 Purpose of the Case Study

The purpose of the case study, was to create an executable model where various combinations of the ECAP system state and the proposed threat response from AS could be analysed.

In order to enable rapid feedback to the user it was decided that the output created by the model should be in a format that the customer was already familiar with. This would reduce the amount of manual translation needed for the customer to understand the result of the analysed scenario and would also reduce the possibility of introducing errors in the manual translation.

6.3 Case Description: VDM Model of ECAP

Several extracts of the model are shown in order to highlight some of the benefits and challenges in using formal methods.

6.3.1 Model Structure

The structure of the model follows Guideline mono_4 ensuring a structural separation of the environment and system (`ECAP`). Figure 6.4 shows a UML class diagram of the main parts of the model.

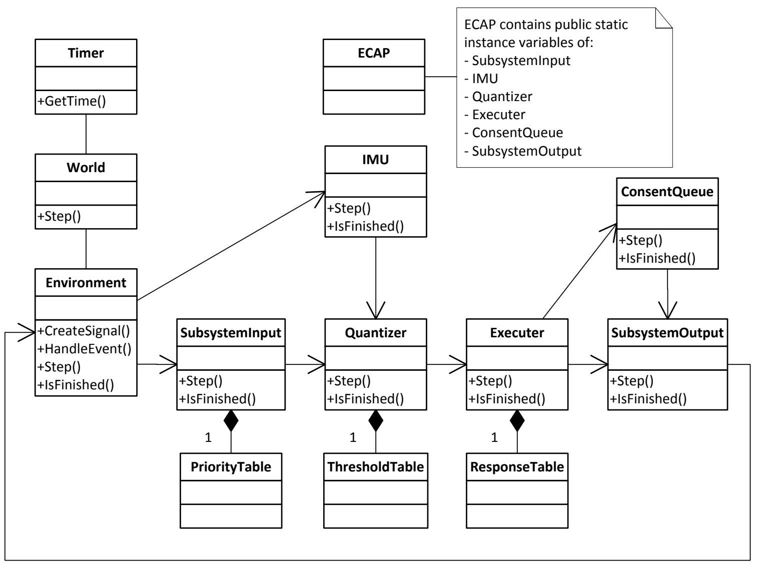


Figure 6.4 UML class diagram of the sequential ECAP VDM++ model.

Following Guideline mono_7, ECAP contains public static instances of the rest of the engineering system, easing the access of these system elements. The Environment class contains CreateSignal and HandleEvent operations for generating input to and receiving responses from the model respectively (Guideline mono_5). The CreateSignal operation can be seen below:

```
class Environment

types
  public inline  = MissileType * Angle * Time;

instance variables
  inlines : seq of inline := [];

operations

private CreateSignal: () ==> ()
CreateSignal () ==
  if len inlines > 0
  then (dcl curtime : Time := World`timerRef.GetTime(),
            done : bool := false;
        while not done do
          def mk_ (type, angle, t) = hd inlines in
            if t <= curtime
            then (ECAP`ssIn.AddInput(type, angle);
                  inlines := tl inlines;
                  done := len inlines = 0);
                 );
```

The CreateSignal operation creates input consisting of type of missile, incoming angle and time from the sequence of inputs using the head (**hd**) sequence operator. When the simulated current time surpasses the time of the input, the AddInput operation of the SubsystemInput class is called, and the sequence of input is updated using the tail (**tl**) sequence operator. Once all input has been processed, the operation terminates.

All active classes were given Step operations for performing the periodic functionality of the class and IsFinished operations to indicate that the process is finished (Guideline mono_6).

6.3.2 Abstract Data Types

To define the boundaries of the model, all the data types to be used by the system were specified. This is a great way to gain insight into the system and helps defining the interfaces between individual components. All necessary types were defined in a class called ECAP_Types (not included in Figure 6.4) that all other classes in the model inherit from — this way all parts of the system share the same types. Below is a simple example of one of the abstract types in the model.

```
class ECAP_Types

types
  public static Coordinate ::
    x : nat
    y : nat;

  public static Area ::
    LowLeft  : Coordinate
    UpRight : Coordinate
  inv a == a.LowLeft.x <= a.UpRight.x and a.LowLeft.y <= a.UpRight.y;

end ECAP_Types
```

When the aircraft is inside *enemy territory* the system has to operate in a specific mode. A rectangular Area defines an enemy territory as two coordinates defining the lower-left and upper-right corners. This is a simplification of the real system where an enemy territory can be defined as any polygon. The benefits of this abstraction is clear in the IMU (Inertial Measurement Unit) which uses these types when checking if the aircraft is within enemy territory: if the aircraft x and y coordinates are within the upper and lower bounds of the Area, the aircraft is inside hostile territory.

```
class IMU is subclass of ECAP_Types

values
  HOSTILE_ZONE : Area = mk_Area(mk_Coordinate(10,30),
                               mk_Coordinate(20,50));
functions

public CheckPosition : Coordinate -> GeoZone
CheckPosition (pos) ==
  if (pos.x <= HOSTILE_ZONE.UpperRight.x and
      pos.x >= HOSTILE_ZONE.LowerLeft.x and
```

```
    pos.y <= HOSTILE_ZONE.UpperRight.y and
    pos.y >= HOSTILE_ZONE.LowerLeft.y)
  then <Hostile>
  else <Friendly>;

end IMU
```

6.3.3 Model Contracts

Design by contract languages like JML [129] and Spec# [23] enable formal interface definitions through: pre- and post conditions; invariants; and errors and exceptions. This interface definition is added as basic annotations of the individual methods and modules. Similarly, VDM can define formal interfaces using pre- and post conditions and invariants. A simple example of an invariant is shown below:

```
types
  Angle : nat
  inv a == a < 360;
```

This invariant ensures that an angle only can be specified as a natural number between 0 and 359. By defining such an invariant the expected interval of an angle is clearly documented.

In a similar fashion pre- and post-conditions can be used to document the accepted entry- and exit-conditions of an operation (Guideline mono_3). It is possible to define functions and operations implicitly by only defining the signature of the operation along with pre- and post-conditions but without specifying the body of the operation. This is a convenient way of documenting the expected use of the operation without having to describe any algorithmic details otherwise needed by the interpreter in order to execute an explicit operation. Below is an example of the use of pre- and post-conditions in the model. The operation `AddThreat` is explicitly defined in the model but the body is left out in the model extract below.

```
public AddThreat : Threat ==> ()
AddThreat (threat) ==
  ...
  pre threat.ID not in set dom Threat_Map
  post threat.ID in set dom Threat_Map;
```

The `Threat_Map` is a mapping from unique thread IDs to the actual threat information. When a new threat is added to the system the pre-condition states that the ID of this new thread must not be known by the system (to avoid registering the same threat several times). The post-condition states that once the body of the operation has been evaluated the ID of the threat should now be known by the system. This is a convenient way of defining the intended behaviour of the operation without having to go into implementation details.

6.3.4 Abstract Operators

VDM has many abstract operators which can be used to describe *what* should happen without having to describe *how* this should be done — similar to the use of implicit specifications. Set and sequence operators provide a compact way of describing functional behaviour without having to consider constraints imposed by the target implementation language. An example is shown below:

```
private PrioritiseQueue : () ==> ()
PrioritiseQueue () ==
  let PrioOneQueue : seq of ThreatResponse =
      [Queue(i) | i in set inds Queue & Queue(i).Prio = 1],
      PrioTwoQueue : seq of ThreatResponse =
      [Queue(j) | j in set inds Queue & Queue(j).Prio = 2],
      ...
  in
    Queue := PrioOneQueue ^ PrioTwoQueue ^
             PrioThreeQueue ^ PrioFourQueue;
```

The *sequence comprehension* operator (described in Appendix B) is used to sort a sequence of threat responses based on the priority of each individual threat. A sequence is created for each of the four levels of priority defined in the system. In the end the four sequences are concatenated in ascending order. This clearly defines the intended behaviour of the operation without having to use target implementation language specific constructs to write a high performance sorting algorithm.

6.3.5 Testing the Protocol

Three different techniques were used to analyse and validate the model created in the ECAP case study: the *Proof Obligation Generator* [29]; the unit testing framework VDMUnit [72]; and the *Combinatorial Testing* tool [123] all built into the Overture tool. These validation tools have already been introduced in Section 3.3.3.

During the system boundary definition, it was decided to model the instance variable `DecoyLeft` as a natural number, since the system cannot have a negative number of flares left. The operation `DecoyUsed` is called when decoys are dispensed ensuring the number of decoys available on board the aircraft is updated. If more decoys are used than is available, a run-time exception would be generated. Below is an example of the output generated by the proof obligation generator:

```
((Dispenser.DecoyLeft) - n) >= 0
```

To manually discharge this proof obligation, a pre-condition was added to the operation. This also documents the intended use of the operation, reminding the model designer to check if the required number of decoys for a given threat response are in fact available.

```
public DecoyUsed: nat ==> ()
DecoyUsed(n) ==
   Dispenser.DecoyLeft := Dispenser.DecoyLeft - n;
pre Dispenser.DecoyLeft - n >= 0;
```

Following Guideline mono_15, 21 different unit test scenarios were constructed using VDMUnit, testing different combinations of ECAP system state and requests from the AS subsystem. Most of these scenarios were defined by the customer who wanted to analyse different corner-cases. Most of these scenarios are long and complex, so only a simple example is included below:

```
class SimpleTest is subclass of TestCase

operations

protected Test: () ==> ()
Test() ==
 (--Req 105: Use Cat1, Cat3; IAT=(4,0,4,0,0); RT=3
  ECAP`AP.Generate_TestInput({{mk_ (1,<Req>),
                               mk_ (2,<Req>),
                               mk_ (105,<Req>)}});
  World`Step();  -- 1 timeunit delay from AS
  World`Step();
  World`Step();  -- Response executed
  AssertTrue(ECAP`IAT.GetIatTable() =
             mk_ECAP_Types`IntAvoid(4,0,4,0,0));
 );
```

In the example above a compound threat message from AS is generated as input for ECAP. The compound threat message has ID 105 and is composed of two components with ID 1 and 2. Since no conflicts exist, the compound threat message is executed by ECAP and the *Interference Avoidance Time* (IAT) specified in the response is set in the system state. Should the assertion be violated the test would fail and the model designer would be pointed to the failing test case.

To reuse these unit tests for future concurrent and real-time models (Guideline mono_16), only minor modifications are needed. Instead of the `World` class having the thread of control repeatedly invoking the `Step` operation of active classes, these processes will run independently processing any available input. The `Environment` class only needs to supply an input to the model, and check if the expected output is generated after a certain time has elapsed.

The model created in the ECAP case study made use of hierarchical mappings to define huge multi-dimension look-up tables used to acquire the response to a given threat situation. The sets used inside the hierarchical mappings were limited to a maximum of four elements, to limit the size of the state-space of the model. The real ECAP system can be configured to manage more than 64.000 different responses. The model can be extended to support an equally large state-space, but it was not deemed necessary for the ECAP case study presented here, where only the conceptual design of the system needed to be tested.

```
class ResponseTable is subclass of ECAP_Types

instance variables

private ResponseTableDB :
  map AzimuthZone to
    (map ChaffLeft to
      (map GeoZone to
        (map Speed to
          (map Altitude to ThreatResponse))));
```

Combinatorial testing (see Section 3.3.3) was used to ensure that the multi-dimension look-up tables contained an output for every possible combination of input accepted by the system. As a concrete example, the trace definition below was used to ensure that the ECAP `Executer` could generate a response to all possible combinations of input the hierarchical mappings were limited to:

```
class CT

instance variables
  table : ResponseTable := new ResponseTable();

traces
  CT_Test_ResponseTable:
    let azi in set {<Front>,<Left>,<Right>,<Rear>}
    in
     let chaff in set {<Normal>,<Low>} in
      let geo in set {<Hostile>,<Friendly>} in
       let speed in set {<Fast>,<Normal>,<Slow>} in
        let alt in set {<High>,<Normal>,<Low>} in
         table.Lookup(azi,chaff,geo,speed,alt)
```

This simple regular expression expanded to 144 individual test cases which were automatically executed by the tool in less than 5 seconds. The tool also enables post-execution inspection of the individual test cases as well as their results. Using this testing approach on other parts of the model ensured that there were no combinations of input that the model could not handle.

An alternative to the combinatorial testing would be to translate the model to another formal specification language and make use of symbolic model checking techniques [20]. This ensures that the entire state-space would be searched in order to find counterexamples to a given claim. The level of detail in our model would cause the state-space to explode if we tried symbolic model checking. In order to utilize model checking we would need to abstract away from details in the model which involves a risk of over-simplifying critical details.

6.4 Case Study Discussion

In total, the model of ECAP and the AS subsystem consists of more than 1800 lines of VDM++ specification. In addition, more than 1500 lines of test scenarios were created to run the many scenarios needed to exercise the new use of the protocol. The Overture tool has the ability to generate test coverage of a model, which gives an indication of parts of the model which are exercised less than other parts. The AS subsystem has a test coverage of 100%, meaning that every line of specification has been exercised by the scenarios. This only ensures that all lines have been executed at least once, but not that all combinations of input have been covered. On average, the complete model of ECAP, AP and all other subsystems has a test coverage of 94.1%. The unit tests were used to find errors in the model where the wrong level of abstraction had been applied. These errors were fixed before more large scale testing was applied.

A total of 253 proof obligations were generated by the Overture proof obligation generator — the majority of these were caused by the extensive use of mapping applications in the model where the model needs to ensure that the domain value is in the domain of the mapping. The proof obligations generated helped locate missing invariants on data types and pre-conditions for functions and operations.

The ECAP and AS model made use of extensive logging, so at any point in time the system state was available for post-execution analysis. The logfiles from the many scenarios were used directly in the communication with the customer to give a precise description of how the systems should react in the different situations. This was a great aid in agreeing on how ECAP should interpret the countermeasure components and compound threat response.

For a system the size and complexity of the one presented here, it is difficult and error-prone to analyse the many combinations of system state and input by hand. The test suite composed for this project does not ensure complete coverage of the state-space of the system model, but provided a simple framework enabling extension of other scenarios to analyse some newly discovered corner-cases. This ensured a short duration of the iterative cycles internally in the company when new scenarios had to be tested. The model test suite was also used as input for the integration test of the realisation of the final system.

The models of ECAP, AS and the protocol were developed by a single person over a period of just two months including knowledge gathering of the systems involved. This was only possible due to the fact that many of the details of the real system were omitted in the model, and only the main functionality of the systems was included in the model. Below are a few examples of abstractions:

- In the real self-defense system the different subsystems are connected by a military standard communication bus which could have been modelled in detail to test collision of messages. This was omitted as it was not relevant to the model's purpose of testing the enhancement of the communication protocol, and hence beyond the scope of this case study.
- Several subsystem commands to enable and disable various subsystems were omitted.

This is one of the main advantages of using system modelling in the early phases of system development: key properties of the system are described in detail while any unnecessary details are abstracted away.

6.5 Lessons Learned

The industrial cases from the literature survey described in Section 3.2 had several common conclusions. Each of these key points are compared with experiences gained from the ECAP case study described in this chapter.

6.5.1 Lightweight Formal Methods

Many of the industrial cases reported a preference for a more lightweight approach to the use of formal methods. By only modelling the ECAP and AS subsystems rather than the entire self-defense system it was possible to finish the model and analyse the numerous scenarios defined by the customer. A more heavyweight formal approach would be more suitable for verifying the absence of errors in the model, however this requires much more work and would definitely have been outside the scope of this four month long development project at Terma.

In projects like the seL4 microkernel [15], full formal verification was done resulting in a high confidence in the resulting system. In the case presented here a higher confidence was gained in the *concept* of the enhancement of the communication protocol: the model clearly showed that the concept was valid but said nothing of the actual implementation. Much can be gained from this early insight into the feasibility of a design or concept. In the case presented here it was all that was needed to assure the customer that a correct solution to the existing protocols limitations had been designed. It is important, though, to understand the limitations of a more lightweight approach in order to avoid expectations of full proof of all properties of the system.

One of the key benefits of using formal methods is the way they force questions to be asked of the requirements. A lot of accuracy and detail is needed in a formal specification to precisely model the system. This forces the formalist to ask accurate and detailed questions to the domain expert. These questions are often completely overlooked in non-formal development techniques which can lead to incompleteness in the requirements specification.

6.5.2 Prior Knowledge

The author had prior experience with both VDM and the Overture tool, so no training was necessary. Domain specific information, readable by both the systems engineers and customer, was produced when exercising the model with the different scenarios. This removed the need for any formal methods knowledge by others than the author, which eased the use of the model. After the model had been interpreted the results could be sent unaltered to the customer who could easily understand them. It is our clear experience that this is a major step forward in ensuring a wider use and acceptance of formal methods.

As a test, the author led a one-day workshop, introducing some of the other engineers to VDM. At the end of this workshop, engineers (especially those with a strong coding background) had no problems reading and understanding VDM models, and they even succeeded in producing their own simple models. Further assistance would be needed if these engineers should use VDM for real tasks, but even such a short introduction was enough for most engineers to actually start creating their own models.

The model was developed over a short period of time to validate the conceptual changes to the communication protocol. It was not the plan to maintain the model beyond this short window of opportunity. Had this been the case, it would have been imperative to include some of the software developeres and/or domain experts in the creation of the model to ensure knowledge transfer and through that increase the maintainability of the model.

6.5.3 Tools

The author found the Overture tool to be stable and easy to use compared to many other open-source formal methods tools. The combinatorial testing helps greatly in producing a large test suite as it is a convenient way of testing all permutations of input of the model, and greatly reduced the time needed on tests. This type of automation is imperative for formal methods to see wider use in industry. The Overture tool does not provide any assistance with regards to proof, except the possibility of translating VDM proof obligations automatically [188, 189] to the theorem prover HOL [83] where these can be formally verified. Since a more lightweight approach was chosen, this feature was not used.

6.5.4 Existing Development Process

As mentioned, the author was the sole formalist in the project, and was as such not fully integrated into the development team working on the actual system. In order to fully utilise the formal model produced, the formalist and conventional developers should be working in much closer collaboration, and the formal method should be fully incorporated into the development process used.

6.5.5 Formal Requirements Specification

Traditionally it is advised to use formal methods as early in the development process as possible because the sooner errors are found the cheaper they are to correct [36, 37]. Often the requirements specification is seen as the most critical document [85] so this is one place where formal methods have been used most in industry. Unfortunately, having a formal requirements specification does not ensure that it actually describes what the customer wants and needs. For this, the ability to interpret an executable specification can provide much more valuable input to the process and help in ensuring that the customer and developer have the same understanding of the system. This was the way the formal model was used in the ECAP case study described here, and proved to be a valuable addition to existing communication methods.

6.6 Summary

The outcome of the ECAP case study was positive — the customer was especially satisfied with the early and rapid feedback provided by the formal model created. The software developers gained some insight into the benefits of using formal specifications, but since the author was not fully integrated into the daily Scrum meetings the knowledge transfer was limited.

This was the first time the systems engineers had any experience with formal methods and their first time using executable models to specify functional requirements in general. One of the systems engineers on the project commented:

> *"The possibility to run numerous scenarios to analyse different combinations of ECAP system state and AS input was invaluable, and the rapid feedback from the model designer was very useful due to the time constraints of the project. We are extremely happy with the results obtained from this case study, which helped us in*

reaching an agreement with the customer within a limited period of time."

The use of a formal model and lightweight formal analysis principles, such as the scenario-based tests and manual inspection of the generated proof obligations, proved to be valuable for the ECAP case study. A lot of insight was gained in the functionality of the system in general and specifically of the new interpretation of the messages passed between ECAP and the AS subsystem which was invaluable in reaching an agreement with the customer.

The ECAP case study was later extended to enable analysis of more complete electronic warfare scenarios, including a continuous-time model of the rigid body dynamics of the flares, and flight dynamics and target seeking of a missile — this is described in Chapter 10. In addition, the industrial partner Terma is planning a follow-up project in the surveillance domain. This clearly shows the industrial impact of the ECAP case study presented in this chapter.

Part III

Multi-Disciplinary Modelling

7

Multi-Disciplinary Modelling

This chapter introduces a multi-disciplinary modelling approach where the discrete controller and the continuous dynamics of the environment are described using two different formalisms. The DESTECS toolchain is introduced, which is capable of collaborative modelling and co-simulating of multi-disciplinary systems. This chapter provides the formal foundation for the rest of this thesis, where the DESTECS tool is used.

7.1 Introduction to Multi-Disciplinary Modelling

In classical physics, aspects like movement, acceleration and forces are described using differential equations. Naturally, models of the environment in which an embedded system is operating are best described using differential equations as well. Continuous-time (CT) models excel at describing mechanical systems and rigid body dynamics. These types of models, however, are not suited for describing discrete-event (DE) controllers. An embedded controller is typically constructed using a layered architecture, separating the concerns of the application layer from the safety layer — CT models rarely support this type of structure. To model both the embedded controller and the dynamics of the environment at a high level of fidelity, a multi-disciplinary modelling approach is needed.

A strong engineering methodology for embedded systems will be *collaborative*. A collaborative modelling approach [1] enables the model designer to describe not only the controller at a high level of fidelity, but also the physical dynamics of the environment. In addition, co-modelling improves the chances of closing the design loop early, encouraging dialogue between disciplines.

Such an approach will provide notations that expose the impact of design choices early, allow modelling and analysis of dynamic aspects and support

[1] Throughout this thesis, we term this approach "collaborative modelling" or "co-modelling".

systematic analysis of faulty as well as normal behaviour. The hypothesis supporting the multi-disciplinary modelling approach is that lightweight formal and domain-specific models that capture system-level behaviour can supply many of these characteristics, provided they can be combined in a suitable way and be evaluated rapidly.

Verhoef has demonstrated that significant gains are feasible by combining VDM and bond graphs [107], using co-simulation as the means of model assessment [184]. The work reported in this part of the thesis mainly made use of the tools developed as part of the EU FP7 Project DESTECS [43]. An introduction to DESTECS is given in Section 7.2. Various other alternative tools and technologies capable of creating and (co-)simulating multi-disciplinary models are introduced in Section 7.3.

7.2 Multi-Disciplinary Modelling in DESTECS

DESTECS (Design Support and Tooling for Embedded Control Software) aim to develop methods and tools that combine CT system models with DE controller models through co-simulation. The focus of the project is on multi-disciplinary modelling, fault modelling and modelling fault tolerance mechanisms.

In the DESTECS approach to collaborative development, a *model* is a more or less abstract representation of a system or component of interest. The primarily concern of the DESTECS project is analysis by execution, so the main interest is formal models that, while they are abstract, are also directly executable. A test run of a model is called a *simulation*. A *design parameter* is a property of a model that affects its behaviour, but which remains constant during a given simulation. A simulation is normally under the control of a *script* that determines the initial values of modelled state variables and the order in which subsequent events occur. A script may force the selection of alternative implementations of submodels and may supply external inputs (e.g. a change of set point) where required. A *test result* is the outcome of a simulation over a model.

A *co-model* (Figure 7.1) is a model composed of:

- Two component models, normally one describing a computing subsystem and one describing the plant or environment with which it interacts. The former model is typically expressed in a DE formalism and the latter using a CT formalism.

- A *contract*, which identifies shared design parameters, shared variables, and common events used to effect communication between the subsystems represented by the models.

A co-model is itself a model and may be simulated under the control of a script. The simulation of a co-model is termed *co-simulation*. A co-model offers an interface that can be used to set design parameters and to run scripts to set initial values, trigger faulty behaviour, provide external inputs and observe selected values as the simulation progresses.

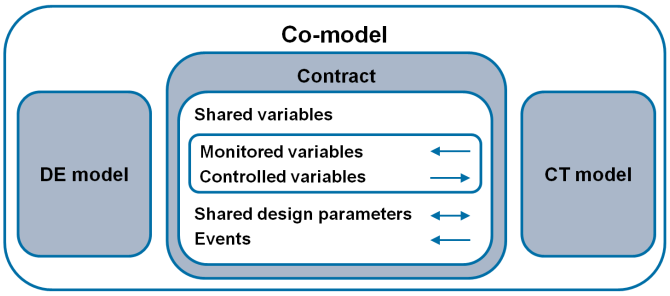


Figure 7.1 Components of a DESTECS co-model.

In a co-simulation, a *shared variable* is a variable that appears in and can be accessed from both component models. *Monitored variables* are monitored by the embedded controller, while *controlled variables* are environment variables controlled by the embedded controller — this is similar to the I's and O's of the four-variable model of Parnas and Madey [156]. Shared variables may change value as the co-simulation progresses. The variables can be of boolean type or real valued, and several can be passed using multi-dimensional array structures. Predicates over the variables in the component models may be stated. The changing of the logical value of a predicate at a certain time is termed an *event*. Events are referred to by name and can be propagated from one component model to another within a co-model during co-simulation. The semantics of a co-simulation are defined in terms of the evolution of these shared variable changes and event occurrences while co-model time is passing.

In a co-simulation, the CT and DE models execute as interleaved threads of control in their respective simulators under the supervision of a *co-simulation engine*. The DE simulator calculates the smallest time Δt it has to run before performing the next action — this Δt is communicated to the co-simulation engine, without the DE simulator actually executing. This time step is communicated to the CT simulator which then runs the *Ordinary Differential Equation* (ODE) solver forward by up to Δt. If the CT simulator observes an event, for example when a continuously varying value passes a threshold, this is communicated back to the DE simulator by the co-simulation engine. If this event occurred prior to Δt, then the DE simulator does not complete the full time step, but it runs forward to this shorter time step and then re-evaluates its simulator state. If no event happens, the DE simulator runs forward the full Δt. Note that it is not possible (in general) to roll the DE simulation back, owing to the expense of saving the full state history, whereas the CT solver can work to specified times analytically. Verhoef et al. [186] provide an integrated operational semantics for the co-simulation of DE models with CT models. Co-simulation soundness is ensured by enforcing strict monotonically increasing model time and a transaction mechanism that manages time triggered modification of shared variables. The co-simulation semantics have been updated as one of the deliverables of the DESTECS project [51].

The DESTECS toolchain uses CT models expressed as differential equations in bond graphs [107] and DE models expressed using the VDM notation explained in Section 3.3. The simulation engines supporting the two notations are, respectively, 20-sim [42, 1] and Overture [116]. An introduction to DE modelling in VDM has already been given in Section 3.3, and a short introduction to modelling in 20-sim is given in the following subsection.

7.2.1 Continuous-Time Modelling in 20-sim

20-sim [42], formerly CAMAS [41], is a tool for modelling and simulation of dynamic systems including electronics, mechanical and hydraulic systems. All models are based on bond graphs which is a non-causal technology, where the underlying equations are specified as equalities. Hence, variables do not initially need to be specified as inputs or outputs — the energy flow is continuously being evaluated at simulation time. In addition, the interface between bond graph elements is port-based where each port has two variables that are computed in opposite directions, for example voltage and current in the electrical domain.

20-sim also supports graphical representation of the mathematical relations between signals in the form of block diagrams and iconic diagrams (building blocks of physical systems like masses and springs) as more user friendly notations. Using a combination of notations is also possible, since bond graphs provide the common basis. It is possible to create submodels of multiple components or even multiple submodels allowing for a hierarchical model structure. 20-sim includes a library of iconic components from several different domains such as hydraulic, electric, mechanical and thermal as well as pre-defined blocks for signal sources, controllers and filters. In addition to blocks from pre-defined libraries, users may also define their own blocks, and add them to the library for later reuse. These blocks may contain equations, bond graphs, or further graphical models. A block may have more than one "implementation" allowing alternative behaviours to be modelled. The DESTECS tool allows the user to select a specific implementation before co-simulation.

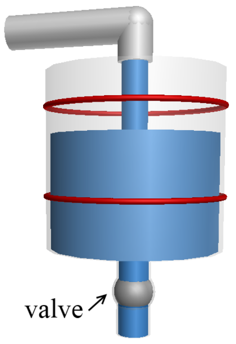
20-sim has several tools for validating a model in both the frequency and time domain. A *Fast Fourier Transformation* (FFT) can be used to calculate the frequency content of a simulated signal. This is useful when comparing the simulated signal with a measured signal from the real system. In the time domain, 20-sim can do *parameter sweeps* where one or more parameters are given a range of values and simulations are automatically done for all permutations of the values. This is valuable when trying to find the optimal value of a given parameter, or combination of parameters, and can greatly reduce time spent on manual testing.

20-sim includes a 2D graph editor enabling the model designer to pick any signal or parameter in the entire model from a hierarchical tree structure, add a name and color and have them plotted either collided or in separate graphs in the same window. 20-sim also has a built-in 3D animator. This enables the model designer to use variable values of the model to scale, rotate or move simple objects like cubes, cylinders or planes. In addition, objects can be imported from other 3D modelling tools in a variety of standard formats. An example of the capabilities of the 3D animator can be seen in Figure 7.2.

7.2.2 Basic Co-Simulation in DESTECS

In this section, a co-simulation is illustrated by means of a simple example based on the level controller of a water tank (Figure 7.2).

The water level of the tank is based on the continuous input flow φ_{in} and output flow φ_{out} described in Equation 7.1. The output flow through the valve, when this is opened or closed, is described by Equation 7.2 in Figure 7.2, where ρ is the density of the water, g is acceleration due to gravity, A is the surface area of the water tank, R is the resistance in the valve and V is the water volume. An iconic diagram model of this system created in

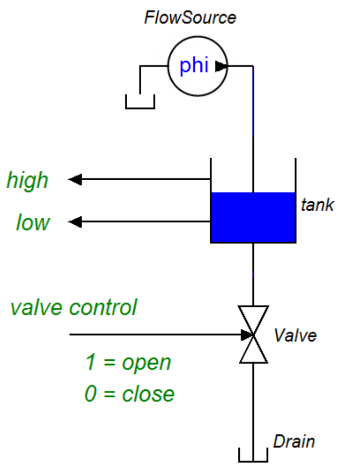


$$\frac{dV}{dt} = \varphi_{in} - \varphi_{out} \tag{7.1}$$

$$\varphi_{out} = \begin{cases} \frac{\rho * g}{A * R} * V & \text{if valve open} \\ 0 & \text{if valve closed} \end{cases} \tag{7.2}$$

Figure 7.2 Water tank level controller case study system overview.

20-sim is shown in Figure 7.3 (a). There are two simple requirements for the DE controller: when the water reaches the “high” level mark the valve must be opened, and when the water reaches the “low” level mark, the valve must be closed. A VDM model of the controller is shown in Figure 7.3 (b).



```
class Controller

instance variables
 private i : Interface

operations
 async public Open: () ==> ()
 Open() == duration(50E3)
   i.SetValve(true);

 async public Close: () ==> ()
 Close() == cycles(1000)
   i.SetValve(false);

sync
 mutex(Open, Close);
 mutex(Open); mutex(Close)

end Controller
```

Figure 7.3 (a) Iconic diagram CT model (left) and (b) event-driven DE controller in VDM (right).

The controller model is expressed in VDM-RT. An instance variable represents the state of the valve and the asynchronous `Open` and `Close` operations set its value. Both operations are specified explicitly in the sense that they are directly executable. In order to illustrate the recording of timing constraints in VDM-RT, the **`duration`** and **`cycles`** statements constrain the time taken by the operations to 50,000 ns in the case of `Open` and 1000 processor cycles in the case of `Close`. The time taken for a `Close` operation is therefore dependent on the defined speed of the computation unit (CPU) on which it is deployed (described in the **`system`** class of the model). The synchronisation constraints state that the two operations are mutually exclusive.

A co-model can be constructed consisting of the CT model and DE model shown above. The co-simulation contract between them identifies the events from the CT model that are coupled to the operations in the DE model and indicates that `valve` is shared between the two models.

```
sdp real maxlevel;
sdp real minlevel;

controlled bool valve;

event high;
event low;
```

The contract indicates which state event triggers which operations. In this case the water level rises above the upper sensor, the `Open` operation shall be triggered and respectively when the water level drops below the lower sensor, the `Close` operation shall be called. Note that `valve` represents the actual state of the valve, not merely the controller's view of it.

7.2.3 Miscellaneous DESTECS Tools

The DESTECS toolchain includes various other tools useful during co-model analysis. These tools are briefly described here.

The *DESTECS Command Language* (DCL) is a scripting language which is used to simulate user input and activate latent fault behaviours during a co-simulation. An example of the use of DCL, can be seen in Section 8.3.2.

To setup a co-simulation run a *Launch Configuration* is created inside the DESTECS tool, defining the entry point of the simulation, simulation time, shared design parameter values, type of ODE solver, etc. If the CT model includes several implementations, the correct one is set in the launch configuration as well.

An interface to the tool Octave [148] has been implemented in DESTECS enabling the user to choose any signal from the CT or DE model to be plotted in a 2D graph. This gives the user further possibilities to monitor and compare parameter values in the model.

The *Automated Co-model Analysis* (ACA) of the DESTECS tool enables the user to setup parameter sweeps on multiple parameters at the same time. A special launch configuration defines the value ranges of the shared design parameters, and the ACA tool will then automatically perform all possible permutations of these sweeps. A *results* file is generated giving a comparative overview of the ACA, and individual co-simulations can be chosen to be co-simulated again.

7.2.4 Limitations of Co-simulation in DESTECS

The ACA tool is an important aid in performing design space exploration, automating some of the tedious tasks involved. The entire state-space of the co-model is not searched though — this is not possible since most continuous-time variables are real-valued types resulting in state-space explosion.

When time is synchronised and variable values are passed between the discrete and continuous-time simulators, there is a communication overhead, that has a negative impact on the co-simulation speed. The model designer must be cautious when designing the two models and the contract between them, as to not synchronise values more often than is needed. For example, the type of CT solver as well as the minimum step-size of the CT solver must be adjusted to allow it to evaluate the differential equations correctly, with minimum overhead. To examine the simulation overhead caused by the DESTECS architecture, Chapter 8 compares DESTECS with the Ptolemy tool that consists of only a single simulator for both CT and DE models.

Even a high fidelity co-model, is still just a model of the real system and environment, and will always contain inaccuracies and abstractions. It is important all significant parameters of the system are included in the co-model to decrease the level of inaccuracy, ensuring a competent model. Physical prototypes can highlight aspects of the system that the co-model must ignore for the sake of simplicity. The prototype can be tested in the real environment in which the embedded system operates. This is not a limitation isolated to co-models, but is an inherent limitation of all types of models.

7.3 Alternative Multi-Disciplinary Modelling Approaches

Various other alternative tool and modelling approaches exist for collaborative modelling and (co-)simulation of multi-disciplinary systems. The most significant of these are described below.

7.3.1 Application Domain Modelling

The objective of domain engineering as introduced by Dines Bjørner [31] is to create a domain description. In principle a domain model does not contain any reference to the machine, and strives to describe the domain *as it is*. A domain description specifies entities, functions, events and behaviours of the domain such as the domain stakeholders think they are. A domain description thus expresses *what there really is*, whereas a requirements model expresses *what there ought to be*.

To develop a proper domain description necessitates a number of development stages: *i)* identification of stakeholders, *ii)* domain knowledge acquisition, *iii)* business process rough-sketching, *iv)* domain analysis, *v)* domain modelling: developing abstractions and verifying properties, *vi)* domain validation and *vii)* domain theory building.

As such, domain engineering is not a multi-disciplinary modelling approach, but some of the principles can be used for defining the system boundaries (i.e. phase one of the VDM-RT process described in Chapter 4, or to better define the contract between the continuous and discrete parts of a co-model).

7.3.2 Hybrid Automaton

A hybrid system is a dynamic system that exhibits both continuous and discrete dynamic behavior [12]. The term *hybrid system* is often used interchangeably with *embedded system* or *cyber-physical system* in the literature. A hybrid system can be described by means of a hybrid automaton consisting of a finite automaton with continuous dynamics associated with each discrete state of the automaton that are typically modeled via differential equations. 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 transition from one discrete state to another is guarded by boolean expressions, typically inequalities on the values of the continuous state. When a discrete transition occurs, then assignments are made to the continuous state that act as initial conditions to the next discrete state.

7.3.3 Problem Frames

The task of developing a specific piece of software for a system can be seen as a problem [99]. The problem consists in developing a machine that satisfies some requirements in a given environment — i.e. a part of the real world. The machine is then the solution to the problem, but in the *problem frames* approach, emphasis is explicitly put on the task of understanding the problem before searching for specific solutions. A problem is understood by describing the environment and the requirements for the behavior that the machine should bring about in terms of resulting interaction between the machine and the physical entities of the environment. In the problem frames approach, *problem diagrams* are used for describing the structural properties of specific software development problems.

Hayes, Jackson and Jones have extended the use of problem frames [86, 104]. They argue that the specifications of a system should be derived formally from a description of required phenomena in the physical world in which the system will operate. Like domain modelling, problem frames can be used to define the system boundaries of a multi-disciplinary system.

7.3.4 MADES Co-Simulation Approach

The project *MADES Co-simulation Approach* (MCA) [22] allows designers to combine logic formulae describing the controller with non-casual CT models created in Modelica [79]. MCA supports simulation of nondeterministic models, where the simulator tries out different alternatives in order to find an execution trace that does not violate either of the two models. Users can specify logical constraints on the execution. An example of this is temporal logical constraints like: *"a variable v must assume a value between 2 and 3 within 5 time units"*. The co-simulation tool then picks a value randomly within the specified interval. It is assumed that the CT environment model is deterministic since the differential equations always evaluate to a unique solution.

In MCA the controller and environment have private variables that are only visible in the individual models. In addition the two models communicate through shared variables. Each variable belongs to a model (shared controller variables and shared environment variables) — similar to the *controlled* and *monitored* shared variables of DESTECS. The shared variables of MCA can be real-valued or boolean signals.

The MCA approach is similar to the DESTECS approach in that it allows designers to combine different, complementary formalisms instead of requiring that the models fit into a single all-encompassing notation. This approach takes advantage of the strengths of each of the individual formalisms.

7.3.5 MathWorks Tools

In industry MATLAB/Simulink [137] created by MathWorks is one of the most widely used tool for creating CT models. MATLAB is a modelling language and interactive environment which lets the user create the models fast compared to 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 customizable set of block libraries which lets the user design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing.

Two tools have been created to increase the co-simulation capabilities of the MathWorks tool suite. These two tools are described briefly below.

Stateflow

Stateflow adds control logic to the MATLAB/Simulink toolbox for modelling reactive systems using state charts and flow diagrams for the controller model. The control logic is then co-simulated with a CT model created in Simulink. The Simulink models use a causal approach where the ODEs describing the continuous behaviour of the environment have a causal relation between input and output variables and are connected using signals. The control logic in Stateflow is represented by finite state machines allowing for hierarchy, parallelism and history within the state charts. Additional tools allows for automated code generation from the Stateflow charts to C, HDL and PLC code.

AMESim

Advanced Modeling Environment for performing Simulations of engineering systems (AMESim) [134] extends the MathWorks toolbox with a non-causal modelling approach where the interface between model elements are ports representing physical entities that operate bi-directionally. This is enabled due to the underlying bond graph theory also used in 20-sim (see Section 7.2.1) which facilitates links between various physical domains.

7.3.6 Ptolemy

Ptolemy II [44, 56] [2] is a modelling and simulation framework for multi-disciplinary systems which is developed at University of California, Berkeley. The framework is an open-source Java project that has been developed since 1996.

Ptolemy has been used in industry, for example in HP's DSP Designer and DSP Synthesizer [159] or MLDesigner [141] which use concepts developed in the Ptolemy project. Mirabilis Design [140] have commercialised Ptolemy in the tool VisualSim. Ptolemy has also seen a wide non-commercial use, most noteworthy are the Kepler project [109] and the Building Controls Virtual Test Bed (BCVTB) [24].

The actor-oriented design principle is used to describe systems. Actors are components that communicate via ports. The semantics of an actor and the communication is given by a *Model of Computation* (MoC). Examples of MoCs in Ptolemy are DE, CT, various data flow variants such as synchronous data flow (SDF), process network (PN) and synchronous reactive (SR).

Heterogeneous composition of different MoCs is enabled via hierarchies where every hierarchy level represents exactly one MoC. A special actor, the director, enforces the MoC on each hierarchy level. Composite actors contain actors and, in the case of opaque composite actors, they contain a director. To the enclosing model, a composite actor behaves like an atomic actor with ports for communication. Transparent composite actors do not contain a director and are mere logical groupings of actors.

Ptolemy includes several types of plotters used to give a 2D graph of model signals over time. To add a signal to a plotter a connection is simply drawn from the signal to the plotter. Ptolemy also has a Java3D interface which enables 3D visualisation of models.

7.3.7 Functional Mock-up Interface

Functional Mock-up Interface (FMI) [35] is an emerging standard (defined by the MODELISAR project [76]) which supports both model exchange and co-simulation of dynamic models. Numerous simulation tools already support

[2] Ptolemy II is from hereon referred to as "Ptolemy"

FMI: AMESim, CosiMate, Dymola, MapleSim, and MATLAB, to mention a few. Ptolemy has a FMI under development, which will enable co-simulation of Ptolemy models and models created in other tools supporting FMI.

7.3.8 Custom-Built Simulators and Game Engines

As an alternative to high fidelity multi-disciplinary modelling, custom-built simulators can be used to analyse the behaviour of embedded systems. An example of this is Gazebo which is a 3D environment simulator which is part of the Player/Stage project [161]. For dynamic physics and rigid body simulation, Gazebo uses the *Open Dynamics Engine* (ODE) [149] physics engine, and for visuals the open-source rendering engine Ogre.

Physics engines like ODE, Bullet [45] or PhysX [158] are often used for computer games or movies, and excel in real-time physics simulations. This is possible through the use of predictive algorithms instead of high fidelity simulation. In recent extensions to 20-sim, an interface to the Bullet physics engine was added to support rigid body collision and contact modelling.

In the work of Baqar [21] MATLAB was used with the virtual reality (VR) toolbox from Mathworks to model and simulate IR signatures of aircrafts and flares. The VR toolbox is a simple renderer environment which lacks high quality visualization.

Most modern game engines includes both high quality rendering environments and a physics engine. Craighead et al. have used the Unity game engine [182] to create a *Search and Rescue Game Environment* (SARGE) [54]. In Unity, a custom rendering engine is integrated with the PhysX physics engine and Mono, which is an open-source implementation of Microsoft's .NET libraries. This, combined with a user-friendly editor, makes the Unity game engine an interesting alternative to high-fidelity modelling of embedded systems.

8

Multi-Disciplinary Modelling Tool Comparison

This chapter presents a comparison between the DESTECS tool (used in Chapter 9 and 10) and the Ptolemy tool — one of the alternatives tools capable of multi-disciplinary modelling presented in Sections 7.3.6. An aircraft fuel system is used as a case study, and both qualitative and quantitative comparison results are discussed.

8.1 Tool Comparison Introduction

Of the available tools capable of multi-disciplinary modelling and (co-)simulating, the DESTECS toolchain was chosen for most of the work presented in this thesis. To evaluate the capabilities and maturity of the DESTECS tool, it was compared to another simulator with roots in academia — the Ptolemy tool. In addition, the comparison was made to broaden the scope of the thesis, to not only include the DESTECS tool. It is not the intent of this comparison to declare a winner, but rather to provide an in-depth investigation of the multi-disciplinary modelling tools, highlighting the pros and cons of each.

Section 8.2 summarises related comparison work on simulators. In Section 8.3 a case study of an aircraft fuel system is introduced as is the modelling of the case in the two tools. Section 8.4 describes the comparison criteria used — both qualitative and quantitative, and Section 8.5 reports the results of the various comparisons done. Finally, a summary is given in Section 8.6.

8.2 Related Comparisons of Simulation Tools

In the Columbus project [48] ten different tools and techniques for modelling multi-disciplinary systems are reviewed. The syntax of each simulator is presented, as well as the different domains they are capable of modelling. A very simple example of a bouncing ball or an electrical RC circuit is modelled to try out the capabilities of each tool. These examples do combine DE and CT elements, but are not good examples of multi-disciplinary embedded systems since they do not include a discrete controller which interacts with a continuous plant.

Bretenecker et al. are working on an ongoing series of classifications and comparison of simulators for physical modelling [39]. The focus of this work is on the physical modelling capabilities of different simulation tools, but in a recent revision multi-disciplinary modelling and co-simulation have been added to the comparison. The *Arbeitsgemeinschaft Simulation* (ARGESIM) benchmarks [40] (which define 19 different case studies) are used to evaluate more than 20 different simulators. The work done is extensive but focuses on how to solve the different case studies and lacks a more in-depth introduction and comparison of the capabilities of the tools.

In the work of Verma et al. [187], research in software evaluation and comparison is reviewed and a large set of comparison criteria are derived. The tools evaluated are of less interest than the comparison criteria used — we have been inspired by these criteria for the comparison presented here. An important conclusion made by Verma et al. is that no single tool is best for every task. For that reason, the comparison presented here is not an attempt to find the best tool, but rather to evaluate the use of the two tools for solving various tasks.

8.3 Case Study Description

The fuel systems on-board aircraft are complex systems, with multiple tanks in each wing as well as in the tail of the aircraft. Fuel must be transfered between these tanks to ensure the engines always have a sufficient supply of fuel and to maintain a proper balance of the aircraft front to back as well as sideways. As a means to manage this complexity, computer models have been developed of such systems for the last twenty years (as an early example, see [57]).

For this tool comparison, a simplified version of the case presented by Jiminez et al. [101] is used, where only the left side of the aircraft fuel system is modelled and the outermost tank in the wing is removed. An overview of the fuel tanks in our study and how they are connected can be seen in Figure 8.1.

There are three tanks in our fuel system case study: the feeder tank (LFT) supplying the engine with fuel, the middle tank (LMT) carrying extra fuel and the trimmer tank (TT) placed in the rear end of the aircraft, which is used to balance ("trim") the aircraft. At takeoff, TT must be empty for safety

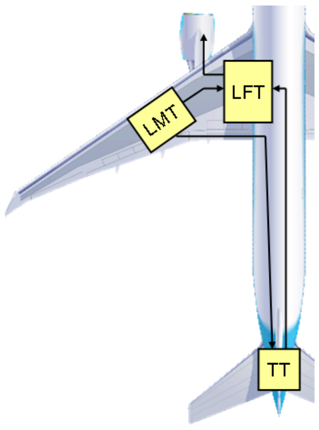


Figure 8.1 Overview of the the fuel tanks in the fuel system case study.

reasons, but once cruise altitude has been reached TT must be filled to obtain better balance of the aircraft. The LFT must always have sufficient supply of fuel to the engine which continuously consumes fuel (more during takeoff than at cruise altitude). If the fuel level of the feeder tank reaches a minimum threshold, fuel must be pumped from the LMT to the feeder tank. If LMT is empty, fuel must be transferred from TT to ensure a sufficient level of fuel for the engine. Finally, before landing the aircraft TT must be emptied for safety reasons.

Two scenarios are used in the comparative study:

"Sunshine" scenario: The aircraft takes off from ground level, and when reaching cruise altitude stops its ascent. The aircraft stays at this altitude until LFT reaches the low threshold at which point the aircraft starts descending. This scenario tests the normative behaviour of the fuel system.

"Faulty" scenario: In this scenario the TT fuel level sensor malfunctions and no signal is sent to the controller. This scenario tests that the controller can handle the malfunctioning sensor, while maintaining a sufficient supply of fuel to the engine ensuring a safe landing of the aircraft.

8.3.1 Modelling in Ptolemy

Airplane fuel system models in Ptolemy are studied in [62]. The various models highlight challenges when modelling multi-disciplinary systems. In this chapter, the fuel system model is extended to a more realistic version which is presented in Figure 8.2.

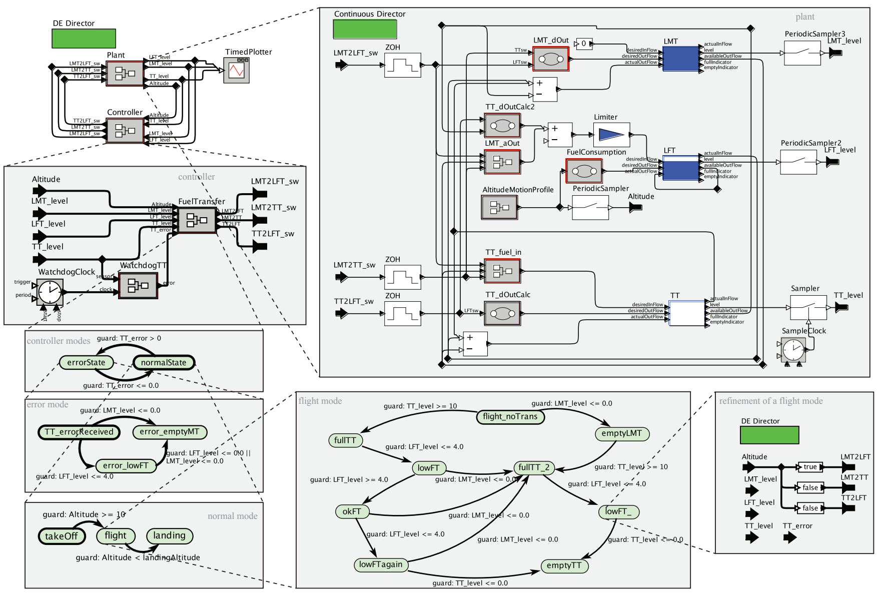


Figure 8.2 The multi-disciplinary fuel system model in Ptolemy.

The top level model consists of two composite actors: a controller and a plant. As the top level *Model of Computation* (MoC), DE was chosen. The plant is modeled with a CT MoC, and the controller inherits the DE MoC from the top level. Inputs to the plant model are discrete and have to be converted to continuous signals with zero-order-hold components. Outputs are discretised with samplers. The plant comprises: the three tanks; fuel mover components that transfer fuel between the tanks; and an altitude profile. The control logic is specified via modal models and mode refinements define the control outputs. Modal models are used to express:

- Normal and faulty operation. An input to the modal model (TT_error) is checked and based on the value of this input, the mode is switched. Refinements of the states express the different behaviors;

- The flight modes takeoff, flight and landing;
- Various error modes; and
- Various modes that, based on certain fuel levels in the tanks, move fuel between the tanks. This represents the main control logic.

Figure 8.3 shows the outputs of the simulation. The fuel levels of the three tanks as well as the altitude of the aircraft are monitored.

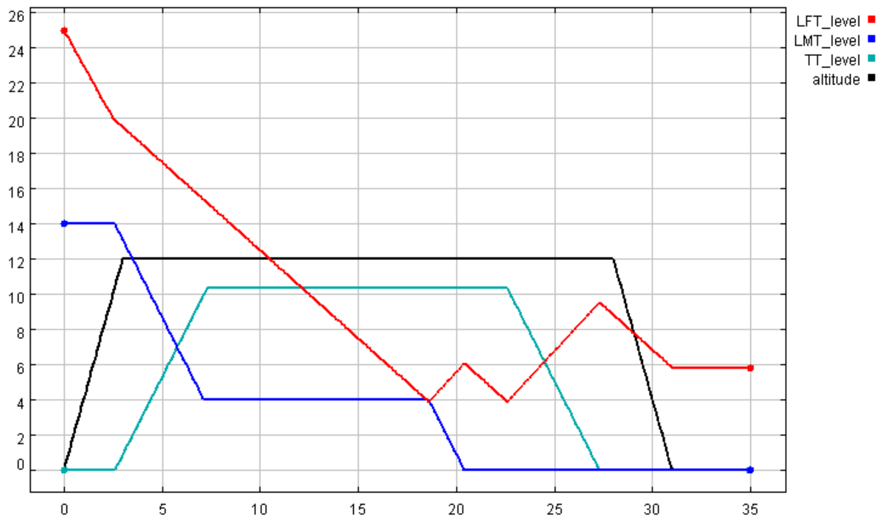


Figure 8.3 Fuel levels obtained through simulation in Ptolemy.

8.3.2 Modelling in DESTECS

The three fuel tanks –as well as the dynamics of transferring fuel between these– were modelled as a CT model in the 20-sim tool, while the discrete control scheme was added as a VDM-RT model in the Overture tool. 20-sim includes pre-defined iconic blocks for modelling components in the hydraulic domain which could have been used in order to add realism to the model with regards to the flow of fuel; volume components for the tanks; pump components; and hydraulic inertia in the pipes connecting the fuel tanks. To better compare the two tools it was chosen to model the tanks using an equational approach similarly to what was done in the Ptolemy tool. Figure 8.4 gives an overview of the DESTECS fuel system model in the 20-sim editor.

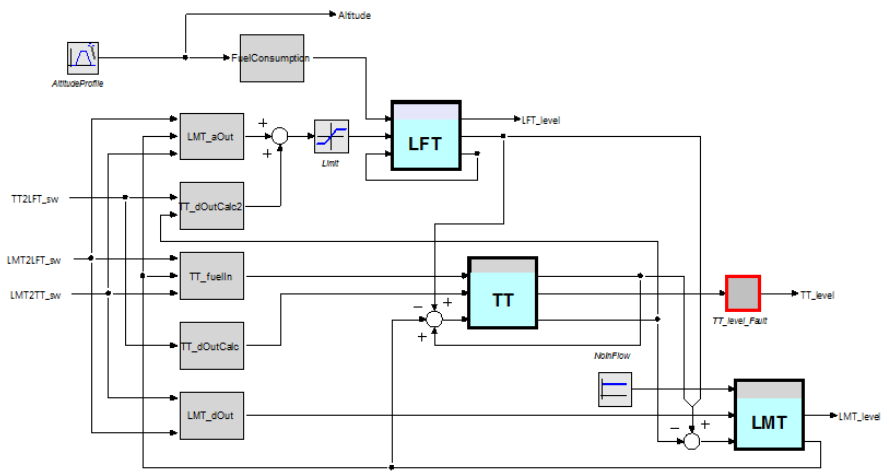


Figure 8.4 The fuel system model in 20-sim.

VDM-RT, being an object-oriented language, enables the modeller to express the DE controller using inheritance, polymorphism and composition. The faulty sensor was modelled as a concrete subclass of an abstract sensor class. To run the faulty scenario the faulty subclass was instantiated for the TT fuel level sensor and the normal sensor subclass for the remaining sensors of the system. To trigger the faulty sensor the DESTECS Command Language (DCL) was used. Below the simple DCL script used to activate the faulty TT level sensor after 3.9 seconds of simulated time is shown. Separate launch configurations were created for the normal and faulty scenario ensuring that the correct concrete sensor class implementations were used and that the DCL script was activated for the faulty scenario.

```
when time >= 3.9 do
(
    ct boolean TTLEVELERROR := true;
);
```

To visualise the complete system model the 3D animator built into 20-sim was used. To add to the visual presentation an aircraft model was imported, and simple cubes were added as the fuel tanks — a screenshot from the resulting 3D animation can be seen in Figure 8.5.

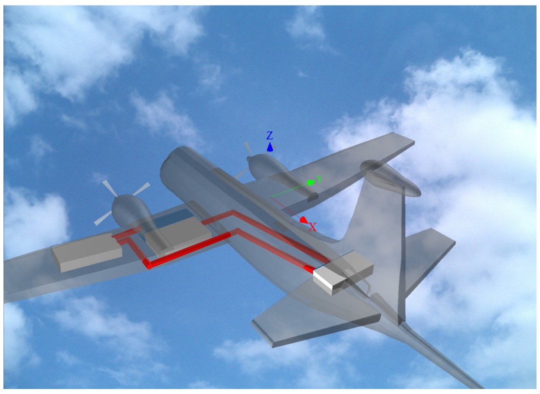


Figure 8.5 Screenshot of the 3D animation of the DESTECS fuel system model.

8.4 Comparison Criteria

The comparison criteria used during the comparison of the two tools are divided into three parts: usability criteria giving an overview of the accessibility of the tools; quantitative criteria comparing the tools on simulation speed and similar measurable metrics; and finally qualitative criteria giving an overview of the pros and cons of each of the tools.

8.4.1 Usability Comparison Criteria

This part of the comparison focuses on the usability of the two tools: how easy is it for a new user to install the tool and get initial help when technical issues are encountered? The usability criteria are divided into the following sub-categories:

Installation: This category covers the installation experience of the tools. The following gives examples of questions that are answered in this category: How easy was it to install the tool? Do the tools depend on any additional software (Java runtime environment or similar) to be installed in order to work, and are these installed automatically as part of the installation process? Are there any license required to use the tool, and how easy is it to acquire such a license?

Updates: This category covers anything related to updating the tools. How frequent are public updates made available? How are users notified of these updates? Is it possible to use developer builds? If yes, how easy is it to access these and what is the frequency of release?

Model management: This category describes how easy it is to manage the models created in the tools: How many files are created for a single model, and how are they organised? Do the tools support the use of different alternative implementations of submodels, and how easy is it to switch between these?

Extensions: How easy is it to extend the tool with additional capabilities? This category analyses internal extensions with the addition of new functional blocks for example, as well as external extensions like linking to other tools or export of data to be used by other tools.

Help: This category answers questions like: How easy is it to get help concerning technical issues? Are there existing examples to get inspiration from? Is there an active community with a forum or mailing list to ask technical questions? How well is the tool documented with regards to user guidance? Is there an in-tool help function to quickly access the documentation of the tool?

8.4.2 Quantitative Metrics

This part of the comparison focuses on quantitative comparison criteria where the two tools are benchmarked against each other, resulting in metrics that are directly comparable.

The (co-)simulation speed, time to load the model, and memory consumption are all metrics which are compared. In addition, the number of ODE solvers each tool support is compared, divided into fixed-step and variable-step solvers. Table 8.1 lists the technical specifications of the laptop used for the quantitative comparison.

Laptop	Lenovo ThinkPad® T500
CPU	Intel® Core™2Duo T9400 (2.53GHz, 6MB L2, 1066MHz FSB)
Operating system	Windows® 7 Professional 64-bit
Installed memory (RAM)	8 GB
Graphical Processing Unit	ATI Mobility Radeon™ HD 3650

Table 8.1 Technical specifications of the PC used for the comparative tests.

8.4.3 Qualitative Comparison Criteria

In the qualitative category the following comparison criteria were chosen prior to creating the two models:

Discrete-event controller expressiveness: Are there limitations as to what can be modelled in the DE controllers in the two tools? Is it possible to model an object-oriented architecture? Do the tools support the use of abstract data types?

Hierarchical modelling: Do the tools support a hierarchical structure in the models?

Model reuse: How easy is it to reuse parts of the model? Can submodels be created and linked to other projects, or is model reuse only supported by copy-paste? Do updates to submodels automatically mitigate to projects in which they are used?

Fault modelling: How easy is it to model faults and fault-tolerance mechanisms? Do the tools contain implicit help to this or must it be done explicitly by the model designer?

Extensions of the model: How easy is it to extend models, for instance adding additional fuel tanks?

In addition to these comparison criteria defined prior to the comparative work, additional criteria were discovered during the creation of the two models. If at any time the model designers encountered some issue or something well supported by one of the tools, this feature (or lack thereof) was added to the comparison criteria. These are described under results in Section 8.5.4.

8.5 Results of the Tool Comparison

Version 1.3.3 of DESTECS from April 2012 and version 8.0.1 of Ptolemy from October 28th 2010 were used for all the tests done.

8.5.1 Usability — Ptolemy

Installation

Ptolemy is an open-source framework built in Java and it is released under the BSD license [151]. Installer and guides for installing on Windows, Linux and

OSX are available online and maintained regularly. Ptolemy comes with various packages where some are by default disabled and not compiled. In order to compile these packages, additional libraries might be necessary. Ptolemy can be executed as a standalone application or embedded into the Eclipse development framework.

Updates
Official updates of the tool are released on an annual or biennial cycle. It is also possible to work with the latest build of the source which is available via a subversion repository. As of beginning of 2012 the code base for the project takes about 1GB of hard disc space. An installation from the source code repository takes up to one hour and following the guides is a lengthy process though well documented. Installers are also built nightly to ease this process.

Model Management
Ptolemy models are stored as XML files in a syntax called MoML [131]. A model is stored as one or multiple XML files. Ptolemy includes a special mechanism for object-oriented modelling. A class can be implemented and instances of that class can be reused in different parts of the model. The object-oriented design principles in Ptolemy are unique [130]. Different versions of the model have to be stored in different files. There is no automatic support for version control.

Extensions
Ptolemy comes with a library of directors and actors. Compared to commercial tools such as MATLAB/Simulink, this library is fairly small. Extending Ptolemy with new functionality such as a new MoC or new actors means creating a new Java class which implements specific interfaces. Tutorials on how to implement a new actor or a new MoC are available online.

Help
The Ptolemy project is released with many demos that illustrate how to create models with different MoCs. Help is also provided via various mailing lists such as the Ptolemy interest mailing list or the Ptolemy hackers mailing list. The code base is constantly changing as a number of students, researchers and industrial collaborators are extending and experimenting with the tools. All extensions to Ptolemy are checked to ensure they still conform to the current code base. This is done by creating regression tests, which can be written in

Tool Command Language (TCL) or implemented in the model by using a special test actor. This actor *learns* the correct values in a *training run* and then checks subsequent executions of the model against these values. Code that is submitted to the source tree undergoes strict checks and has to adhere to coding guidelines which are documented. Documentation is fairly extensive and utilizes Javadoc [152] to automatically generate documentation.

8.5.2 Usability — DESTECS

Installation

The DESTECS tool is developed on top of the Eclipse [177] *Integrated Development Environment* (IDE) and official releases can be downloaded from SourceForge. In addition, nightly builds can be accessed directly from the build server. A part of the installer is the open-source IDE Overture. A free *viewer* version of the commercial tool 20-sim is also installed which gives access to all of 20-sim's capabilities except saving models. To enable this last feature the free license must be upgraded to a full professional license. 20-sim is only built for Windows (XP and newer versions) so the DESTECS tool only runs on this platform even though Overture is built for Windows, OSX as well as Linux.

Updates

Only a few official updates of the DESTECS toolchain have been released on an annual cycle. Developer builds used internally in the DESTECS project have been released on a monthly basis. As of October 2012 public releases outside the consortium have been made available on the project download page.

Model Management

Different formats are used for the different parts of a DESTECS co-model: 20-sim models are stored in the XML-based proprietary format *emx*, VDM models are stored in the *vdmrt* format and co-simulation settings are stored as launch configurations in the DESTECS tool.

20-sim does not support object-oriented modelling of the CT components like Ptolemy does. Instead, submodels can be created and imported into other models (multiple instances if needed). If changes are needed to this part of the model, only the separate submodel needs to be changed and the imported instances can be updated to reflect the changes. There is no automatic support for version control inside the DESTECS tool, but since it is based on Eclipse, third-party plug-ins like Subclipse [179] can be used for this task inside the IDE.

Extensions

Since the Overture tool is open-source it is possible to make extensions on the DE side of the DESTECS toolchain. Since 20-sim is a commercial tool, it is not possible to make extensions to the CT simulation tool. It is possible, though, to define new blocks being either equation or graphical based. This enables the model designer to manually describe the differential equations of the CT components or to create graphical representations using the underlying bond graph technology and saving these in a common submodel library.

Help

20-sim comes with a large library of example projects, exemplifying the use of: 1D, 2D and 3D mechanics modelling, block diagrams, bond graphs, electric motors, hydraulics, signal processing, and many more. Numerous sample VDM-RT projects can be downloaded from the project wiki page [155] to get a good introduction to DE modelling in the Overture tool. Currently only 15 publicly available co-model examples exist for the DESTECS tool — these are small example projects explaining the use of all of the capabilities of the tool.

8.5.3 Quantitative Comparison Results

Clean installations of both tools were made in order to gather data on the installation process. The startup of the tools was monitored as well as the duration of running (co-)simulations of the two scenarios defined in Section 8.3. Finally, consumption of system resources was monitored. Results of the quantitative comparison can be seen in Table 8.2.

The Ptolemy installer takes a lot longer to run than the DESTECS equivalent, but most of the installation time was spent on installing the full source code of Ptolemy. If the source code is not needed, installation time as well as the final size of the tool are decreased.

Since DESTECS is built on top of the Eclipse platform, models are not loaded into the tool. Instead they exist in a workspace which is automatically loaded as part of the tool — hence no measurement of model loading duration was possible for the DESTECS tool.

Criteria	**DESTECS**	**Ptolemy**	**Unit**
Installer size	147599	229022	KB
Installation duration	0:59	2:53	min
Size of installed tool	264	229	MB
Time to startup tool	4.3	3.1	sec
Time to load model	N/A	3.7	sec
Size of model	1098 (167)	672	KB
Simulation speed			
- Scenario 1	7.3	6.1	sec
- Scenario 2	18.3	5.1	sec
Memory consumption			
- Idle (model loaded)	167540	212740	KB
- Peak (during simulation)	260640	580284	KB
CPU consumption			
- Idle (model loaded)	0	0	%
- Peak (during simulation)	94	99	%
# ODE solvers			
- Fixed step	5	2	
- Variable step	5	2	

Table 8.2 Overview of the qualitative comparison results.

As described in Section 8.3.2, a 3D animation was created as part of the DESTECS model. The core objects of the aircraft model was created in an external 3D modelling tool called Blender [34] and imported into the DESTECS model. The size of these objects were 931KB in total, so to compare the model size directly with Ptolemy (the Java3D interface of Ptolemy was not used in the work presented here) these objects should be subtracted, resulting in a DESTECS model size of only 167KB.

In the DESTECS tool, simulation of the faulty scenario was three times slower than the "sunshine" scenario. In order to trigger the error an additional value had to be passed between the Overture and 20-sim tools of DESTECS. The co-simulation engine is only optimised with regards to the passing of values defined in the contract between the two tools, and not these additional values triggered via the DCL script. This is the reason for the significant performance decrease in the simulation of the faulty scenario in the DESTECS tool.

The Ptolemy tool performed better in the faulty scenario. This could be due to the fact that when the TT fuel level sensor fails, Ptolemy stops drawing the fuel-level in the 2D graph. This indicates that rendering the graphs is quite demanding in Ptolemy, so reducing the number of signals plotted in 2D graphs will increase performance of the tool.

When running multiple subsequent simulations the peak memory consumption of Ptolemy slowly increased until settling at the value listed in Table 8.2. This indicates that the tool has some memory of state between simulations. If it had been a case of a memory leak, the memory consumption would keep increasing — this was not the case.

Since 20-sim is a fully fledged CT modelling tool, it has more diversity in the choice of ODE solvers. The wider choice can help optimising the duration of simulations since it is easier to find one that is suitable for the model at hand. Optimisations like this requires a deep insight into the CT model though, so they are reserved for experts in this area.

8.5.4 Qualitative Comparison Results

This subsection describes several specific benefits and limitations of the two tools which were discovered during creation of the models.

DE controllers

In Ptolemy, the DE controller was modelled as a hierarchically layered modal model: the top layer switching between normal and faulty mode, and individual underlying modal models managing the transfer of fuel between the different tanks in each of these two modes. The hierarchical structure is achieved by making new modal models as refinements of the individual modes in a modal model higher in the hierarchy. Examples of this can be seen in Figure 8.2. This has the unfortunate side effect that the controller is 7 levels deep and has 15 individual modal models (even without counting all the individual state refinements which set the value states) which makes it complex to navigate. As an alternative, the controller could have been modelled using a more flat structure. This would have made the model easier to navigate, but the benefits of having a hierarchical model would be lost.

The use of a modal model description of the controller also have some major advantages though. In embedded software the controller is commonly structured as a state machine since the controller has state dependent behaviour. Using the same notation in the model makes it easier for embedded software engineers to learn how to use the tool without having to learn all the syntactic subtleties of a new language. Another benefit, is that the fault tolerance mechanism is encapsulated in its own mode — this reduces what might crudely be termed *pollution* of models with descriptions of these non-ideal behaviours.

In the DESTECS tool the model designer has a full object-oriented language to express the DE controller. This is a major advantage when more complex control algorithms are needed, which can be harder to describe using state machines. The object-oriented structure enables the model designer to try out different control strategies by having an abstract controller main class with several alternative concrete controller implementations each specify different control schemes. This approach was used when modelling the faulty sensor which was defined as a concrete subclass of an abstract sensor class. To run the faulty scenario, the faulty subclass was instantiated for the TT fuel level sensor and the normal sensor subclass for the remaining sensors of the system. This is a great way of using alternative versions of the model in different scenarios without polluting the model with unneeded information and without forcing manual alterations to the model.

CT plant

The CT plant model in Ptolemy was created using a signal-based block diagram. When modelling the altitude profile for the aircraft an expression actor was used, manually ramping the output from zero to cruise altitude; stay at cruise altitude for a certain period of time; and finally descent to zero altitude for landing the aircraft. This was done using three nested if-then-else constructs written using the ternary operator, since the expression actor only allows single line expressions. This makes the expression hard to read and a source of errors. In DESTECS the CT model was created in the 20-sim tool. To model the same altitude profile the built-in *motion profile wizard* was used which guides the model designer through a series of steps constructing the desired signal. This is done by making a single or continuous combination of a series of ramp, constants, (co)sine or similar types of sources. The user gets a graphical overview of the generated output to visually validate that the correct motion profile has been generated. This is a much more user-friendly approach to creating custom made sources compared to the Ptolemy equivalent.

The simpler (so-called *signal level*) model of the fuel tanks was built in 20-sim. A key reason for this was to permit a better comparison with Ptolemy in terms of functionality of the CT model. This choice means that if a tank is full the back pressure must be modelled explicitly between each pair of tanks. The bond graph notation supported by 20-sim could be used here, which offers a more compositional solution. All connections are “two way" so that the back pressure from a full tank is automatically taken into account. Further tanks can simply be connected and the causality determined runtime. An initial bond graph model of the fuel tanks has been developed by Qian [164]. This model includes more realistic tanks, pipes and pumps, with the flows of fuel appearing as curves on the plots. The model also includes a calculation for changing the centre of gravity during the flight (along the nose-tail axis of the plane only), bringing it closer to the original model by Jiminez et al. [101]. Note that it was possible to plug this alternative CT model directly into the existing co-model, permitting the use of the same DE controller and test scenarios.

Fault Modelling

In the Ptolemy model, the TT fuel level sensor is monitored using a *watchdog*. If two consecutive readings are missing an error is issued, which switches the DE controller to its error state. Ptolemy includes an alternative implementation of fault handling [77]. This mechanism allows for models to throw errors in the same manner as programming languages such as Java do. These errors are propagated up the hierarchy until there is a model that can deal with the error. A special error transition in a modal model catches the error and the model switches to a fault mode. The cited paper mentions how to deal with timing errors in such models but this error handling mechanism could be extended to arbitrary errors like the one in the fuel system case study presented here.

One of the main goals for the DESTECS project, when modelling realistic and faulty behaviour, is to avoid pollution of models with fault and fault tolerance descriptions. Hence, it is advocated to keep a clear distinction between ideal and realistic/faulty behaviours in models. One way of achieving this is by using the DCL scripts as described in Section 7.2.3. Using a combination of subclassing to model both normal and faulty sensors and using the DCL script to trigger the error is a great way of ensuring that the (de)activation of the faulty sensor is decoupled from the rest of the model without polluting the model.

Visualisation of the Simulations

The 2D plotters of Ptolemy are simple and user friendly. When comparing variable values which exist at different hierarchical levels of a model, the signals cannot seamlessly be dragged to the plotter, though. To plot two such variables in the same window they must be passed in the interface of the hierarchical blocks. Alternatively, the MATLAB interface of Ptolemy can be used, passing the relevant variables and plotting these in MATLAB. In the 2D graphs of 20-sim the user chooses any variable from the model which is then plotted, which makes it easier to compare variable values existing in different parts of the model.

As mentioned in Section 7.2.1, 20-sim also includes a 3D animator which can be used to give an even better representation of the system model. Such 3D animations have been used with great success to explain the functionality of co-models in the DESTECS project. Ptolemy has a Java3D interface which can be used with similar results.

8.6 Summary

As stated at the beginning of this chapter, the intent of the tool comparison was not to declare a winner, but rather to offer the reader insight into their use on a common case study, presenting information that can help users make an informed choice about which tool to use. Both tools were able to model and simulate the fuel system case study, including modelling the plant in the CT domain and the controller in the DE domain. Both tools were able to model a single fault in a sensor with associated fault tolerance in the controller.

There are a few key ways in which the work on this chapter could be expanded to give a better comparison. First, the case study was deliberately kept simple, however this does not necessarily exercise both tools to their full potential. The current fuel system case study could be expanded to include more features such as more tanks, more faults and fault tolerance. Especially the CT elements of the case study are simplified — fuel moves instantly between tanks; fuel transfer is not affected by gravity; and the movement and rotation of the aircraft is completely abstracted away. Adding some of these element would ideally push both tools towards their limits and demonstrate if one tool is better than the other *for certain situations*. Another way to compare the tools is to try other models from different domains. This might offer a way for users to decide between the tools based on the domain of interest.

Finally, an extension of this comparison to other tools would be of great interest and offer insight into a broader range of modelling and (co-)simulation capabilities. The definition of challenge problems together with a set of evaluation criteria for tools would allow for better comparison and evaluation for specific needs of the tool users. The comparison criteria and case study defined in this chapter, could be used as a starting point for creating such a common platform for collaborative modelling and (co-)simulation tools.

9

Collaborative Modelling Guidelines

This chapter provides guidelines for modelling multi-disciplinary embedded systems. The guidelines support the use of collaborative modelling and (co-)simulation tools like the ones introduced in Chapter 7 and compared in Chapter 8. The guidelines are put into the context of a four-phase iterative spiral model inspired by the mono-disciplinary process of Chapter 4 and the agile process of Chapter 5. The value of the collaborative modelling guidelines is evaluated in Chapter 10.

9.1 Introduction

A major challenge in the modelling of multi-disciplinary systems, lies in the combination of models used by the different disciplines involved in the development of the system. This chapter provides guidelines supporting the process of creating collaborative models of multi-disciplinary embedded systems.

For the work presented in this thesis, the DESTECS toolchain was used for collaborative modelling and co-simulation. The guidelines were developed in conjunction with the DESTECS toolchain supporting various tool-specific constraints and capabilities. With minor modifications, the guidelines are applicable to most of the multi-disciplinary modelling tools mentioned in Section 7.3.

Section 9.2 gives an introduction to the modelling language SysML [150] which is used throughout the following chapters. Section 9.3 gives an overview of a four-phase modelling process the guidelines support. Each of the four phases of the process are described in the following four sections: model purpose and requirements in Section 9.4, system decomposition in Section 9.5, system modelling in Section 9.6 and analysis of the models in Section 9.7. Finally, a short summary is given in Section 9.8.

9.2 Overview of the Systems Modelling Language SysML

Since the methodological guidelines described here advocate the use of a more holistic system view, using techniques commonly used by systems engineers, is appropriate. SysML [150] 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. This section gives a quick overview of some of the SysML diagram types and constructs. For a more in-depth description of SysML the reader is advised to seek one of the numerous SysML books available e.g. [78, 91].

9.2.1 Requirements Modelling

SysML provides modelling constructs to describe requirements and expected system behaviour and link these to other elements of the system model. High level system behaviour can be described in *use case diagrams*, which are directly inherited from UML. In these diagrams, the interaction between the system and various *actors* is defined. Use cases excel at describing user stories and can be used for deriving the system requirements. *Requirements diagrams* are a new type of diagram in SysML, where sub-requirements can be *derived* from parent-requirements or a *refinement* relation can link requirements. Test cases can be linked to requirements with the *verified by* relation and system components can *satisfy* a certain requirement.

9.2.2 Structure Modelling

The main modelling element in SysML is the *block* which is a modular unit of the system. A block can represent anything: a system on its own, a process, a software component, a function or a context in which something happens. Blocks are arranged hierarchically in *Block Definition Diagrams* (BDDs) where their relations are described graphically. BDDs are modified versions of UML class diagrams. Using *composite associations* part properties of the block representing subsystems or components can be described.

The internal structure of a block is described in an *Internal Block Diagram* (IBD) where the connections between properties are defined. The interface of the internal parts can be described using *ports* which are divided into two types: *standard ports* which are mostly used in software design and *flow ports* which are typed ports ensuring that the data, energy or material passing through the port is well defined.

9.2.3 Behaviour Modelling

SysML defines three diagram types to specify system behaviour, which are all directly inherited from UML. The *sequence diagram* describes message oriented communication between and internally in blocks. Messages can be interpreted as command requests between actors and the system or between system parts inside the system itself.

Activity diagrams are used to give a graphical representations of workflows and stepwise activities (rounded rectangles). The diagram has support for choice (diamonds), iteration and concurrency (split or join bars).

Statecharts are used to describe system state and behaviour and the transitions between states. States are represented as rounded rectangles annotated with state names. The transitions between states, represented as arrows, are labeled with the triggering events followed optionally by the list of executed actions.

9.2.4 System Constraints Modelling

Parametric diagrams are a completely new diagram type in SysML, that are used to describe constraints on system properties. *Constraints* are mathematically or logical expressions that precisely define performance and quantitative constraints on system parameters. In parametric diagrams, constraints can be nested in order to create complex constraints from more basic ones.

9.3 Collaborative Modelling Process Overview

The exploration of alternative designs is a creative activity. It is hard to estimate the time the exploration of design alternatives will take, so forcing a design space exploration to fit into a fully fledged development process will tend to hinder creativity. This is the main reason why this chapter focuses on describing methodological guidelines which can be used as tools during collaborative modelling. To describe the different phases involved in model development and design space exploration, and to put the guidelines into this context, the spiral model is outlined in Figure 9.1.

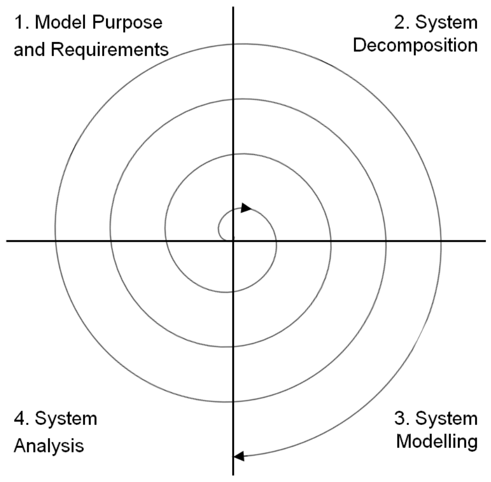


Figure 9.1 Overview of the iterative spiral model that the methodological guidelines supports.

These four phases are inspired by the steps of the mono-disciplinary guidelines from Chapter 4: the model purpose and system decomposition phases are inspired by the system boundary definition; the system modelling phase is inspired by the various VDM models produced; and the system analysis phase is inspired by the validation described in the mono-disciplinary guidelines. Due to its iterative nature, the spiral process can also fit into an agile setting as described in Chapter 5. In that case, each iteration of the process must fit into a single Scrum sprint.

Each of the four phases are described in detail in the following four sections. The four phases should be followed sequentially through several iterations. The guidelines supporting each of the individual phases can be used in the order the model designer finds most useful.

9.4 Model Purpose and Requirements

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.

As part of the process of defining the model purpose, it is advised to document all (implicit as well as explicit) assumptions made of the system as well as any definitions agreed upon: positive rotation direction, available hardware, the environment in which the system must operate, units on system parameters and so on. This process is often enough to discover many simple cross-disciplinary errors and misconceptions. Using a list of assumptions can also help defining the purpose of the co-model, ensuring that all involved in the project agree on the main purpose.

Guideline multi_1: Document all assumptions made of the system as well as definitions agreed upon.

One of the key aspects of the model purpose is to define 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. Modelling faults is a key aspect in DESTECS collaborative modelling — use cases are useful for this. Both normative and faulty interactions with the system can be modelled, and since these are separate actors the normative and faulty behaviour of the system are kept separate which is one of the goals of the DESTECS project.

Guideline multi_2: Define the purpose of the model by identifying all actors and use cases of the system, documenting them in a SysML use case diagram.

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.

Guideline multi_3: Derive requirements of the system from the use cases defined.

9.5 System Decomposition

Once the model purpose and requirements have been determined, the system must be decomposed into its main parts. A 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*.

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.

Guideline multi_4: Define the main parts of the system in a *Block Definition Diagram* and determine which domain each individual block belongs to.

9.5.1 CT Constructs

To add levels of detail to the SysML model, an 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).

Guideline multi_5: Define the internal composition of the main blocks using *Internal Block Diagrams*. The interfaces between child-blocks are defined using ports.

To specify constraints on the parameters of the system, an additional BDD can be made, containing *constraint blocks* which define constrains on the physical properties of the system. Such constraints can also be used to identify critical performance parameters and their relationships to other parameters — this is done in a *parametric diagram*. A parametric diagram that is a child of a block shows the usage of one or more constraint blocks within the context of the owning block. This diagram shows how constraint block parameters are bound: to value properties of the parent block, its parts or parameters of other constraint properties. 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 20-sim calculates the causality run-time.

Guideline multi_6: Use a combination of the *constraint blocks* and *parametric diagrams* for documenting the differential equations of the system as well as the causality between these.

9.5.2 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 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.

Guideline multi_7: Use UML class diagrams to define the structure of the DE controller.

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.

Guideline multi_8: Use UML behavioural diagrams to define the intended behaviour of the DE controller.

9.5.3 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.

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.

Guideline multi_9: For the co-simulation contract, derive the *monitored* and *controlled shared variables* from the interface between DE and CT blocks in the *internal block diagrams*; the *shared design parameters* from the *parametric diagrams*; and the *events* from *sequence diagrams*.

9.6 System Modelling

Once the system has been decomposed and the internal details have been specified, the next phase of the process is the system modelling. In this phase the co-model is produced using an iterative approach.

9.6.1 Modelling Approach

Partners from the DESTECS consortium have described different approaches to collaborative modelling called DE-first and CT-first [73]. These approaches advocate the construction of an initial mono-disciplinary model in order to reduce the complexity and simulation overhead. In addition, an approach called contract-first has been defined, where the contract is defined first and two constituent models are constructed in parallel using the CT and DE formalisms:

CT-first: The entire system is initially modelled in the CT formalism — DE elements are either modelled in a simpler form in the CT tool or stubs are used to enable testing of the CT model without the DE controller.

Consider CT-first if the system mainly consists of CT elements with only simple controllers. An example of such a system is a temperature controller that needs to maintain the temperature at a preset value, using e.g. a PID controller and a thermostat to achieve this. If legacy models exist in the CT domain which can be used directly or adapted to describe the continuous behaviour of the system the CT-first approach is recommended.

DE-first: Focuses on modelling a high fidelity controller, so the entire system is initially modelled in the DE formalism. The CT elements of the system are either modelled as discrete abstractions (see for example Alur et al. [13]) or stubs are used to enable isolated test of the DE model.

Consider DE-first if the main complexity of the system is in the DE domain or the continuous behaviour of the system can be discretised while still maintaining a competent model. If the controller only needs to monitor the CT model without directly affecting the environment, the CT input can easily be modelled in the DE formalism as discrete input values. If legacy models exist in the DE domain, which can be used directly or adapted to describe the controller of the system, the DE-first approach is recommended.

Contract-first: The interface between the CT and DE models is described first, and based on this, the two domains are modelled in parallel. In this approach, it is beneficial to use stubs to enable testing of each of the models in isolation before combining them to the full co-model.

Consider contract-first if two separate teams have to work on the two domain-specific parts of the model. This avoids one team waiting for the other to generate the baseline for the co-simulation.

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 determining which parts of the system contains the main complexity and hence helps determining the most appropriate modelling approach.

Guideline multi_10: Determine the correct modelling approach (CT-first, DE-first or contract-first) by analysing the system decomposition and the background of the team as well as available legacy models.

9.6.2 Modelling Guidelines

With the detailed system decomposition completed in the previous phase, the initial steps of creating a DESTECS co-model is made less complex. For each of the internal blocks, create an empty submodel in 20-sim with the input/output as specified in the internal block diagrams. The internals of the blocks in the CT domain can be modelled using either bond graphs or the iconic diagrams and equation submodels supported by 20-sim. Using the differential equations defined in the parametric diagrams eases the task of creating the initial behaviour of the submodels in the CT domain.

Guideline multi_11: For each of the internal blocks, create an empty submodel in 20-sim with the input/output as specified in the internal block diagrams. Use the differential equations specified in the parametric diagrams for describing initial continuous behaviour.

When developing the CT parts of the co-model, it is generally recommended to start simple and slowly add details to the model in small incremental steps. The reason for this is that the more differential equations the CT model needs to evaluate, the harder it is to track down errors or ODE solver problems caused by an unstable CT model. It is hard (or even impossible) to debug the model stepwise in CT modelling tools, so tracking down errors is much easier when only doing small incremental steps between simulations.

Guideline multi_12: When developing CT models start simple and add functionality in small increments — this limits the risk of issues with the ODE solver and eases debugging.

Using idealised parameters further simplifies the model. It is advocated to start with ideal sensors and only add more realism, like noise in the analog-to-digital conversion, once the idealised model is working as intended.

Guideline multi_13: Start by creating an idealised model where all parameters are ideal and any possible fault scenario cases are disregarded.

When creating models it is of great importance to discover the significant parameters of the model — the parameters that have the greatest impact on the model performance when changed. These are the parameters that define the scope of the design space that needs to be explored in order to ensure optimal system performance. Once an idealised co-model has been created, the significant parameters can be found using the parameter sweep tool of 20-sim or the ACA tool. These tools automate the task of changing the value of key parameters in order to analyse co-model performance impact.

Guideline multi_14: Find the significant parameters in the co-model using parameter sweeps or “Automated Co-model Analysis”.

Once the significant parameters have been identified, it is advised to put these into the co-simulation contract as shared design parameters. This ensures that all the significant parameters of the co-model are located in a single place and not scattered all over the co-model. In addition, the parameters actually *need* to be SDPs in order to enable the use of the ACA tool of DESTECS.

Guideline multi_15: Put the significant parameters of the co-model into the contract as *shared design parameters* — this makes it easier to optimise the performance of the co-model using the *Automated Co-model Analysis* tool of DESTECS.

If encountering a complex issue with the model, isolate the problem in a separate model. Not only does this increase simulation performance, since fewer differential equations need to be evaluated, but it is also easier to understand the issue with fewer parameters to change.

Guideline multi_16: Isolate issues in submodels to ease debugging.

The object-oriented structure of the DE blocks is defined in UML class diagrams. Having the structure as well as operations and interactions defined in UML eases the task of creating the VDM-RT model using a process like the one described by Larsen et al. [119].

Many cross-disciplinary concerns are created by frequent changes to the interface not communicated to everyone affected. In order to lower the risk of such issues, we introduce the notion of *micro-* and *macro-steps*:

Micro-step: In a micro-step, changes are only done inside a single model. Examples of such changes are refactoring of the model to get a better structure, or adding functionality which does not require an interface update towards the other models of the co-model.

Macro-step: In a macro-step the changes introduced require updates to the interface. An example of such a change is adding functionality in the DE controller which requires additional input from the CT side, or adding actuators in the CT model that need to be triggered by the DE controller. System level changes affecting both domains (e.g. interpretation of positive rotation direction) are also macro-steps.

Having this clear distinction makes it much easier to discuss the impact of various changes to the co-model. We recommend that the number of macro-steps in each iteration is limited as much as possible, and are carefully communicated to all engineers affected. This ensures that any issues introduced by the interface changes are easier to track down.

Guideline multi_17: Limit the number of macro-steps in a single iteration of co-model development.

9.7 System Analysis

After each iteration of system modelling, the co-model needs to be analysed. The most important result of this phase is to determine whether the co-model fulfils the model purpose with a high enough degree of fidelity, or if additional changed and additions are necessary.

9.7.1 Validation Guidelines

Section 9.6.2 advocates the use of isolated submodels. Validating these submodels should be done outside the collaborative modelling environment, making use of domain-specific validation tools.

Guideline multi_18: Use domain-specific tools on isolated submodels to validate the functionality disconnected from the rest of the system.

Some possible domain-specific tools are listed below:

CT domain-specific tools

20-sim has built-in 2D graph and 3D animation toolboxes which can be used to visually validate CT models. 20-sim also has several tools for validating a model in both the frequency and time domain — these are explained briefly in Section 7.2.1.

DE domain-specific tools

The Overture tool, used to create the DE models, has a number of built-in tools as well. Debugging facilities enable the user to step through a simulation giving a detailed view of the execution. Breakpoints can be used to ensure that the execution stops at exact points in the model, and individual variables can be monitored. The *Proof Obligation Generator* [29] and *combinatorial testing* [123] tools introduced in Section 3.3.3 can be useful domain-specific tools for gaining confidence in the DE models.

One of the ways a co-model can be analysed is through *visual validation*. Using the 3D animator built into 20-sim is not the most precise way of validating correct behaviour of a co-model, but it is useful as an early "sanity check". Viewing a 3D animation makes it easier to spot obvious system-level errors that are easily missed on 2D graphs showing simulated values. The 3D animation is a way to reduce complexity — it is an abstraction of the model, only visualising the key properties of the model. Hence, it is also a great way of showing non-technical stakeholders the functionality of the co-model.

Guideline multi_19: Use 3D animations early in the validation process to ensure appropriate overall behaviour of the co-model.

Having a prototype of the embedded system being modelled available for measurements makes it much easier to improve the fidelity and realism of the co-model. The idealised parameters of the co-model can be tweaked by making small iterative changes based on feedback from measurements of the real system.

Guideline multi_20: Make the ideal parameters more realistic in an iterative process until the co-model and the real system behaves sufficiently similarly.

9.8 Summary

Collaborative modelling can be one of the means for managing multi-disciplinary system complexity. Each of the disciplines involved can create domain-specific models in the most suitable tools and combine the analysis of these in a co-simulation. A spiral model with four phases has been described, supported by 20 guidelines accompanying the DESTECS toolchain. The guidelines have been developed by generalising experiences gained from various collaborative modelling projects using the DESTECS tool.

The system dynamics are described using the CT formalism bond graphs and the embedded controller is described using the object-oriented DE formalism VDM-RT. These two domain-specific models are then analysed using co-simulation. The methodological guidelines supports various phases of co-model development: model purpose definition; system decomposition; system modelling; and system analysis. The guidelines mainly accompany the DESTECS tool, but with minor modifications they are applicable in various other tools capable of co-simulation.

The guidelines presented in this chapter are put into context by describing a simple four-phase iterative process, but more work is needed in order to describe the business perspective of collaborative modelling. Chapter 5 describes the use of formal specifications in an agile setting — namely in the agile project management process Scrum. The agile mindset, of self-contained teams consisting of people with experience within all necessary disciplines, fits well with the collaborative modelling philosophy.

The guidelines presented in this chapter provide a practical approach to co-model development which guides both novice and experienced modellers in managing multi-disciplinary system complexity by applying collaborative modelling techniques.

10

Evaluation of Multi-Disciplinary Modelling Guidelines

This chapter describes the different initiatives that were taken in order to evaluate the value of the multi-disciplinary modelling guidelines described in Chapter 9. A case study was conducted, adding CT element to the mono-disciplinary case study from Chapter 6; and an M.Sc. project evaluated the use of the guidelines through the development of a co-model of an autonomous robot. The findings of the various evaluation efforts are discussed in Chapter 11.

10.1 Introduction

The guidelines presented in Chapter 9 were developed by generalising experiences gained from various collaborative modelling and co-simulation projects using the DESTECS tool. To evaluate the value of the guidelines, various initiatives were taken. First and foremost, the mono-disciplinary ECAP case study from Chapter 6 was extended with CT elements to encompass: a helicopter; an IR guided missile; flares; and the existing DE model of a self-defense system intended to protect the helicopter from the incoming missile. The guidelines were followed strictly, and experiences documented.

It was also evaluated how well the guidelines supports the collaboration of engineers with different fields of expertise, by setting up an M.Sc. project where two students from software engineering and robotics engineering respectively created a co-model of an autonomous robot following the guidelines. A prototype of the real robot was also built, and it was studied to which extent the co-model resembled the real robot, and how well the co-model could predict how the real robot would react in different scenarios.

The extended case study is introduced in Section 10.2 and Sections 10.3 to 10.6 describe the application of the guidelines on the case study. Section 10.7 summarise the results of the case study, and Section 10.8 describes additional initiatives taken in order to evaluate the value of the guidelines.

10.2 Case Study Description

In Chapter 6 a self-defense system for aircraft called ECAP was introduced. A pure DE model of this system was created and analysed. To increase the level of fidelity of the model, a CT model of the dynamic behaviour of the dispensed flares was added. An IR guided missile was also added to the co-model, with rigid body dynamics and a complex missile guidance controller. This case study was called the electronic warfare case study.

The co-model was developed through three individual iterations - these are shown in Figure 10.1.

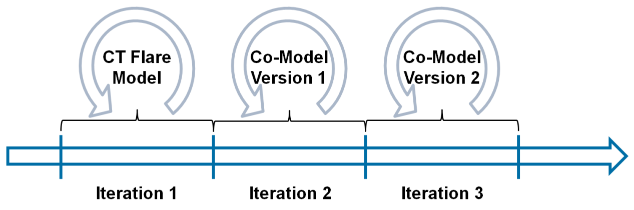


Figure 10.1 Overview of the three iterations of co-model construction.

A CT-first approach was used in the development of the electronic warfare case study. During the first iteration, a CT model of a single flare was developed. This was done to ensure that all the rigid body dynamics of the flare had been modelled correctly before proceeding to a full co-model. In the second iteration the initial co-model was created. In this model, the missile used a simple pure pursuit navigation strategy (see Section 2.4), and the helicopter could only dispense a single flare. In the third iteration the missile used a proportional navigation strategy, and the helicopter could dispense entire flare patterns, along with multiple other improvements.

10.3 Model Purpose and Requirements

The first three guidelines are concerned with the purpose of the model and describing additional requirements. The purpose of the co-model was to analyse how different generations of IR-guided missiles could be countered by dispensed flare patterns. It was decided to create a co-model to get a better understanding of:

- The trajectory of dispensed flares to get a visual representation of the position of these over time. Since modern missiles have complex tracking

algorithms, they simply filter out any IR sources which move in radical directions compared to the current target. Having a visualisation of how the aircraft and flares move over time would give a better understanding of which countermeasures are more effective;
- Scenarios seen from the missile point of view. Since missiles have a very narrow field of view to enhance their resistance against countermeasures, having a missile's point of view would give a better visualisation of how long flares stay within view of the missile and hence are effective;
- How different generation IR seekers work against different countermeasures; and
- Explore different dispenser placements, and their impact on the generated flare patterns.

Initially, a list of assumptions and definitions were documented as advised by Guideline multi_1. A few examples of such assumptions and definitions are provided below:

1. The missile has a 2 degree diagonal field of view.
2. Position, velocity, acceleration and forces are all 3D vectors giving both a direction and a magnitude.
3. Rotation in 3D space is described using quaternions.
4. The max velocity of the missile is 680 m/s.
5. The max velocity of the helicopter is 85 m/s.

In order to document the purpose of the model (Guideline multi_2) a use case diagram was derived from the textual purpose description. The main actors of the system were identified to be: the helicopter, the missile and the flares. The use cases identified reflect the purpose of the model. The helicopter must be able to detect an incoming missile and react to the threat by dispensing flares. The missile must be able to sense the helicopter and any flares currently deployed, find a main target, and steer towards this target using various generation of guidance mechanisms. The use case diagram can be seen in Figure 10.2.

Since this case study was a "proof-of-concept" without a real customer, it was not deemed necessary to specify a long list of requirements (Guideline multi_3). The purpose description and the use case diagram clearly defines what the co-model should include, and what it was going to be used for.

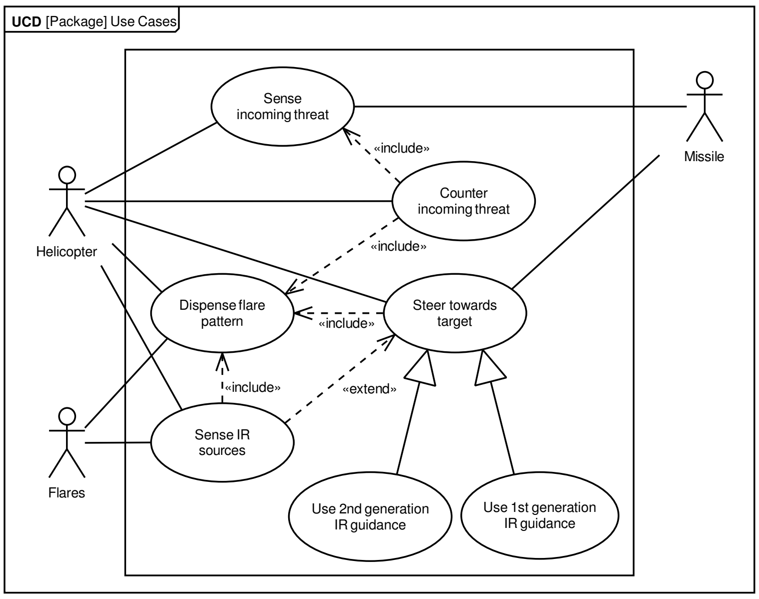


Figure 10.2 Use case diagram defining the purpose of the model.

10.4 System Decomposition

The use case diagram in Figure 10.2 was used as a basis for the initial system decomposition. The actors were used as the main blocks of the system model, and child-blocks were derived from the nouns used in the description of the individual use cases. Following Guideline multi_4 the blocks were structured in a Block Definition Diagram (BDD), and can be seen in Figure 10.3.

The main blocks (flares, helicopter and missile) are all nested blocks of the main countermeasure system model. Linking each of the child-blocks to its parent is done using the *composition* relation.

The missile is composed of: a physical airframe which represents the rigid body missile; an IR sensor which can monitor all IR sources within the field of view of the missile; and a guidance controller which finds a main target and steer the missile airframe towards this target.

The helicopter is composed of: a physical airframe which represents the rigid body helicopter; a Missile Warner System (MWS) which senses incoming threats; an Inertial Measurement Unit (IMU) which monitors the helicopter position, rotation and velocity; ECAP which is the main counter-

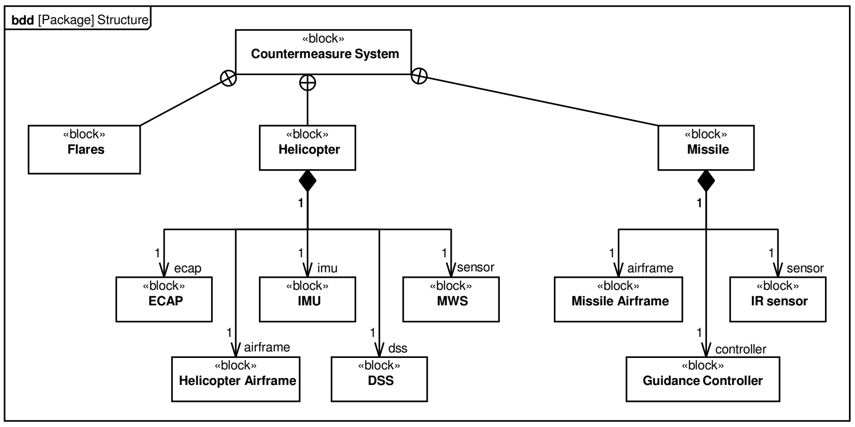


Figure 10.3 Block definition diagram defining the main blocks of the system along with any contained child-blocks. The missile Guidance Controller is the only DE block of the initial co-model. ECAP will be a future DE extension.

measure controller which calculates and executes the optimal countermeasure strategy against the current threat; and a Digital Sequencer Switch (DSS) which manages the dispensing of the individual flares.

The main parts of the model were defined and allocated to either the DE or CT domain (Guideline multi_4). The main purpose of the model was to precisely describe and monitor the flare trajectory. To create a high fidelity model of the rigid body dynamics of the flares they were modelled in the CT domain.

The helicopter airframe was also modelled in the CT domain. It was chosen to exclude a lot of details in the helicopter ECAP controller in the initial version of the collaborative model: once a threat was within a given distance of the helicopter, a flare pattern would be dispensed in the direction of the threat. The full ECAP model from Chapter 6 would be added as a DE component in later iterations of co-model construction.

The airframe of the missile was modelled in the CT domain. The determination of the main target and steering calculation maintained by the guidance controller was done in the DE domain. This enabled the use of an object-oriented structure, easing the use of different generation guidance algorithms and choose between these.

10.4.1 CT Constructs

To give a detailed specification of the internals of the helicopter block, an Internal Block Diagram (IBD) was created (Guideline multi_5). The five child-blocks and their interconnections were described. In addition, other child-blocks like the missile airframe and flare were added to give a complete description of the helicopter. The IBD diagram can be seen in Figure 10.4.

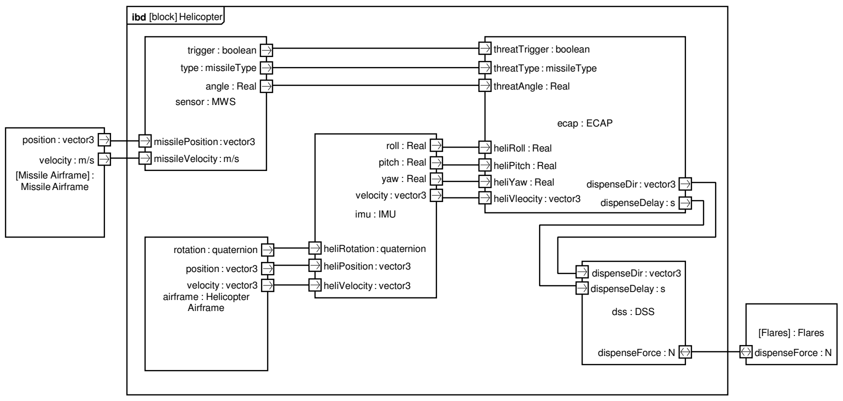


Figure 10.4 Internal block diagram specifying the internals of the helicopter.

An IBD of the missile was also created in order to specify the internals of this block. This diagram can be seen in Figure 10.5.

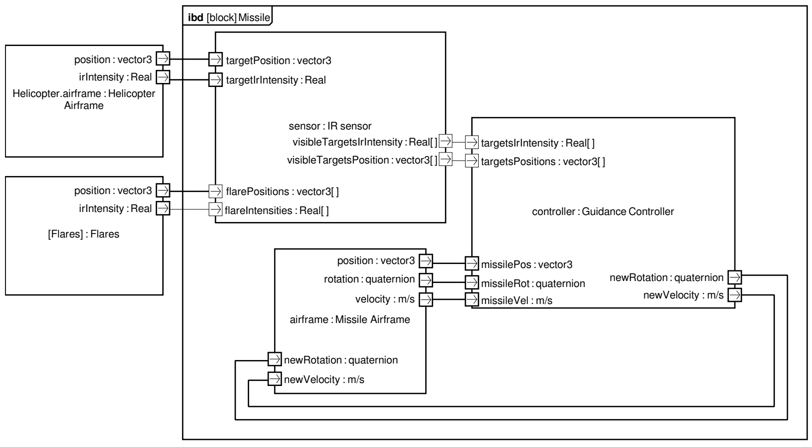


Figure 10.5 Internal block diagram specifying the internals of the Missile.

Instead of modelling the IR signatures of the flares and the helicopter as IR images, they were modelled as vectors describing their position on 3D space as well as a real valued variable describing the IR intensity. This abstract representation is sufficient for modelling first to third generation missile guidance. Accurate information on how fourth generation IR guidance actually works is extremely hard to obtain. Adding a highly abstract model of modern missile guidance could introduce uncertainty and reduce accuracy of the model, so it was decided to not include this in the model. Instead, visual analysis of the generated flare patterns would be used to validate the effectiveness of the countermeasures against fourth generation missiles.

Since the modelling of the flares was of utmost importance, a constraint diagram was created which describes the constraints on the physical flare parameters (Guideline multi_6). This diagram can be seen in Figure 10.6.

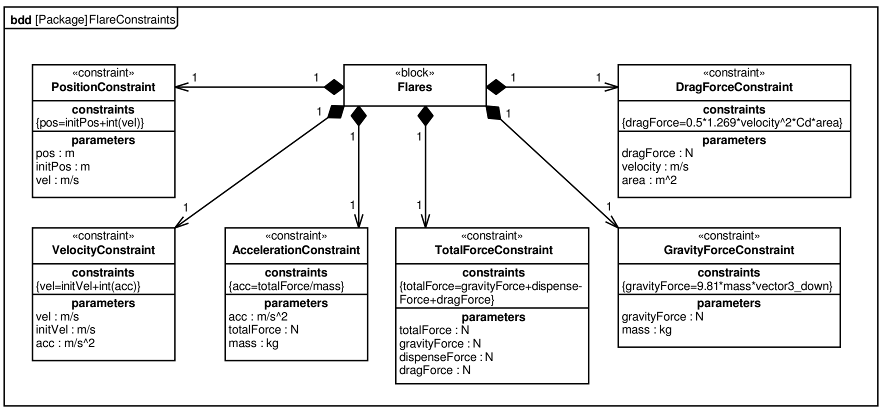


Figure 10.6 Constraint diagram specifying constraints on the flare parameters.

The model continuously calculates the 3D position of the flare. This was done by integrating over the velocity and adding the initial position of the flare. In order to calculate the velocity, the acceleration of the flare was integrated and added to the initial velocity of the flare. The acceleration of the flare was calculated using Newton's second law: dividing the total force acting on the flare by the mass of the flare. In this model, the gravity, drag and dispense forces were included in this calculation.

It was decided to use a causal equation-based CT model of the flares, so to specify the causality of the constraints a parametric diagram was created — this can be seen in Figure 10.7.

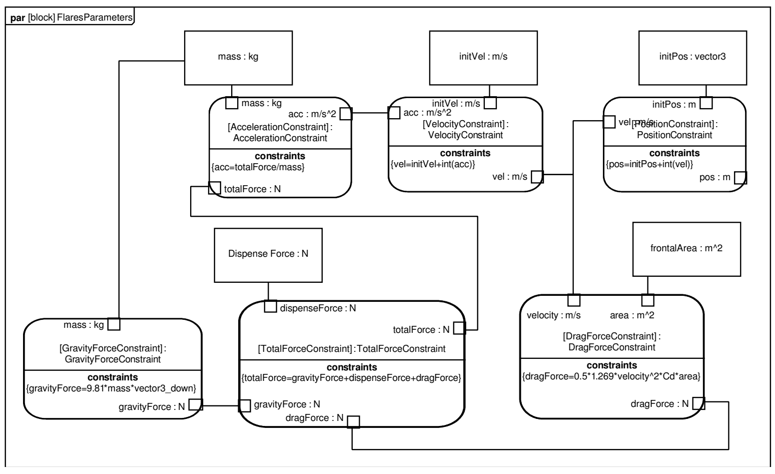

par [block] FlaresParameters
mass : kg
initVel : m/s
initPos : vector3
mass : kg
acc : m/s^2
[AccelerationConstraint] : AccelerationConstraint
constraints
{acc=totalForce/mass}
totalForce : N
initVel : m/s
acc : m/s^2
[VelocityConstraint] : VelocityConstraint
constraints
{vel=initVel+int(acc)}
vel : m/s
initPos : m
vel : m/s
[PositionConstraint] : PositionConstraint
constraints
{pos=initPos+int(vel)}
pos : m
Dispense Force : N
frontalArea : m^2
mass : kg
[GravityForceConstraint] : GravityForceConstraint
constraints
{gravityForce=9.81*mass*vector3_down}
gravityForce : N
dispenseForce : N
totalForce : N
[TotalForceConstraint] : TotalForceConstraint
constraints
{totalForce=gravityForce+dispenseForce+dragForce}
gravityForce : N
dragForce : N
velocity : m/s
area : m^2
[DragForceConstraint] : DragForceConstraint
constraints
{dragForce=0.5*1.269*velocity^2*Cd*area}
dragForce : N

Figure 10.7 Parametric diagram specifying the use and interrelation of flare parameter constraints.

Each of the six constraint blocks from Figure 10.6 have been added, and the individual parameters have been connected. The diagram clearly shows the causal dependencies of the flare model, and precisely specify the differential equations to be continuously evaluated in the CT model.

If even more detail should be added to the model (change the mass of the flare over time) this can easily be added as well. A new constraint block must be added to the diagram in Figure 10.6 describing the physical constraints on the mass, and the calculated mass can then be connected to the `AccelerationConstraint` and `GravityForceConstraint` constraints in Figure 10.7.

10.4.2 DE Constructs

The only elements of the co-model to be modelled in the DE domain was the missile guidance controller. In order to add a more detailed specification of the structure of this software controller a UML class diagram was created (Guideline multi_7) which can be seen in Figure 10.8.

Interface classes were added towards the IMU, IR sensor and steering blocks of the missile. These interfaces contain attributes for each of the ports towards these blocks as specified in the IBD in Figure 10.5. In addition *get* and *set* operations were added, so the controller can access the monitored

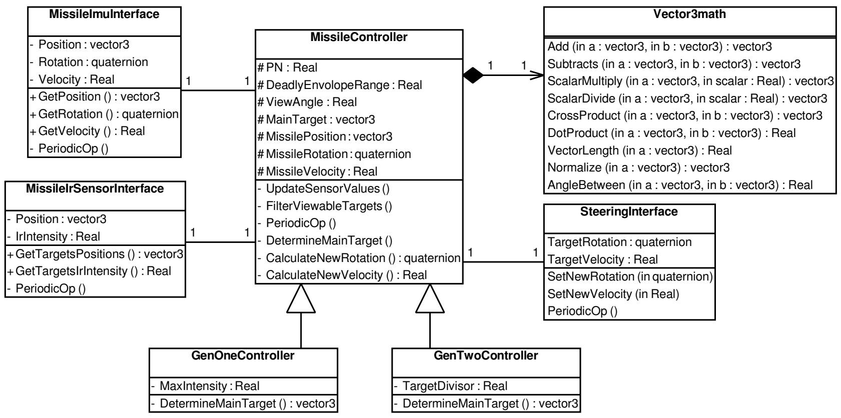

MissileImuInterface
- Position : vector3
- Rotation : quaternion
- Velocity : Real
+ GetPosition () : vector3
+ GetRotation () : quaternion
+ GetVelocity () : Real
- PeriodicOp ()
MissileController
# PN : Real
# DeadlyEnvolopeRange : Real
# ViewAngle : Real
# MainTarget : vector3
# MissilePosition : vector3
# MissileRotation : quaternion
# MissileVelocity : Real
- UpdateSensorValues ()
- FilterViewableTargets ()
- PeriodicOp ()
- DetermineMainTarget ()
- CalculateNewRotation () : quaternion
- CalculateNewVelocity () : Real
Vector3math
Add (in a : vector3, in b : vector3) : vector3
Subtracts (in a : vector3, in b : vector3) : vector3
ScalarMultiply (in a : vector3, in scalar : Real) : vector3
ScalarDivide (in a : vector3, in scalar : Real) : vector3
CrossProduct (in a : vector3, in b : vector3) : vector3
DotProduct (in a : vector3, in b : vector3) : Real
VectorLength (in a : vector3) : Real
Normalize (in a : vector3) : vector3
AngleBetween (in a : vector3, in b : vector3) : Real
MissileIrSensorInterface
- Position : vector3
- IrIntensity : Real
+ GetTargetsPositions () : vector3
+ GetTargetsIrIntensity () : Real
- PeriodicOp ()
SteeringInterface
TargetRotation : quaternion
TargetVelocity : Real
SetNewRotation (in quaternion)
SetNewVelocity (in Real)
PeriodicOp ()
GenOneController
- MaxIntensity : Real
- DetermineMainTarget () : vector3
GenTwoController
- TargetDivisor : Real
- DetermineMainTarget () : vector3
1
1
1
1
1
1
1
1

Figure 10.8 UML class diagram specifying the object-oriented structure of the missile controller.

variables and set the controlled variables. A 3D vector math helper-class was also added, which is used by the target seeking and guidance of the controller.

The missile controller was created as a superclass with subclasses for each of the two generation guidance controllers which needed to be modelled. The behaviour of the missile controller class was described in a separate sequence diagram (Guideline multi_8) which can be seen in Figure 10.9.

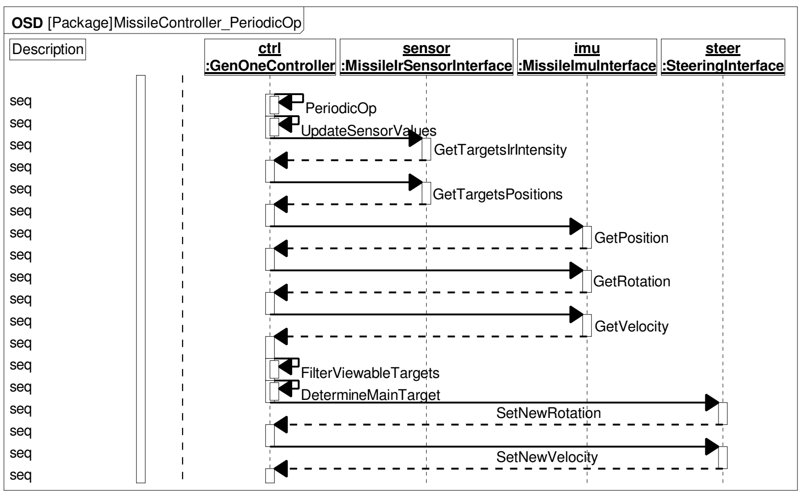


Figure 10.9 Sequence diagram specifying the behaviour and inter-object communication of the main loop in the missile controller.

In the periodically invoked `PeriodicOp` operation of the missile controller, all sensor values are initially updated in the `UpdateSensorValues` operation. Following this, the controller filters the targets within its field of view and from this set of possible targets determines the main target to steer towards. Finally, the steering commands are sent via the steering interface to the missile airframe block.

10.4.3 Co-Simulation Contract

The initial co-simulation contract was derived directly from the IBD of the missile from Figure 10.5 (Guideline multi_9) — a closeup of the missile guidance controller can be seen in Figure 10.10.

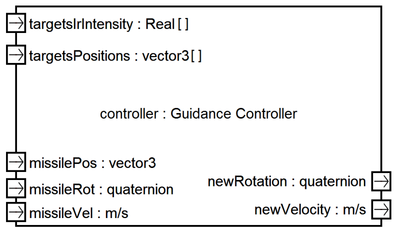


Figure 10.10 The missile guidance controller used for defining the co-simulation contract.

The DE controller must *monitor* the position, rotation and velocity of the CT missile airframe, as well as the IR intensity and position of the helicopter airframe and all deployed flares. The controller must also be able to *control* the desired rotation and velocity of the missile airframe in order to steer the missile towards the calculated main target.

In the initial version of the collaborative model, all the blocks in the helicopter IBD were modelled in the CT domain. At a later iterations of the co-model development the model of ECAP from Chapter 6 would be added as a DE component. The interface between ECAP and MWS, IMU, and DSS must then be added to the current co-simulation contract.

10.5 System Modelling

One of the main goals of the co-model was to precisely model the trajectory of the flares after they were dispensed from the helicopter. Modelling the rigid body dynamics of the flares was done in the CT domain in order to obtain an adequate level of fidelity, so it was chosen to use a CT-first approach in the first iteration of model development (Guideline multi_10). A submodel of the flare dynamics of a single flare was created to enable isolated analysis of this part of the model (Guideline multi_16).

Initially, the mass and radius of the flare were kept constant through a co-simulation run to simplify the model (Guideline multi_13). From analysing the constraint equations from the parametric diagram in Figure 10.7 the drag coefficient of the flare was deemed significant since it directly affects the drag force acting upon the flare. Another significant parameter is the dispense force acting on the flare when it is dispensed from the helicopter. These two parameters were added to the co-simulation contract as shared design parameters enabling parameter sweeps using the Automated Co-model Analysis functionality of the DESTECS tool (Guideline multi_15).

Once the flare submodel was analysed with satisfactory results, the initial co-model was created in the second iteration of model development. The main blocks (helicopter, missile and flares) were created with all of their internal child-blocks (Guideline multi_11). The interface of the children as defined in the IBD in Figure 10.4 and Figure 10.5 were also added. The top-level view of the co-model inside the 20-sim editor can be seen in Figure 10.11.

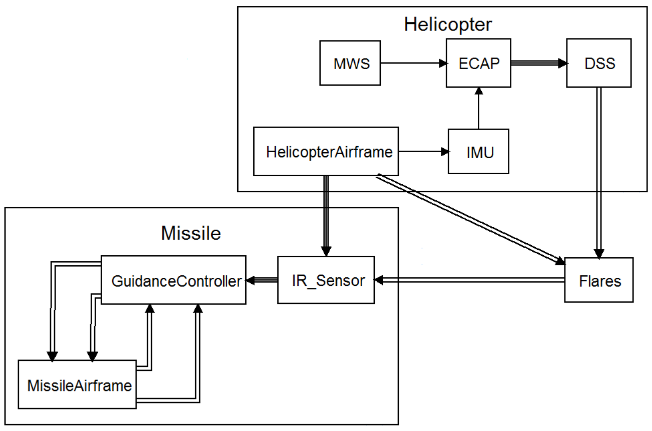


Figure 10.11 Top level view of the DESTECS continuous-time model shown inside the 20-sim editor. The arrows indicate the causality of the submodels.

Figure 10.11 shows that a deliberate abstraction was made — the missile warner sensor (MWS) of the helicopter is not associated with the missile, meaning it cannot read the position and velocity of the missile. This was done in order to reduce the complexity of the countermeasure system (ECAP) onboard the helicopter in the initial version of the co-model. Instead, the MWS issues a static missile warning message at a given point in time, and ECAP reacts to this.

Only the `GuidanceController` of the missile was modelled in VDM. The object-oriented structure of the controller and utility classes had already been specified in a UML class diagram in Figure 10.8 and the behaviour of the main loop of the controller in the sequence diagram in Figure 10.9. These diagrams formed the basis of the initial DE model. As an example of the DE controller, the `FilterViewableTargets` operation of the `Missile-Controller` class mentioned in Section 10.4.2 is shown below:

```
class MissileController is subclass of Types

values
  VIEW_ANGLE : real = 0.0175;  -- one degree in radians

instance variables
  targetsPos : seq of Vector3D;
  targetsInt : seq of real;
  missilePos : Vector3D;
  missileVel : Vector3D;
  viewableTargets : seq of Vector3D;

operations

private FilterViewableTargets : () ==> ()
FilterViewableTargets () ==
 (viewableTargets := [];  -- Reset sequence

  for all i in set inds targetsPos
  do
   (dcl toTarget : Vector3D := VectorMath`Subtract(targetsPos(i),
                                                   missilePos),
        angle : real := VectorMath`AngleBetween(missileVel, toTarget);
    if(angle) < VIEW_ANGLE)
    then viewableTargets := viewableTargets ^ [targetsPos(i)];
   );
 );

end MissileController
```

The `MissileController` generates a 3D vector from the missile to the target for each target in the electronic warfare scenario. The angle between this vector and the velocity vector of the missile is calculated in the `AngleBetween` operation of the `VectorMath` helper-class. The angle is determined by calculating the inverse cosine of the dot product of the two vectors. Since the diagonal field-of-view of the missile is 2 degrees, if the angle is below one degree (0.0175 radians) the target is added to the `viewableTargets` instance variable. All viewable targets are evaluated in the `DetermineMainTarget` operation of the `MissileController` class.

10.6 System Analysis

For the initial modelling of the flare submodel, visual validation (Guideline multi_19) was used. A simple 3D object of a spherical flare was dispensed from a representation of the helicopter. The helicopter was moved and rotated to resemble the 6 degrees of freedom of a helicopter. Visual validation ensured that the flare inherited the velocity and rotation of the helicopter, so if the helicopter was in a roll, the flare was dispensed accordingly.

The real system was not easily accessible for measurement, but Guideline multi_20 was still followed. Comparing the trajectory of the flares in the 3D animation with high-speed camera footage of real flares being dispensed gave confidence in the fidelity of the model of the flares. The force with which the flares were dispensed as well as the drag coefficient of the flares were tweaked until the modelled trajectory resembled the real footage. Making these parameters more realistic was made easier by using the Automated Co-model Analysis for sweeping these two shared design parameters.

Visual validation was also used later in the modelling process after several iterations of the system modelling phase. A much more elaborate 3D environment was built inside the 20-sim 3D animator. A camera was placed in the nose of the missile, enabling a missile point-of-view in order to validate the guidance controller. Another camera was placed close to the helicopter giving a good view of the dispensed flare patterns. Screenshots from these cameras can be seen in Figure 10.12.

As mentioned, the high fidelity model of ECAP described in Chapter 6, could be added as a DE model in a future addition to the co-model presented in this chapter. As described in Chapter 6, three different domain-specific analysis techniques were used to validate this DE model of ECAP (Guideline multi_18) — see Section 6.3.5 for details.

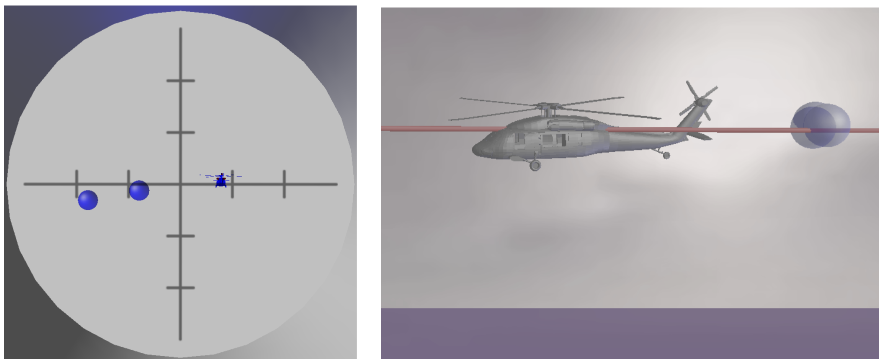


Figure 10.12 Screenshots from the 3D simulation plot from inside the 20-sim editor. On the left the scenario is seen from the missile point of view and on the right the helicopter is seen from the side.

10.7 Case Study Results

The main purpose of the electronic warfare case study was to analyse how effectively flares patterns are as countermeasures against different generations of IR guided missiles. The missile guidance controller of the co-model could use either first or second generation guidance strategies. Co-simulation of the model showed that existing countermeasure strategies work very robustly against these older IR guidance systems. The co-model enabled analysis of how long the flares stayed inside the point-of-view of the missile, and hence were effective as countermeasures. By changing the dispense angles of the flares it was possible to analyse if this could be optimised — this is a very concrete example of effective model-based design space exploration.

A proposed countermeasure strategy against fourth generation IR guidance systems was to use flare patterns to generate a well resembling IR image of the helicopter. As described in Section 10.4.1 detailed information regarding the most recent generations of IR guidance is not publicly available, and hence was not included in the co-model to avoid reducing the accuracy of the model. Instead, visual validation of the countermeasure strategy was used. After analysing multiple co-simulations it was concluded that it was not possible to create an IR image resembling a helicopter using flares, and hence another countermeasure strategy was needed.

As described in Section 2.7 modern missiles make use of *electronic protective measures* (EPM) to reduce the effect of countermeasures deployed by the target. One such EPM used by modern missiles is to analyse the movement of all IR sources. If a new IR source moves in a radical direction compared to the current target the missile identifies this as being a flare dispensed from the target, and hence it is not considered as a possible target. In other words: for flares to be the most effective they must follow the movement of the helicopter and only slowly leading the missile away. The co-model enabled analysis of the *rate of separation* of the flares from the helicopter seen from the point-of-view of the missile. In Figure 10.13 below, the helicopter flies in a positive x-direction and dispenses a flare directly to its right in a negative y-direction. Missiles are placed all around the helicopter to monitor the rate of separation of the flare seen from different angles.

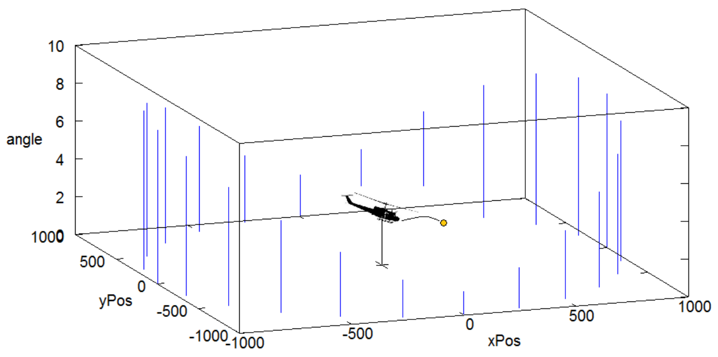


Figure 10.13 Experiment analysing the rate of separation of flares dispensed from a helicopter.

Countermeasures can be a combination of flare patterns and helicopter maneuvers. The results of the co-simulation can be used to optimise countermeasures: changing the heading of the helicopter slightly before dispensing flares can result in lowering the rate of separation seen from the point-of-view of the missile. By having a system-level overview of the entire encounter, it is possible to optimise the performance of the countermeasure system.

The electronic warfare case study has shown the industrial partner that there are completely new approaches to system development. Models enable the exploration of a huge design space and analysis of key system parameters. In addition to the added analytical power offered by models, the 3D animations have been a great aid in communicating complex systems operating in 3D space to non-technical stakeholders and customers. Recently, a variant of the co-model was used in the education of Norwegian fighter pilots. Using a 3D model to explain countermeasure concepts and ECAP functionality proved invaluable and the feedback from the pilots was very positive.

10.8 Additional Evaluation of the Guidelines

An M.Sc. project [106] has been conducted to further evaluate the usefulness of the guidelines. The electronic warfare case study presented in this chapter was also modelled in the game engine Unity described in Section 7.3.8. The guidelines were followed to determine if they are applicable to tools other than DESTECS.

10.8.1 M.Sc. Project Evaluating the Guidelines

In the M.Sc. project a software engineering student (representing the DE expert) and a robotics engineering student (representing the CT expert) developed a co-model of an autonomous robot by following the methodological guidelines. The two engineers followed the guidelines through all four phases: model purpose, system decomposition and the first iteration of system modelling and analysis. This ensured that all guidelines were applied in a collaborative process between two engineers with different backgrounds.

The use of SysML as a common notation for decomposing the system was well received by both project participants. Defining the interface adding both type/unit and direction helped clarify the intended use of the individual domain-specific blocks, and helped removing potential cross-disciplinary errors. The importance of sensor placement (position and rotation) was discussed during the process of defining the interfaces. As a result, the software engineer got a much deeper understanding of the physical properties he had to monitor and control in the DE model he was creating.

The use cases and actors identified during use case modelling, were used for constructing the BDD. The fact that a block of the BDD corresponds to an actor of the use case diagram contributes positively to the traceability of the system description.

The description of the DE constructs enabled easy transitioning from SysML to VDM, since both support object-oriented concepts, and almost every modelling construct of UML has a VDM equivalent. When transitioning from the SysML descriptions to 20-sim, the guidelines provided a natural way of structuring the resulting CT-model into submodels. In general these intermediate SysML descriptions enable a systematic approach for the construction of the two domain models.

The usefulness of the constraint and parametric diagrams was questioned, though. An attempt was made to describe constraints on the individual wheels as well as the rigid body of the robot using these types of diagrams. It was found that the causality of these elements was too strong, and the many interrelations made it infeasible to document using the suggested diagrams. Instead, informal sketches and mechanical drawings were used for defining the causality constraints of the robot. It was concluded, however, that for some physical systems, the benefits of using constraint and parametric diagrams was clearer — the flare dynamics described earlier in this chapter is a good example of this.

Capturing the essence of the methodology in the form of small statements or guidelines, allowed for easy reference throughout development. During the M.Sc. project the guidelines served well as input for the development work, since the enforcement of this kind of structure helps ensuring that important system aspects are dealt with in a timely manner.

10.8.2 Evaluation of Alternative Tool

To further evaluate the guidelines they were applied to the electronic warfare case study described in this chapter using an alternative tool: the Unity game engine. To evaluate the level of fidelity Unity could achieve, a small test was conducted. A spring-damper system was built in both 20-sim and Unity, and the position of a mass was monitored — see Figure 10.14.

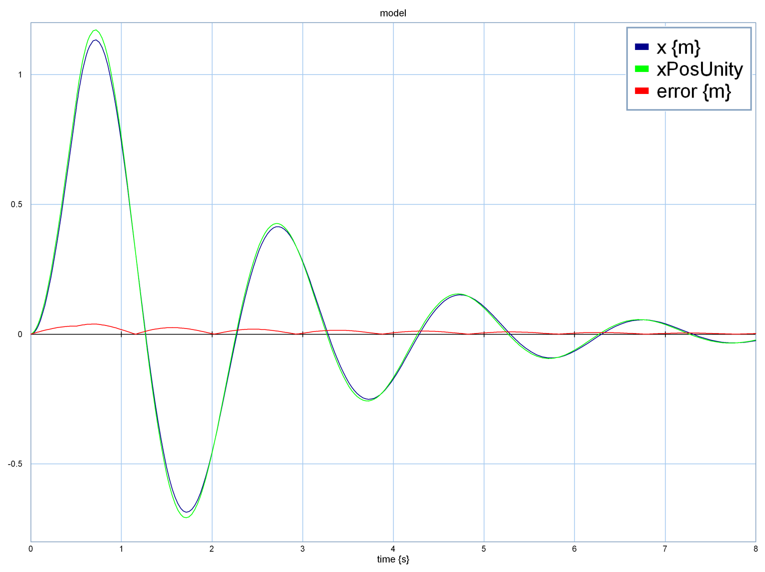
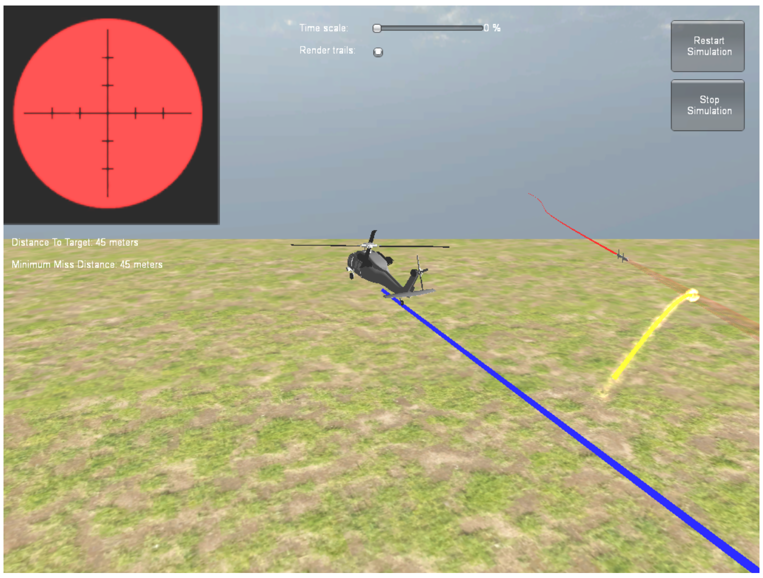


Figure 10.14 Comparing 20-sim with Unity monitoring the position of a mass in a spring-damper system.

The physics engine of Unity over-shoots by 3.5% compared to 20-sim. The important thing to note is that the error does not accumulate over time. This shows that PhysX is slightly inaccurate in the extremes, but over the course of the entire simulation the error goes towards zero.

The electronic warfare case study presented previously in this chapter was developed in Unity, resulting in a simulator called *Virtual Electronic Warfare Simulator* (ViEWS). A screenshot from ViEWS can be seen in Figure 10.15.



Distance To Target: 45 meters

Minimum Miss Distance: 45 meters

Figure 10.15 Screenshot from the ViEWS simulator.

In Unity, several cameras can be rendered simultaneously. In the upper-left corner of Figure 10.15, the view of the missile is shown — in the screenshot the missile is very close to the missile, so it only sees the flare. Figure 10.16 shows the 64-by-64 pixel resolution missile point-of-view from a greater distance.

In addition, a camera can be panned around the helicopter at run-time, to view the simulation from different angles. The simulation happens real-time, but it is possible to scale time down enabling better analysis of the simulation. This, along with the better rendering capabilities of Unity, makes ViEWS a good tool for showing the capabilities of the self-defense system to non-technical stakeholders.

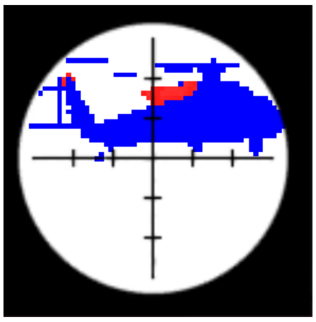


Figure 10.16 Screenshot from the ViEWS simulator, showing the missile point-of-view.

The guidelines supported the development of ViEWS with only minor alterations needed. The model purpose and system decomposition were reused, so all the guidelines were applicable. Since no co-simulation contract is needed in Unity, Guideline 9 and 15 were not applied. No automated tools exist for sweeping parameters, so Guideline 14 had to be done manually. Since 3D visualisation is such an integrated part of Unity, Guideline 19 was applied extensively.

Part IV

Evaluation, Discussion and Conclusion

11

Conclusion

This chapter concludes on the results achieved in this thesis. The objectives of the thesis defined in Chapter 1 are related to the modelling guidelines described in Chapters 4 and 9 and the combination of formal and agile methods described in Chapter 5. These methodological guidelines for creating models of heterogeneous embedded systems at various levels of fidelity, using either mono- or multi-disciplinary modelling approaches, comprise the results of this thesis.

11.1 Introduction

This thesis provides methodological guidelines for developing models of embedded systems of various levels of fidelity. The guidelines fit into one of two general approaches: modelling the entire system and environment dynamics using a single formalism, or creating separate models for the discrete-event (DE) controller and continuous-time (CT) dynamics of the environment, and analysing them collaboratively using co-simulation.

The purpose of this chapter is to evaluate the outcome of the thesis, and to assess to what extent the objectives have been met. Section 11.2 summarises the research contribution made. Section 11.3 evaluates to which extent the research contribution meets the objectives of the thesis defined in Chapter 1. Future work is identified and presented in Section 11.4, and an outlook of the use of modelling in the development of embedded systems is given in Section 11.5.

11.2 Research Contribution

Modelling Guidelines

The main contribution of the PhD thesis, is a collection of lightweight methodological guidelines supporting the modelling of heterogeneous embedded systems. Chapter 4 describes guidelines supporting a stepwise approach to VDM-RT model construction, where both the embedded controller and its environment are described using a single DE formalism. Adding concurrent behaviour to a sequential model, was a generally well known technique, but adding the real-time behaviour and distributed architecture was the real contribution of the work.

To add further detail to the model, the environment is modelled using a CT formalism which is combined with the DE model of the controller to create a collaborative model. The creation of co-models is supported by twenty methodological guidelines described in Chapter 9. These guidelines describe a pragmatic approach to: model purpose and requirements modelling, system decomposition, system modelling, and system analysis. Since co-modelling requires a more holistic system-level approach, SysML was chosen as a domain-independent notation supporting these phases of model construction. This ensures that conceptual discussions are not hindered by domain-specific notations, not understood by all of the disciplines involved.

The various guidelines have been applied to two case studies. In Chapter 6 a DE model of a countermeasure system is created. Only the first two phases of the stepwise process were applied, showing that the guidelines can be adapted to support the level of modelling fidelity needed. In Chapter 10 the model is extended with a CT model of the physical dynamics of the environment.

Tool Independent Guidelines

To ensure that the modelling guidelines support a certain degree of tool independence, two tools using different approaches to collaborative modelling are compared in Chapter 8. In addition, the electronic warfare case study presented in Chapter 10 was modelled using a real-time rigid body simulation approach. Applying subsets of the guidelines to case studies modelled in various simulation tools shows that with minor modifications the guidelines are applicable to a range of tools.

Agile Model Development

To support the project management aspect of using a modelling approach to embedded systems development, the integration of formal modelling into agile methods has been researched. Chapter 5 discussed to which extent formal and agile methods can be combined and benefit from each other. It is concluded that combining these seemingly orthogonal approaches should not be done lightly, without proper methodological guidelines.

A concrete example of such guidelines is also provided in Chapter 5 that illustrates the combination of formal methods and the agile project management framework Scrum. The roles of the project participants are described as are the activities and artifacts produced throughout the project.

The agile approach to model construction has been applied to the ECAP case study in Chapter 6 and to a lesser extent to the electronic warfare case study presented in Chapter 10.

11.3 Evaluation of the Guidelines

In this section, we evaluate to which extent the guidelines described in Chapters 4 and 9 provide for the properties listed in Section 1.6. It is challenging to evaluate the guidelines in isolation though, because a modelling approach consists of several things: the capabilities of the available tools, the methods and guidelines describing how to exploit the possibilities provided by those tools, and the experience with which the guidelines are applied. This section evaluates the combined modelling approach, thereby indirectly evaluating the guidelines. When individual guidelines have helped directly in achieving one of the properties, these will be mentioned.

The limited number of case studies means that no statistical significance can be attached to the results. Further, these are not controlled trials like those carried out in the ConForm project mentioned in Section 3.3.4. However, based on the experiences gained in the context of this Industrial PhD thesis, we conclude the following.

11.3.1 Prediction of System Behaviour

In both case studies it was possible to predict system behaviour before the systems were fully developed. In one case the feasibility of a change in the design was validated using a DE model, and in the other case a novel idea to countering state-of-the-art missiles was abandoned based on the simulation of the co-model.

The mono-disciplinary ECAP case study presented in Chapter 6 validated the redesign of the new interpretation of messages passed between ECAP and AS. Using the model it was possible to predict system behaviour prior to software implementation and before test rigs were available.

The multi-disciplinary electronic warfare case study presented in Chapter 10 created visual representations of the resulting IR image created by a combination of dispensed flares and helicopter maneuvers. Initially, this model was created to predict how effective different countermeasures were against missiles using early generation IR-based target seeking. It was possible to analyse which dispense angles ensured that the flares stayed in the field-of-view of the missile the longest, and hence were the most effective. The model also enabled the analysis of the precision of which IR images could be generated using readily available flare technologies. If it would be possible to generate an IR image resembling a helicopter, even fourth generation missiles could be countered effectively. Using the model, it was concluded that it was not possible to generate a well resembling IR image of the helicopter using flares, and hence another strategy is needed to counter missiles using state-of-the-art IR target seeking.

In the M.Sc. project described in Section 10.8.1, a co-model of an autonomous robot was created. Following guideline multi_20 the fidelity of this co-model was increased by calibrating the model parameters until the co-model performed similar to a physical prototype that was also developed. Calibration was done iteratively until the co-model performed within an error margin of 15% of the physical prototype. The co-model was then used to predict the duration of course completion of several other tracks. The predictions were within the same error margin as the initial test. We believe that by further increasing the fidelity of the co-model, the precision of these predictions would increase as well.

11.3.2 Cross-Disciplinary Collaboration

The case studies show that the guidelines and modelling approach strengthen collaboration across technical boundaries, but also between technical project participants and non-technical stakeholders.

Several initiatives were made to ensure better collaboration with the customer and systems engineers in the ECAP case study presented in Chapter 6. The model was created in such a way that the output generated by the model when evaluating different scenarios was directly readable by the customer. In addition, all state changes of ECAP and the AS sub-system were printed in a format that the customer could read and understand without further translation needed. This ensured that the results of the scenarios could be sent unaltered to the customer. This was a key ingredient in ensuring collaboration with the non-technical stakeholders of the project.

Making extensive use of 3D animations in the electronic warfare case study presented in Chapter 10 helped strengthen communication with non-technical stakeholders. The visual feedback of changes in the model enabled electronic warfare domain experts to provide input to the model. Having a graphical interface to the model, helps demystifying the model and enables a wider range of stakeholders to actively use it.

The use of SysML in the M.Sc. project described in Section 10.8.1 enhanced the communication across the disciplinary boundaries of the software and robotics engineering students participating in the project. Discussing the importance of sensor placement and orientation was augmented by using the high-level, discipline-neutral, yet standardised modelling language SysML.

11.3.3 Understanding System-Level Impact of Design Decisions

The case studies presented in this thesis show that even abstract models, where only parts of the entire system are modelled, can be used to analyse system-level implications of design decisions.

When making small local updates to a large system it is important to ensure the absence of system-level side-effects. In the ECAP case study in Chapter 6, only ECAP, AS and the communication between the two systems were modelled. In addition, the model was created in such a way that ECAP could be fed threat messages from other sensor-systems. This enables the analysis of system-level properties like threat priority, ensuring that ECAP still treated threat messages with different priorities appropriately.

The co-model created in the electronic warfare case study presented in Chapter 10 included parameters like: timing of the dispense of individual flares, dispense angles used, orientation of the helicopter, and rigid body physics of the flares. By including all these different types of elements it was possible to analyse the system-level impact of changes to the individual parameters. As an example, it was tested if the timing of the dispensing of flares could be changed dynamically based on the orientation of the helicopter to better control the IR image created. The conclusion was that while the timing had an impact, it was not enough to fully control the resulting IR image.

11.3.4 Industry-Ready

The guidelines advocate the use of an iterative approach to model development which eased the integration with the existing processes used by the industrial partner. Chapter 6 acts as empirical evidence of the benefits a modelling approach can bring to embedded systems development.

Since agile principles are used by the industrial partner in the daily development of systems, a combination of modelling and agile system development was made. Presenting the use of formal modelling incorporated into well-known development methods increases the likelihood of wider industrial acceptance. For the same reason, the collaborative modelling guidelines presented in Chapter 9 are put into context of the well-known spiral model.

To show examples of the benefits modelling can bring, different case studies within the area of electronic warfare have been conducted showing the applicability of the methods and guidelines in an industrial setting. These case studies have largely been a success: the model of the ECAP system presented in Chapter 6 helped solving a month long dispute with a customer, clearly convincing him of the feasibility of a proposed communication protocol design change.

The success of the ECAP case study presented in Chapter 6 has shown the industrial partner some of the benefits of a modelling approach to systems development. The electronic warfare case study from Chapter 10 has generated some interest at Terma — not only in the analytical power of co-modelling, but also as a sales tool explaining potential costumers how the system operates. Having a model with a graphical interface has been a great aid in describing complex system behaviour to customers.

The industrial partner is planning to run a follow-up project as a continuation of the industrial PhD presented in this thesis. In this project a model of a surveillance system for perimeter protection of airports, power plants and similar critical parts of the infrastructure will be developed. The model will be used to analyse the effect of different sensor data correlation algorithms, supporting the design of the real system.

11.4 Future Work

In this section, some possible directions for future work are given. Just like the guidelines in Chapter 9, the suggested future work mainly accompanies the DESTECS tool.

11.4.1 Process for Modelling Embedded Systems

Instead of having separate processes for developing mono- and multi-disciplinary models of embedded systems, a process combining the mono- and multi-disciplinary guidelines is suggested here. An outline of the process can be seen in Figure 11.1. The process is based on the phases defined in the spiral process described in Section 9.3.

Having a combined process will make it clearer when the individual guidelines should be applied, and how the mono- and multi-disciplinary guidelines can coexist. The *system boundary definition* phase of the mono-disciplinary model development process is supported by only a single guideline. In the combined process described below the analogue phases are supported by nine guidelines which will ensure a much better structure of the mono- as well as the multi-disciplinary models produced by following this process.

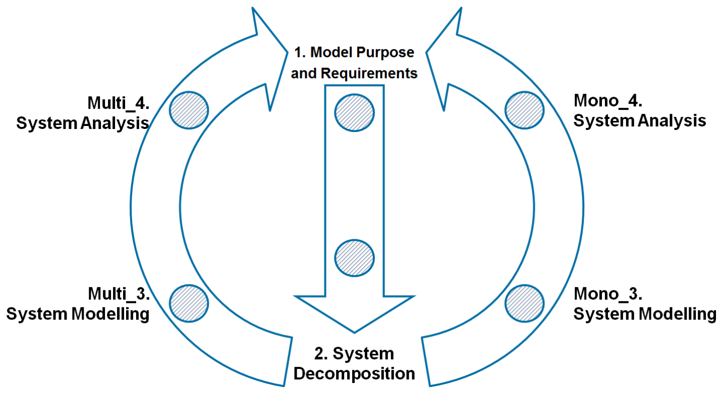


Figure 11.1 Outline of process combining the mono- and multi-disciplinary guidelines.

The phases of the process as well as the use of the guidelines is described in the following:

1. Model Purpose and Requirements

In this phase Guidelines Multi_1 to Multi_3 as well as Mono_1 should be applied. The guidelines stress the importance of documenting the assumptions made, the purpose of the model, and the requirements that must be met.

2. System Decomposition

In this phase Guidelines Multi_4 to Multi_9 should be applied. The results of this phase are:

- specification of the main system parts (SysML BDD);
- interface specification (SysML IBDs);

- specification of the dynamics of physical parts of the system and causality between these parts (SysML constraint and parametric diagrams);
- architecture specification of the embedded controller (UML class diagrams);
- behaviour of the embedded controller (UML activity, sequence or state charts); and
- description of interface between software controller and physical parts of the system.

The system decomposition is an enabler for choosing the correct modelling approach. Following Guideline Multi_10, either a CT-first (outside the scope of this thesis), DE-first (right branch of Figure 11.1), or contract-first (left branch of Figure 11.1) modelling approach is chosen.

Mono_3. System Modelling
In this phase Guidelines Mono_2 to Mono_14 should be applied. Each of the three tiers of models (sequential VDM++, concurrent VDM++, and VDM-RT) should be developed in separate iterations.

Mono_4. System Analysis
In this phase Guidelines Mono_15 and Mono_16 should be applied. For each iteration of model analysis there are three possible outcome:

1. A competent model has been created, and the desired analysis have been carried out. There is no need to continue modelling.
2. A more accurate model is needed. It is still believed that a competent model can be developed using a mono-disciplinary approach, so the next tier model is planned for the following iteration.
3. A more accurate model is needed. In order to achieve this, a more accurate model of the dynamics of the physical parts of the embedded system is needed, so a multi-disciplinary model is planned for the next iteration.

Multi_3. System Modelling
In this phase Guidelines Multi_11 to Multi_17 should be applied. In each iteration details will be added to the different parts of the model.

Multi_4. System Analysis
In this phase Guidelines Multi_18 to Multi_20 should be applied. For each iteration of model analysis there are two possible outcome:

1. A competent model has been created, and the desired analysis have been carried out. There is no need to continue modelling.
2. A more accurate model is needed. Details to be added in the following iteration are planned.

Due to its iterative nature, the process fits into an agile setting as described in Chapter 5. In that case, each iteration of the process must fit into a single Scrum sprint.

The existing case studies must be reevaluated using the suggested combined process: The ECAP case study presented in Chapter 6 should follow the right branch of Figure 11.1 for several iterations until a VDM-RT model of the ECAP system has been developed. Once this is achieved, the continuous dynamics of the system can be added following the left branch of Figure 11.1 resulting in a co-model of the entire electronic warfare case study presented in Chapter 10.

11.4.2 Modelling of Multiple Embedded Systems

The tools supporting collaborative modelling of multi-disciplinary systems are mainly focused on developing models of a single system. An important issue, not yet addressed, is how to model several systems in such a way that they are separate but can still interact. In the electronic warfare case study presented in Chapter 10, the co-model should ideally be separated into individual models of the DE controllers and CT dynamics of the missile, helicopter and flares, all sharing a common environment model. Since this is currently not possible in the DESTECS tool, the dynamics of all three systems were described in a single 20-sim CT model, while both the missile guidance controller and ECAP were created in a single DE model. Given the tools supporting co-simulation of more than a single DE and a single CT model would also require an update to the guidelines presented in Chapter 9. As with the currently available tools, the guidelines mainly support system decomposition of a single system, and distribution of system elements to a single co-model. Additional guidelines are needed to guide the creation of multi-disciplinary models with multi-party integration, where a single co-models must be decomposed and distributed to multiple CT/DE models.

11.4.3 Tool Extension

It was carefully considered how the SysML constructs would map to the corresponding parts of the co-model. Section 10.4.1 describes how the SysML internal blocks map to submodels in 20-sim and how the SysML atomic ports and bi-directional ports map to 20-sim signals and power ports respectively. The Overture tool contains a UML transformation tool [128] that imports UML class diagrams in XMI format and create the class skeleton in VDM. The translation from the SysML diagrams to the initial co-model structure is currently done manually, but given the mapping rules described above it would be possible to automate parts of the process similar to the UML transformation already supported.

11.4.4 More Elaborate Case Studies

The importance of industrial acceptance and adoption of the modelling techniques described has been emphasised in this thesis. Various case studies have been conducted, all with the author as one of only a few people involved (if not the only one). Conducting a case study with multiple groups of engineers with different backgrounds, following the guidelines described in this work, would help evaluating the usefulness of the guidelines. The M.Sc. project described in Section 10.8.1 was initiated to ensure that other people than the author had followed the guidelines, resulting in a more objective evaluation.

11.5 Outlook

In academia, there is a great interest in moving from expensive and time consuming testing of physical prototypes to large scale design space exploration of computerised models. Industry is equally interested, but few companies are willing to accept the risk involved in changing the way embedded systems are developed. Projects, like the one presented in this thesis, where industry and academia collaborate on a government supported project, are important to develop mature methods and tools that support modelling of embedded systems. An existing compendium of case studies where the method was successfully applied will help industry trust in the technology. This will in turn increase the likelihood of more companies moving towards a modelling approach to embedded systems development.

Part V

Appendices

A

Glossary

Advanced Sensor (AS): A subsystem which contains the combined functionality of a MWS and ECAP.
(p. 68)

Anti-lock Braking Systems (ABS): An automobile safety system that allows the wheels to maintain tractive contact with the road while braking preventing the wheels from locking up (ceasing rotation) and avoiding uncontrolled skidding.
(p. 4)

Automated Co-model Analysis (ACA): Part of the DESTECS toolchain. Enables parameter sweeps on multiple parameters from the entire co-model.
(p. 94)

Block Definition Diagram (BDD): SysML diagram type used for defining the main entities of a system and their relations.
(p. 120)

Chaff: Dispensable decoy used as ECM against radar-guided threats. When dispensed, chaff creates a cloud reflecting radar signals.
(p. 22)

Co-model: Short for collaborative model. Consists of two component models (one DE and one CT) and a contract used to link the two models together.
(p. 88)

Collaborative modelling: The development of a co-model
(p. 87)

Co-simulation: The simulation of a co-model.
(p. 7, p. 89)

Co-simulation engine: Supervising the co-simulation, and managed the synchronisation of time and shared variable values of a co-model.
(p. 89)

Competent model: A model is competent for a given analysis if it contains sufficient detail to permit that analysis, and the results are truthful with respect to reality.
(p. 6)

Continuous-Ttime (CT): A type of formalism where a physical system is represented as a set of equations (typically involving differential equations) that are continuously evaluated.
(p. 7)

DESTECS Command Language (DCL): A scripting language –part of the DESTECS toolchain– used to simulate user input and activate faults.
(p. 93)

Design parameter: A property of a model that affects its behaviour, but which remains constant during a given simulation.
(p. 88)

Design Space Exploration (DSE): The task of exploring alternative designs in order to find the optimal solution.
(p. 27)

Digital Sequencer Switch (DSS): A subsystem that administers the correctly timed dispensing of chaff or flares.
(p. 68)

Discrete-Eevent (DE): A type of formalism where the system is represented as a chronological sequence of events, which happen in discrete points in time.
(p. 7)

Electronic Combat Adaptive Processor (ECAP): A programmable system that calculates the optimal countermeasure to any given threat scenario.
(p. 67)

Electronic Countermeasure (ECM): Used to suppress the enemy use of ESM. Examples are flares that are operating in the infrared frequency band.
(p. 15)

Electronic Protective Measures (EPM): Used to counter the enemy use of ECM, hence also called Electronic counter-countermeasures.
(p. 15)

Electronic Support Measures (ESM): Used to gain knowledge of the adversary using sensors operating in the electromagnetic spectrum.
(p. 15)

Electronic Warfare (EW): Military forces taking actions to dominate the electromagnetic spectrum, to find, track, target, engange and assess the adversary, while denying that adversary the same ability.
(p. 15)

Embedded system: A single-purpose computing system characterised by being integral to and wholly encapsulated by the system it controls.
(p. 3)

Event: The changing of the logical value of a predicate. Defined in a co-model contract.
(p. 89)

Flare: Dispensable decoy used as ECM against IR-guided threats. When dispensed, flares radiate energy in the IR frequency band.
(p. 23)

Graphical User Interface (GUI): A type of user interface that allows users to interact with the system using images rather than text commands.
(p. 50)

Integrated Development Environment (IDE): A software application that provides facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools and a debugger.
(p. 111)

Internal Block Diagram (IBD): SysML diagram type used for defining the internal structure of a block.
(p. 120)

Inertial Measurement Unit (IMU): A device that measures and reports on the velocity, orientation, and gravitational forces imposed on a vehicle.
(p. 74, p. 136)

Infrared (IR): Electromagnetic radiation with longer wavelengths than those of visible light. This range of wavelengths includes most of the thermal radiation emitted by objects near room temperature.
(p. 15)

Launch configuration: Defines the entry point of a co-simulation, sets total simulation time, sets shared design parameter values, and other co-simulation settings.
(p. 93)

Model: A more or less abstract representation of a system or component.
(p. 6, p. 88)

Modelling: The process of creating a model.
(p. 6)

Model of Computation (MoC): The definition of the set of allowable operations used in computation of a certain model. Examples are: DE, CT, and synchronous data flow.
(p. 98)

Missile Warner System (MWS): Sensor system monitoring the IR frequency band of the surroundings to detect the exhaust plume of missile being fired towards it.
(p. 21)

Ordinary Differential Equation (ODE): An equation containing a function of one independent variable and its derivatives. Used to describe dynamic physical behaviour.
(p. 90)

Parameter sweep: A functionality of the tool 20-sim where one or more parameters are given a range of values, and simulations using all permutations of parameter values are automatically performed.
(p. 91)

Personal Computer (PC): A general-purpose computer is intended to be operated directly by the end-user with no intervening computer operator.
(p. 4)

Quaternion: A four-dimensional structure used to describe rotations in three-dimensional space.
(p. 24)

Radar: Short for RAdio Detection And Ranging. A radar system continuously transmit waves of short bursts of electromagnetic energy that bounces off objects. Parts of the transmitted signal is returned to the radar, and is used to determine the range and direction of the object.
(p. 16)

Radar-Cross-Section (RCS): A measure of how detectable an object is with a radar. The RCS value of an object can rely on the angle of which the object is being observed.
(p. 17)

Radar Warning Receiver (RWR): Sensor system used to warn of incoming radar guided threats by detecting the transmitted radar signal.
(p. 20)

Radio Frequency (RF): The frequency band of the electromagnetic spectrum in which radars operate.
(p. 16)

Scrum: An agile project management method, which focusses on short feedback loops in system development. Scrum is a structured method, where roles, responsibilities, activities and artifacts are defined.
(p. 13, p. 60)

Shared variable: Defined in a co-model contract. A variable that appears in and can be accessed from both component models.
(p. 89)

Simulation: A test run of a model. The model receives input, and the generated output is compared to the expected results.
(p. 6, p. 88)

Test-Driven Development (TDD): An agile method, where initially a test case is described, followed by the creation of the minimum required code to satisfy the test case.
(p. 36)

Unified Modelling Language (UML): A standardized general-purpose modeling language in the field of object-oriented software engineering. It is managed by the Object Management Group.
(p. 34)

Vienna Development Method (VDM): A collection of formal techniques to specify and develop software. It consists of a formally defined language, as well as strategies for abstract descriptions of software systems.
(p. 28, p. 30)

B

Overview of VDM Operators

The Boolean Type

Operator	Name	Signature
not b	Negation	**bool** → **bool**
a **and** b	Conjunction	**bool** * **bool** → **bool**
a **or** b	Disjunction	**bool** * **bool** → **bool**
a => b	Implication	**bool** * **bool** → **bool**
a <=> b	Biimplication	**bool** * **bool** → **bool**
a = b	Equality	**bool** * **bool** → **bool**
a <> b	Inequality	**bool** * **bool** → **bool**

The Numeric Types

Operator	Name	Signature
-x	Unary minus	**real** → **real**
abs x	Absolute value	**real** → **real**
floor x	Floor	**real** → **int**
x + y	Sum	**real** * **real** → **real**
x - y	Difference	**real** * **real** → **real**
x * y	Product	**real** * **real** → **real**
x / y	Division	**real** * **real** → **real**
x **div** y	Integer division	**int** * **int** → **int**
x **rem** y	Remainder	**int** * **int** → **int**
x **mod** y	Modulus	**int** * **int** → **int**
x**y	Power	**real** * **real** → **real**
x < y	Less than	**real** * **real** → **bool**
x > y	Greater than	**real** * **real** → **bool**
x <= y	Less or equal	**real** * **real** → **bool**
x >= y	Greater or equal	**real** * **real** → **bool**
x = y	Equal	**real** * **real** → **bool**
x <> y	Not equal	**real** * **real** → **bool**

The Character, Quote and Token Types

Operator	Name	Signature
c1 = c2	Equal	**char** * **char** → **bool**
c1 <> c2	Not equal	**char** * **char** → **bool**

Tuple Types

Operator	Name	Signature
`t1 = t2`	Equality	T * T → **bool**
`t1 <> t2`	Inequality	T * T → **bool**

Record Types

Operator	Name	Signature
`r.i`	Field select	A * Id → Ai
`r1 = r2`	Equality	A * A → **bool**
`r1 <> r2`	Inequality	A * A → **bool**
is_`A(r1)`	Is	Id * `MasterA` → **bool**

Union and Optional Types

Operator	Name	Signature
`t1 = t2`	Equality	A * A → **bool**
`t1 <> t2`	Inequality	A * A → **bool**

Set Types

Operator	Name	Signature
e **in set** s1	Membership	A * **set of** A → **bool**
e **not in set** s1	Not membership	A * **set of** A → **bool**
s1 **union** s2	Union	**set of** A * **set of** A → **set of** A
s1 **inter** s2	Intersection	**set of** A * **set of** A → **set of** A
s1 \ s2	Difference	**set of** A * **set of** A → **set of** A
s1 **subset** s2	Subset	**set of** A * **set of** A → **bool**
s1 **psubset** s2	Proper subset	**set of** A * **set of** A → **bool**
s1 = s2	Equality	**set of** A * **set of** A → **bool**
s1 <> s2	Inequality	**set of** A * **set of** A → **bool**
card s1	Cardinality	**set of** A → **nat**
dunion ss	Distributed union	**set of set of** A → **set of** A
dinter ss	Distributed intersection	**set of set of** A → **set of** A
power ss	Finite power set	**set of** A → **set of set of** A

Sequence Types

Operator	Name	Signature
hd l	Head	**seq1 of** A → A
tl l	Tail	**seq1 of** A → **seq of** A
len l	Length	**seq of** A → **nat**
elems l	Elements	**seq of** A → **set of** A
inds l	Indices	**seq of** A → **set of nat1**
l1 ^ l2	Concatenation	(**seq of** A) * (**seq of** A) → **seq of** A
reverse l	Reverse	**seq of** A → **seq of** A
conc ll	Distributed concatenation	**seq of seq of** A → **seq of** A
l ++ m	Sequence modification	**seq of** A * **map nat to** A → **seq of** A
l(i)	Sequence index	**seq of** A * **nat1** → A
l1 = l2	Equality	(**seq of** A) * (**seq of** A) → **bool**
l1 <> l2	Inequality	(**seq of** A) * (**seq of** A) → **bool**

Mapping Types

Operator	Name	Signature
dom m	Domain	(**map** A **to** B) → **set of** A
rng m	Range	(**map** A **to** B) → **set of** B
m1 **munion** m2	Map union	(**map** A **to** B) * (**map** A **to** B) → **map** A **to** B
m1 ++ m2	Override	(**map** A **to** B) * (**map** A **to** B) → **map** A **to** B
merge ms	Distributed merge	**set of** (**map** A **to** B) → **map** A **to** B
s <: m	Domain restrict to	(**set of** A) * (**map** A **to** B) → **map** A **to** B
s <-: m	Domain restrict by	(**set of** A) * (**map** A **to** B) → **map** A **to** B
m :> s	Range restrict to	(**map** A **to** B) * (**set of** B) → **map** A **to** B
m :-> s	Range restrict by	(**map** A **to** B) * (**set of** B) → **map** A **to** B
m(d)	Mapping apply	(**map** A **to** B) * A → B
inverse m	Map inverse	**inmap** A **to** B → **inmap** B **to** A
m1 = m2	Equality	(**map** A **to** B) * (**map** A **to** B) → **bool**
m1 <> m2	Inequality	(**map** A **to** B) * (**map** A **to** B) → **bool**

General Examples

if *predicate* **then** Expression **else** Expression

cases expression:
  (pattern list 1)−> Expression 1,
  (pattern list 2),
  (pattern list 3)−> Expression 2,
  others −> Expression 3
  end;

for all value **in set** setOfValues
  do Expression

dcl variable : *type* := Variable creation ;

let variable : *type* = Variable creation **in** Expression

let variable **in set** setOfValues **be st** pred(variable) **in** Expression

Comprehensions (Structure to Structure)

```
{element(var) | var in set setexpr & pred(var)}

[element(i) | i in set numsetexpr & pred(i)]
```

From Structure to Arbitrary Value

```
Select: set of nat -> nat
Select(s) ==
  let e in set s
  in
    e
pre s <> {}
```

From Structure to Single Value

```
SumSet: set of nat -> nat
SumSet(s) ==
  if s = {}
  then 0
  else let e in set s
       in
         e + SumSet(s\{e})
measure Card
```

From Structure to Single Boolean

```
forall p in set setOfP & pred(p)

exists p in set setOfP & pred(p)

exists1 p in set setOfP & pred(p)
```

VDM++ Class Example

```
class Person

types
  public String = seq of char;

values
  protected Name : seq of char = "Peter";

instance variables
  public nationality : seq of char:="Danish";
  yearOfBirth        : int;
  sex                : Male | Female;
  friends            : map String to Person;

operations
  public Person: int * (Male | Female) ==> 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;

thread
  while true do
    skip;

traces
  Mytrace: --regular expression with operation calls

end Person

class Male is subclass of Person
end Male
```

C
Related Projects

ADVANCE [28] aims to develop a unified tool-based framework for automated formal verification and simulation-based validation of embedded systems. The project is developing methods and tools for construction, refinement, composition and proof of formal models.

AVACS [25] investigates automatic verification and analysis of complex systems in particular embedded systems, using model checking.

CESAR [82] develops multi-viewpoint based development processes to improve requirements engineering and component-based engineering. A "Reference Technology Platform" has been developed making it possible to integrate or interoperate existing or emerging available technologies.

COCONUT [50] addresses the design and verification of modern embedded platforms by focusing on the formal specification of software and compilation, formal refinement and formal proof.

COMPASS [52] focuses on modelling and formal analysis systems of systems using varied techniques like simulation, proof and automated testing.

CREDO [58] focuses on the development and application of an integrated suite of tools for compositional modelling, testing, and validation of software for evolving networks of dynamically reconfigurable components.

DEPLOY [170] addresses the industrial deployment of formal methods for fault tolerant (DE) computing systems.

DESTECS [43] developed methods and tools that combine CT system models with DE controller models through co-simulation to allow multi-disciplinary modelling, including modelling of faults and fault tolerance mechanisms.

INTERESTED [68] focuses on creating an interoperable, open, reference toolchain for embedded systems. The project's main focus is on DE tools for graphical overview and code generation.

MEDEIA [178] aims to bridge the gap between engineering disciplines in the discrete engineering domain, by using containers that contain design models from various disciplines that can be seamlessly interconnected. Like the INTERESTED project, MEDEIA aims to connect tools in the DE domain.

Modelica [79] is a non-proprietary, object-oriented, equation-based language to conveniently model complex physical systems.

MODELISAR [76] has developed an open co-simulation interface for coupling models called the *Functional Mock-up Interface* (FMI).

MOGENTES [142] aims to significantly enhance testing and verification of dependable embedded systems by means of automated generation of efficient test cases.

PREDATOR [162] aims to advance the state of the art in the development of safety-critical embedded systems focusing on timing aspects.

Ptolemy [44] has studied modelling, simulation and design of concurrent, real-time embedded system, using a heterogeneous mixture of models of computation using an actor-oriented modelling approach.

QUASIMODO [165] aims to develop techniques and tools for model-driven design, analysis, testing and code-generation for embedded systems. A central problem is ensuring quantitative bounds on resource consumption. It focuses on formal notations for timed, probabilistic and hybrid systems that can be subjected to exhaustive state space analysis techniques such as model checking.

SPEEDS [157] have defined a semantic-based modelling method that supports the construction of complex embedded systems by composing heterogeneous subsystems, and which enables the sound integration of existing and new tools. The heterogeneous subsystem models are exported to a standard format to form the composed system, which can be simulated by a single simulation tool.

VERTIGO [190] aims to develop a systematic methodology to enhance modelling, integration and verification of architectures targeting embedded systems. It will use a co-simulation strategy that allows the correctness of the interaction between HW and SW to be assessed by simulation and assertion checking.

Bibliography

[1] 20-sim product homepage, 2012. `http://www.20sim.com/`.

[2] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods Review and analysis. Technical Report 478, VTT Technical Research Centre of Finland, 2002.

[3] Jean-Raymond Abrial. *The B Book – Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[4] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 761–768, New York, NY, USA, 2006. ACM Press.

[5] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[6] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

[7] Sten Agerholm. Translating Specifications in VDM-SL to PVS. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes of Computer Science*. Springer-Verlag, 1996.

[8] Sten Agerholm and Jacob Frost. Towards an Integrated CASE and Theorem Proving Tool for VDM-SL. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[9] Sten Agerholm and Peter Gorm Larsen. Modeling and Validating SAFER in VDM-SL. In Michael Holloway, editor, *Fourth NASA Langley Formal Methods Workshop*. NASA, September 1997.

[10] Manifesto for Agile Software Development, 2012. `http://agilemanifesto.org/`.

[11] Principles behind the Agile Manifesto, 2012. `http://agilemanifesto.org/principles.html`.

[12] Rajeev Alur, Costas A Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, PeiHsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *THEORETICAL COMPUTER SCIENCE*, 138:3–34, 1995.

[13] Rajeev Alur, Thomas A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22:181–201, 1996.

[14] Michael Andersen, René Elmstrøm, Poul Bøgh Lassen, and Peter Gorm Larsen. Making Specifications Executable – Using IPTES Meta-IV. *Microprocessing and Microprogramming*, 35(1-5):521–528, September 1992.

[15] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In *Proceedings of the 34th International Conference on Software Engineering (ISCE 2012)*, pages 1002–1011, June 2012.

[16] ANSI/ISA-84.01-1996. Application of safety instrumented systems for the process industries.

[17] Farhad Arbab. Reo: a Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[18] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[19] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *Z to B Conference / Nantes*, pages 334–354, 2005.

[20] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[21] Shahid Baqar. *Low-Cost PC-Based Hight-Fidelity Infrared Signature Modelling and Simulation*. PhD thesis, Cranfield University, Defence College of Management and Technology. Department of Aerospace, Power and Sensors, United Kingdom, July 2007.

[22] Luciano Baresi, Gianni Ferretti, Alberto Leva, and Matteo Rossi. Flexible logic-based co-simulation of modelica models. In *Industrial Informatics (INDIN), 2012 10th IEEE International Conference on*, pages 635 –640, july 2012.

[23] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfgang Schulte, and Herman Venter. Specification and Verification: The Spec# Experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[24] Building Controls Virtual Test Bed (BCVTB), 2012. `https://simulationresearch.lbl.gov/bcvtb`.

[25] Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog, Andreas Podelski, and Reinhard Wilhelm. SFB/TR 14 AVACS – automatic verification and analysis of complex systems. *it – Information Technology*, 49(2):118–126, 2007.

[26] Patrick Behm, Paul Benoit, Alain Faivre, and Jean marc Meynadier. METEOR : A successful application of B in a large project. In J.M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999*, volume 1709 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 1999.

[27] Hans Bekić, Dines Bjørner, Wolfgang Henhapl, Cliff B. Jones, and Peter Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, IBM Laboratory, Vienna, December 1974.

[28] Jens Bendisposto, Joy Clarke, John Colley, Andy Edmunds, Lukas Ladenberger, Michael Leuschel, Vitaly Savicks, and Harald Wiegard. D4.2 – Methods and Tools for Simulation and Testing I. Technical report, ADVANCE, FP7 project, 287563, September 2012.

[29] Bernhard K. Aichernig and Peter Gorm Larsen. A Proof Obligation Generator for VDM-SL. In John S. Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313 of *Lecture Notes in Computer Science*, pages 338–357. Springer-Verlag, September 1997. ISBN 3-540-63533-5.

[30] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.

[31] Dines Bjørner. From domain engineering via requirements to software. formal specification and design calculi. Technical report, Department of Information Technology, Software Systems Section, Technical University of Denmark, DK-2800 Lyngby, Denmark, 1997. Paper published in SOFSEM'97 Proceedings, Springer-Verlag, Lecture Notes in Computer Science. http://www.it.dtu.dk/~db/sofsem/sofsem.ps.

[32] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[33] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal versus agile: Survival of the fittest? *IEEE Computer*, 42(9):37–45, September 2009.

[34] Blender open-source 3D content creation suite, 2012. `http://www.blender.org/`.

[35] Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, Hans Olsson, and Antoine Viel. The Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *Proceedings of the 9th International Modelica Conference*, Munich, Germany, September 2012.

[36] Barry W. Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, January 2001.

[37] Barry W. Boehm and Philip N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1988.

[38] Michelle Boucher and David Houlihan. System design: New product development for mechatronics, January 2008.

[39] Felix Breitenecker, Niki Popper, Günther Zauner, Michaek Landsiedl, Matthias Rößler, Bernhard Heinzl, and Andreas Körner. Simulators For Physical Modelling - Classification and Comparison of Features /Revision 2010. In M. Snorek, Z. Buk, M. Cepek, and J. Drchal, editors, *Proceedings of EUROSIM 2010 - 7th Congress on Modelling and Simulation*, volume 2, pages 1051–1061, September 2010.

[40] Felix Breitenecker, Siegfried Wassertheurer, Nikolas Popper, and Gunter Zauner. Benchmarking of Simulation Systems–The ARGESIM Comparisons. In *Proceedings of the First Asia International Conference on Modelling & Simulation*, AMS '07, pages 568–573, Washington, DC, USA, 2007. IEEE Computer Society.

[41] Jan F. Broenink. *Computer-aided physical-systems modeling and simulation: a bond-graph approach*. PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, 1990.

[42] Jan F. Broenink. Modelling, Simulation and Analysis with 20-Sim. *Journal A Special Issue CACSD*, 38(3):22–25, 1997.

[43] Jan F. Broenink, Peter Gorm Larsen, Marcel Verhoef, Christian Kleijn, Dusko Jovanovic, Ken Pierce, and Frederik Wouters. Design Support and Tooling for Dependable Embedded Control Software. In *Proceedings of Serene 2010 International Workshop on Software Engineering for Resilient Systems*, pages 77–82. ACM, April 2010.

[44] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous System. In *Int. Journal of Computer Simulation*, 1994.

[45] Bullet Physics Library, 2012. `http://bulletphysics.org/wordpress/`.

[46] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proceedings of the International Workshop on Specification and validation of UML models for Real Time and embedded Systems, SVERTS2004*. UML2004, 2004.

[47] Eliza S. F. Cardozo, J. Benito F. Araujo Neto, Alexandre Barza, A. Cesar C. Franca, and Fabio Q. B. da Silva. SCRUM and Productivity in Software Projects: A Systematic Literature Review. In *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. Keele University, UK, 2010.

[48] Luca Carloni, Maria D. Di Benedetto, Alessandro Pinot, and Alberto Sangiovanni-Vincentelli. Modeling techniques, programming languages, design toolsets and interchange formats for hybrid systems. Deliverable DHS3, Project IST-2001-38314 COLUMBUS project - Design of Embedded Controllers for Safety Critical Systems, March 2004.

[49] Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002, volume 2374 of LNCS*, pages 231–255. Springer, 2002.

[50] Project presentation and COCONUT web site. Deliverable D6.1. Technical report, April 2008. `http://coconut-project.edalab.it/files/FP7-2007-IST-1-217069-Coconut-D6.1.pdf`.

[51] Joey W. Coleman, Kenneth G. Lausdahl, and Peter Gorm Larsen. Deliverable D3.4b: Co-simulation Semantics. Available online from `http://www.destecs.org/deliverables.html`, 2012.

[52] COMPASS: Comprehensive Modelling for Advanced Systems of Systems, 2011. `http://www.compass-research.eu/`.

[53] Rational S. Corporation. Rational Unified Process - Best Practices for Software Development Teams, 1998.

[54] Jeff Craighead, Jenny Burke, and Robin Murphy. Using the Unity Game Engine to Develop SARGE: A Case Study. In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*, September 2008.

[55] CSK. Development Guidelines for Real Time Systems using VDMTools. Technical report, CSK Systems, 2008.

[56] John Davis, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Y. Xiong. Ptolemy-II: Heterogeneous concurrent modeling and design in Java. Technical Memorandum UCB/ERL No. M99/40, University of California at Berkeley, July 1999.

[57] Timothy G. Davis. Aircraft fuel system simulation. In *Aerospace and Electronics Conference, 1990. NAECON 1990., Proceedings of the IEEE 1990 National*, pages 905 –911 vol.2, may 1990.

[58] Frank de Boer. CREDO: Modeling and Analysis of Evolutionary Structures for Distributed Services, 2007. `http://www.cwi.nl/projects/credo/`.

[59] Gjalt de Jong. A UML-Based Design Methodology for Real-Time and Embedded Systems. In *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE.02)*. IEEE, 2002.

[60] Vieri del Bianco, Dragan Stosic, and Joe Kiniry. Agile Formality: A "Mole" of Software Engineering Practices. In Stefan Gruner and Bernhard Rumpe, editors, *Proc. AM+FM'2010*, Lecture Notes in Informatics. Gesellschaft für Informatik, 2010. To appear.

[61] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. The PROSPER Toolkit. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Berlin, Germany, March/April 2000. Springer-Verlag, Lecture Notes in Computer Science volume 1785.

[62] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE special issue on CPS*, December 2011.

[63] Lionel Devauchelle, Peter Gorm Larsen, and Henrik Voss. PICGAL: Lessons Learnt from a Practical Use of Formal Specification to Develop a High Reliability Software. In *DASIA'97*. ESA, May 1997.

[64] Bruce Powel Douglass. *Doing Hard Time – Developing Real-Time Systems with UML Objects, Frameworks, and Patterns*. Addison-Wesley, 1999. ISBN 0-201-49837-5.

[65] Bruce Powel Douglass. *Real Time UML – Advances in the UML for Real-Time Systems*. Addison-Wesley, third edition, 2004.

[66] George Eleftherakis and Anthony J. Cowling. An agile formal development methodology. In *Proc. 1st. South-East European Workshop on Formal Methods, SEEFM'03*, pages 36–47. Springer-Verlag, 2003.

[67] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.

[68] Eric Bantegnie. INTERESTED: INTERoperable Embedded Systems Toolchain for Enhanced Design, prototyping and code generation, March 2011. `http://www.interested-ip.eu/files/INTERESTED-Final-Report.pdf`.

[69] John Fitzgerald. The Typed Logic of Partial Functions and the Vienna Development Method. In D. Bjørner and M. C. Henson, editors, *Logics of Specification Languages*, EATCS Monographs in Theoretical Computer Science, pages 427–461. Springer, 2007.

[70] John Fitzgerald and Peter Gorm Larsen. Triumphs and Challenges for the Industrial Application of Model-Oriented Formal Methods. In T. Margaria, A. Philippou, and B. Steffen, editors, *Proc. 2nd Intl. Symp. on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2007)*, 2007. Also Technical Report CS-TR-999, School of Computing Science, Newcastle University.

[71] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, Second edition, 2009. ISBN 0-521-62348-0.

[72] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[73] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, and Marcel Verhoef. A Formal Approach to Collaborative Modelling and Co-simulation for Embedded Systems. *To appear in Mathematical Structures in Computer Science*, 2012.

[74] John Fitzgerald, Peter Gorm Larsen, Ken Pierce, Marcel Verhoef, and Sune Wolff. Collaborative Modelling and Co-simulation in the Development of Dependable Embedded Systems. In D. Méry and S. Merz, editors, *IFM 2010, Integrated Formal Methods*, volume 6396 of *Lecture Notes in Computer Science*, pages 12–26. Springer-Verlag, October 2010.

[75] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, 43(2):3–11, February 2008.

[76] FMI Development Group. MODELISAR — Functional Mock-up Interface (FMI) `https://www.fmi-standard.org/`, October 2012. FMI Specification v2.0 Beta 4.

[77] Shanna-Shaye Forbes. Toward an error handling mechanism for timing errors with Java Pathfinder and Ptolemy II. Technical Report UCB/EECS-2010-123, EECS Department, University of California, Berkeley, September 2010.

[78] Sandford Friedenthal, Alan Moore, and Rick Steiner. *A Prictical Guide to SysML*. Morgan Kaufman OMG Press, Friendenthal, Sanford, First edition, 2008. ISBN 978-0-12-374379-4.

[79] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modelling and Simulation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 67–90. Springer-Verlag, 1998.

[80] Brigitte Fröhlich. *Towards Executability of Implicit Definitions*. PhD thesis, TU Graz, Institute of Software Technology, September 1998.

[81] Norbert E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[82] Gerhard Griessnig and Roland Mader and Thomas Peikenkamp and Bernhard Josko and Martin Törngren and Eric Armengaud. CESAR: Cost-Efficient Methods and Processes for Safety Relevant Embedded Systems. In *Embedded World 2010 - ARTEMIS Session*, 2010.

[83] Michael J.C. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*. Kluwer Academic Publishers, 1987.

[84] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison Wesley, 2000.

[85] Anthony Hall. Realising the benefits of formal methods. *Journal of Universal Computer Science*, 13(5):669–678, 2007.

[86] Ian J. Hayes, Michael A. Jackson, and Cliff B. Jones. Determining the specification of a control system from that of its environment. In Keijiro Araki, Stefani Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, September 2003.

[87] Ian J. Hayes and Cliff B. Jones. Specifications are not (Necessarily) Executable. *Software Engineering Journal*, pages 330–338, November 1989.

[88] Sharam Hekmatpour and Darrel Ince. *Software Prototyping, Formal Methods and VDM*. Addison-Wesley, 1988.

[89] Thomas A. Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, pages 1–15, 2006.

[90] Thomas A. Henzinger and Joseph Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 40(10):32–40, October 2007.

[91] Jon Holt and Simon Perry. *SysML for Systems Engineering*. IET, 2008.

[92] Jozef Hooman and Marcel Verhoef. Formal Semantics of a VDM Extension for Distributed Embedded Systems. In *Correctness, Concurrency and Compositionality*, LNCS Festscrift Series, 2008. Festscrift to honour professor Willem-Paul de Roever, Springer.

[93] Watts S. Humphrey. *Managing the Software Process*. SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts, USA, 1989.

[94] IEC 61508. Functional safety of electrical, electronic, programmable electronic safety-related systems. (`www.iec.ch/zone/fsafety`).

[95] IEEE. *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, 2008.

[96] Alexei Iliasov. Refinement Patterns for Rapid Development of Dependable Systems. In *EFTS '07: Proceedings of the 2007 Workshop on Engineering Fault Tolerant Systems*. ACM, 2007.

[97] ISO/IEC 13817-1: Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language, December 1996.

[98] Daniel Jackson and Jeanette Wing. Lightweight Formal Methods. *IEEE Computer*, 29(4):22–23, April 1996.

[99] Michael Jackson. *Problem Frames: Analysing and Structuring Software Development Problems*. Addison-Wesley, New York, 2001.

[100] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In *in Relational Methods in Computer Science*, pages 184–196. Springer Verlag, 1997.

[101] Juan F. Jimenez, Jose M. Giron-Sierra, Carlos C. Insaurralde, and Miguel A. Seminario. A simulation of aircraft fuel management system. *Simulation Modelling Practice and Theory*, 15(5):544 – 564, 2007.

[102] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 6(1):39–58, March 2007.

[103] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990. ISBN 0-13-880733-7.

[104] Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Deriving Specifications for Systems That Are Connected to the Physical World. In *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[105] Cliff B. Jones, Kevin Jones, Peter Lindsay, and Richard Moore, editors. *mural: A Formal Development Support System*. Springer-Verlag, 1991. ISBN 3-540-19651-X.

[106] Peter W.V. Jørgensen. Evaluation of Development Process and Methodology for co-models. Master's thesis, Aarhus University/Engineering College of Aarhus, December 2012.

[107] Dean Karnopp and Ronald C. Rosenberg. *Analysis and simulation of multiport systems: the bond graph approach to physical system dynamic*. MIT Press, Cambridge, MA, USA, 1968.

[108] Richard A. Kemmerer. Integrating Formal Methods into the Development Process. *IEEE Software*, pages 37–50, September 1990.

[109] Kepler project. Your Science. Enabled., 2012. `https://kepler-project.org/`.

[110] Jeff Kramer. Is Abstraction the Key to Computing? *Communications of the ACM*, 50(4):37–42, 2007.

[111] Taro Kurita, Miki Chiba, and Yasumasa Nakatsugawa. Application of a Formal Specification Language in the Development of the "Mobile FeliCa" IC Chip Firmware for Embedding in Mobile Phone. In J. Cuellar, T. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods*, Lecture Notes in Computer Science, pages 425–429. Springer-Verlag, May 2008.

[112] Taro Kurita and Yasumasa Nakatsugawa. The Application of VDM++ to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, 3(2-3):343–355, October 2009.

[113] Kevin Lano. Logic Specification of Reactive and Real-time Systems. *Journal of Logic and Computation*, 8(5):679–711, 1998.

[114] Peter Gorm Larsen. *Towards Proof Rules for VDM-SL*. PhD thesis, Technical University of Denmark, Department of Computer Science, March 1995. ID-TR:1995-160.

[115] Peter Gorm Larsen. Ten Years of Historical Development: "Bootstrapping" VDMTools. *Journal of Universal Computer Science*, 7(8):692–709, 2001.

[116] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture Initiative – Integrating Tools for VDM. *ACM Software Engineering Notes*, 35(1), January 2010.

[117] Peter Gorm Larsen and John Fitzgerald. Recent Industrial Applications of Formal Methods in Japan. In P. Boca and J. P. Bowen, editors, *Proc. BCS-FACS Workshop on Formal Methods in Industry*. British Computer Society, 2008.

[118] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[119] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Methods for the Development of Distributed Real-Time Embedded Systems using VDM. *Intl. Journal of Software and Informatics*, 3(2-3), October 2009.

[120] Peter Gorm Larsen, John Fitzgerald, and Sune Wolff. Are formal methods ready for agility? a reality check. In Stefan Gruner and Bernhard Rumpe, editors, *2nd International Workshop on Formal Methods and Agile Methods*, pages 13–25. Lecture Notes in Informatics, September 2010. ISSH 1617-5468.

[121] Peter Gorm Larsen and Bo Stig Hansen. Semantics for underdetermined expressions. *Formal Aspects of Computing*, 8(1):47–66, January 1996.

[122] Peter Gorm Larsen and Poul Bøgh Lassen. An Executable Subset of Meta-IV with Loose Specification. In *VDM '91: Formal Software Development Methods*. VDM Europe, Springer-Verlag, March 1991.

[123] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial Testing for VDM. In *8th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2010*, September 2010.

[124] Peter Gorm Larsen, Kenneth Lausdahl, Augusto Ribeiro, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, May 2010.

[125] Peter Gorm Larsen and Wiesław Pawłowski. The Formal Semantics of ISO VDM-SL. *Computer Standards and Interfaces*, 17(5–6):585–602, September 1995.

[126] Peter Gorm Larsen, Sune Wolff, Nick Battle, John Fitzgerald, and Ken Pierce. Development Process of Distributed Embedded Systems using VDM. Technical Report TR-2010-02, The Overture Open Source Initiative, April 2010.

[127] Kenneth Lausdahl, Peter Gorm Larsen, and Nick Battle. A Deterministic Interpreter Simulating a Distributed Real Time System using VDM. In *ICFEM 2011*, October 2011.

[128] Kenneth Lausdahl, Hans Kristian Agerlund Lintrup, and Peter Gorm Larsen. Connecting UML and VDM++ with Open Tool Support. In *Formal Methods 09*. Springer-Verlag, November 2009. LNCS-5850.

[129] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.

[130] Edward A. Lee, Xiaojun Liu, and Steve Neuendorffer. Classes and inheritance in actor-oriented design. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(4):29:1–29:26, 2009.

[131] Edward A. Lee and Steve Neuendorffer. MoML - A Modeling Markup Language in XML - Version 0.4, 2000.

[132] Shaoying Liu. An Approach to Applying SOFL for Agile Process and Its Application in Developing a Test Support Tool. *Innovations in Systems and Software Engineering*, 6:137–143, December 2010.

[133] Shaoying Liu and Yong Sun. Structured Methodology + Object-Oriented Methodology + Formal Methods: Methodology of SOFL. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems*, pages 137–144, Ft. Landerdale, Florida, U.S.A., November 1995. IEEE Press.

[134] LMS Engineering Innovation. AMESim: Advanced Modeling Environment for performing Simulations of engineering systems `http://www.lmsintl.com/imagine-lab-amesim-rev-11`, October 2012. AMESim Revision 11 product home page.

[135] Hugo Daniel Macedo, Peter Gorm Larsen, and John Fitzgerald. Incremental Development of a Distributed Real-Time Model of a Cardiac Pacing System using VDM. In Jorge Cuellar, Tom Maibaum, and Kaisa Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 2008.

[136] Martín López-Nores and José J. Pazos-Arias and Jorge García-Duque and others. Bringing the Agile Philosophy to Formal Specification Settings. *International Journal of Software Engineering and Knowledge Engineering*, 16(6):951–986, 2006.

[137] MathWorks. MATLAB official website, October 2012. `http://www.mathworks.com`.

[138] Steven P. Miller, Elise A. Anderson, Lucas G. Wagner, Michael W. Whalen, and Mats P.E. Heimdahl. Formal Verification of Flight Critical Software. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, San Francisco, August 2005. AIAA.

[139] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. Software model checking takes off. *Commun. ACM*, 53:58–64, February 2010.

[140] MIRABILIS design — Simulation Of Electronics, 2012. `http://mirabilisdesign.com`.

[141] MLDesign Technologies. MLDesigner, 2012. `http://www.mldesigner.com`.

[142] MOGENTES: Model-based Generation of Tests for Dependable Embedded Systems, 2012. `http://www.mogentes.eu/`.

[143] David Holst Møller and Christian Rane Paysen Thillermann. Using Eclipse for Exploring an Integration Architecture for VDM. Master's thesis, Aarhus University/Engineering College of Aarhus, June 2009.

[144] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, London, UK, 1990.

[145] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring Timing Properties Using VDM++ on an Industrial Application. In J.C. Bicarregui and J.S. Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, September 2000. Available at www.vdmportal.org.

[146] Claus Ballegaard Nielsen, Kenneth Lausdahl, and Peter Gorm Larsen. Combining VDM with Executable Code. In *ABZ 2012, LNCS 7316*, pages 266–279, Heidelberg, 2012. Springer.

[147] Nan Niu and Steve Easterbrook. On the use of model checking in verification of evolving agile software frameworks: An exploratory case study. In *3rd International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2005)*, pages 115–117, Miami, Florida, USA, May 2005. INSTICC Press.

[148] GNU Octave, 2012. `http://www.gnu.org/software/octave/`.

[149] ODE: Open Dynamics Engine, 2012. `http://ode.org/`.

[150] OMG. OMG System Modeling Language (SysML) Formal ver. 1.3. `http://www.omg.org/spec/SysML/`, June 2012.

[151] Open Source Initiative OSI. The BSD License, 2012. `http://www.opensource.org/licenses/bsd-license.php`.

[152] Oracle. Javadoc Tool, 2012. `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`.

[153] Jonathan S. Ostroff, David Makalsky, and Richard F. Paige. Agile specification-driven development. In J. Eckstein and H. Baumeister, editors, *XP 2004*, volume 3092 of *Lecture Notes in Computer Science*, pages 104–112. Springer, 2004.

[154] Overture-Core-Team. Overture Web site. `http://www.overturetool.org`, 2007.

[155] VDM-RT examples for Overture, 2012. `http://overturetool.org/?q=node/15`.

[156] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25:41–61, 1995.

[157] Roberto Passerone, Werner Damm, Imene Ben Hafaiedh, Susanne Graf, Alberto Ferrari, Leonardo Mangeruca, Albert Benveniste, Bernhard Josko, Thomas Peikenkamp,

Daniela Cancila, Arnaud Cuccuru, Sebastien Gerard, Francois Terrier, and Alberto Sangiovanni-Vincentelli. Metamodels in europe: Languages, tools, and applications. *Design Test of Computers, IEEE*, 26(3):38–53, may–june 2009.

[158] PhysX: nVidia physics engine, 2012. `https://developer.nvidia.com/physx`.

[159] Jos Luis Pino, Soonhoi Ha, Edward A. Lee, and Joseph T. Buck. Software Synthesis for DSP Using Ptolemy. *Journal of VLSI Signal Processing*, 9:7–21, 1993.

[160] Nico Plat and Peter Gorm Larsen. An Overview of the ISO/VDM-SL Standard. *Sigplan Notices*, 27(8):76–82, August 1992.

[161] Player/Stage project, 2012. `http://sourceforge.net/projects/playerstage/`.

[162] PREDATOR: Design for predictability and efficiency, 2012. `http://www.predator-project.eu/`.

[163] Armand Puccetti and Jean Yves Tixadou. Application of VDM-SL to the Development of the SPOT4 Programming Messages Generator. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 127–137, September 1999.

[164] Junjing Qian. Co-Design of Embedded Systems: an Aircraft Fuel Tank (to appear). Master's thesis, Newcastle University, UK, September 2012.

[165] QUASIMODO: Quantitative System Properties in Model-Driven-Design of Embedded Systems, 2012. `http://www.quasimodo.aau.dk/`.

[166] Lars Rosenberg Randleff. *Decision Support System for Fighter Pilots*. PhD thesis, Technical University of Denmark, Informatics and Mathematical Modelling, Denamrk, 2007. "IMM-PHD: ISSN 0909-3192".

[167] Augusto Ribeiro. An Extended Proof Obligation Generator for VDM++/OML. Master's thesis, Minho University with exchange to Engineering College of Arhus, July 2008.

[168] Augusto Ribeiro, Kenneth Lausdahl, and Peter Gorm Larsen. Run-Time Validation of Timing Constraints for VDM-RT Models. In *9th Overture Workshop, June 2011, Limerick, Ireland*, 2011.

[169] Erkuden Rios, Teodora Bozheva, Aitor Bediaga, and Nathalie Guilloreau. MDD maturity model: a roadmap for introducing model-driven development. ECMDA-FA'06, pages 78–89, Berlin, Heidelberg, 2006. Springer-Verlag.

[170] Alexander Romanovsky and Martyn Thomas (Eds.). *Industrial deployment of system engineering methods providing high dependability and productivity*. Lecture Notes in Computer Science. Springer-Verlag, 2012.

[171] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009.

[172] Hossein Saiedian. An invitation to formal methods. *IEEE Computer*, 29(4):16–30, April 1996. Roundtable with contributions from experts.

[173] D. Curtis Schleher. *Introduction to Electronic Warfare*. Artech House, 685 Canton Street, Norwood, MA 02062, first edition, 1986.

[174] Ken Schwaber. *Agile Project Management with Scrum*. Prentice Hall, 2004. ISBN: 073561993X.

[175] Paul R. Smith and Peter Gorm Larsen. Applications of VDM in Banknote Processing. In John S. Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice: Proc. First VDM Workshop 1999*, September 1999. Available at www.vdmportal.org.

[176] Fritz Solms and Dawid Loubser. URDAD as a semi-formal approach to analysis and design. *Innovations in Systems and Software Engineering*, 6:155–162, 2010.

[177] Ian Sommerville, Ray Welland, Stuart Potter, and John Smart. The ECLIPSE User Interface. *Software - Practice and Experience*, 19(4):371–391, April 1989.

[178] Thomas Strasser, Christoph Sunder, and Antonio Valentini. Model-driven embedded systems design environment for the industrial automation sector. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 1120–1125, july 2008.

[179] Tigris Subclipse: Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE, 2012. `http://subclipse.tigris.org/`.

[180] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An Incremental Methodology for Developing Formal Models. *ACM Transactions on Design Automation of Electronic Systems*, 10(4):589–609, October 2005.

[181] Sebastian Uchitel, Jeff Kramer, and Jeff Magee. Incremental Elaboration of Scenario-Based Specifications and Behavior Models Using Implied Scenarios. *ACM Transactions on Software Engineering and Methodology*, 13(1):37–85, January 2004.

[182] Unity Technologies. Unity3D Game Engine, 2012. `http://unity3d.com/`.

[183] Manuel van den Berg, Marcel Verhoef, and Mark Wigmans. Formal Specification of an Auctioning System Using VDM++ and UML – an Industrial Usage Report. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 85–93, September 1999.

[184] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[185] Marcel Verhoef, Peter Gorm Larsen, and Jozef Hooman. Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, Lecture Notes in Computer Science 4085, pages 147–162. Springer-Verlag, 2006.

[186] Marcel Verhoef, Peter Visser, Jozef Hooman, and Jan Broenink. Co-simulation of Real-time Embedded Control Systems. In Jim Davies and Jeremy Gibbons, editors, *Integrated Formal Methods: Proc. 6th. Intl. Conference*, Lecture Notes in Computer Science 4591, pages 639–658. Springer-Verlag, July 2007.

[187] Rajesh Verma, Ashu Gupta, and Kawaljeet Singh. A Critical Evaluation and Comparison of Four Manufacturing Simulation Softwares. *Kathmandu University Journal of Science, Eengineering and Technology*, 5(1):104–120, January 2009.

[188] Sander Vermolen. Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department, August 2007.

[189] Sander Vermolen, Jozef Hooman, and Peter Gorm Larsen. Automating Consistency Proofs of VDM++ Models using HOL. In *Proceedings of the 25th Symposium on Applied Computing (SAC 2010)*, Sierre, Switzerland, March 2010. ACM.

[190] VERTIGO: Verification and Validation of Embedded System Design Workbench, 2012. `http://vertigo-project.edalab.it/`.

[191] Ana Fernández Vilas, José J. Pazos Arias, Rebeca P. Diaz Redondo, and A. Belén Barragáns Martinez. Formalizing Incremental Design in Real-time Area: SCTL/MUS-T. In *Proceedings of the 26 th Annual International Computer Software and Applications Conference (COMPSAC'02)*. IEEE, 2002.

[192] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *AUTOMATED SOFTWARE ENGINEERING JOURNAL*, pages 3–12, 2000.

[193] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In *FME 2003: International Symposium of Formal Methods Europe Proceedings. Volume 2805 of Lecture Notes in Computer Science*, pages 133–153. Springer-Verlag, 2003.

[194] Alan Wassyng and Mark Lawford. Software tools for safety-critical software development. *Int. J. Softw. Tools Technol. Transf.*, 8(4):337–354, August 2006.

[195] Sune Wolff. Formalising Concurrent and Distributed Design Patterns with VDM. November 2009. The 7th Overture workshop at FM'09.

[196] Sune Wolff. Methodological Guidelines for Collaborative Modelling — Managing Heterogeneous System Complexity. *International Journal of Software and Systems Modeling - SoSym*, 2012. Submitted — under review.

[197] Sune Wolff. Scrum Goes Formal: Agile Methods for Safety-Critical Systems. In *ICSE 2012: Proceedings of the 34th International Conference on Software Engineering*, pages 23–29, June 2012. Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches, FormSERA 2012.

[198] Sune Wolff. Using Formal Methods for Self-defense System for Fighter Aircraft — An Industrial Experience Report. *International Journal of Empirical Software Engineering*, 2012. Submitted — under review.

[199] Sune Wolff, Peter Gorm Larsen, and Tammy Noergaard. Development Process for Multi-Disciplinary Embedded Control Systems. In *EuroSim 2010*. EuroSim, September 2010.

[200] Sune Wolff, Ken Pierce, and Patricia Derler. Multi-domain Modelling in DESTECS and Ptolemy — a Tool Comparison. *SIMULATION: Transactions of The Society for Modeling and Simulation International (journal)*, 2012. Submitted — under review.

[201] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, October 2009.